Numba Documentation

Release 0.51.0-py3.7-linux-x86_64.egg

Anaconda

Aug 14, 2020

This is the Numba documentation. Unless you are already acquainted with Numba, we suggest you start with the *User manual*.

User Manual

1.1 A ~5 minute guide to Numba

Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. The most common way to use Numba is through its collection of decorators that can be applied to your functions to instruct Numba to compile them. When a call is made to a Numba decorated function it is compiled to machine code "just-in-time" for execution and all or part of your code can subsequently run at native machine code speed!

Out of the box Numba works with the following:

- OS: Windows (32 and 64 bit), OSX and Linux (32 and 64 bit)

- Architecture: x86, x86_64, ppc64le. Experimental on armv7l, armv8l (aarch64).

- GPUs: Nvidia CUDA. Experimental on AMD ROC.

- CPython

- NumPy 1.15 - latest

1.1.1 How do I get it?

Numba is available as a conda package for the Anaconda Python distribution:

```
$ conda install numba
```

Numba also has wheels available:

```
$ pip install numba
```

Numba can also be *compiled from source*, although we do not recommend it for first-time Numba users.

Numba is often used as a core package so its dependencies are kept to an absolute minimum, however, extra packages can be installed as follows to provide additional functionality:

- `scipy` - enables support for compiling `numpy.linalg` functions.

- `colorama` - enables support for color highlighting in backtraces/error messages.

- `pyyaml` - enables configuration of Numba via a YAML config file.

- `icc_rt` - allows the use of the Intel SVML (high performance short vector math library, x86_64 only). Installation instructions are in the *performance tips*.

1.1.2 Will Numba work for my code?

This depends on what your code looks like, if your code is numerically orientated (does a lot of math), uses NumPy a lot and/or has a lot of loops, then Numba is often a good choice. In these examples we'll apply the most fundamental of Numba's JIT decorators, `@jit`, to try and speed up some functions to demonstrate what works well and what does not.

Numba works well on code that looks like this:

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

It won't work very well, if at all, on code that looks like this:

```python
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1                        # Numba doesn't understand what this is
    return df.cov()                # or this!

print(use_pandas(x))
```

Note that Pandas is not understood by Numba and as a result Numba would simply run this code via the interpreter but with the added cost of the Numba internal overheads!

1.1.3 What is `nopython` mode?

The Numba `@jit` decorator fundamentally operates in two compilation modes, `nopython` mode and `object` mode. In the `go_fast` example above, `nopython=True` is set in the `@jit` decorator, this is instructing Numba to operate in `nopython` mode. The behaviour of the `nopython` compilation mode is to essentially compile the decorated function so that it will run entirely without the involvement of the Python interpreter. This is the recommended and best-practice way to use the Numba `jit` decorator as it leads to the best performance.

Should the compilation in `nopython` mode fail, Numba can compile using `object mode`, this is a fall back mode for the `@jit` decorator if `nopython=True` is not set (as seen in the `use_pandas` example above). In this mode Numba will identify loops that it can compile and compile those into functions that run in machine code, and it will run the rest of the code in the interpreter. For best performance avoid using this mode!

1.1.4 How to measure the performance of Numba?

First, recall that Numba has to compile your function for the argument types given before it executes the machine code version of your function, this takes time. However, once the compilation has taken place Numba caches the machine code version of your function for the particular types of arguments presented. If it is called again the with same types, it can reuse the cached version instead of having to compile again.

A really common mistake when measuring performance is to not account for the above behaviour and to time code once with a simple timer that includes the time taken to compile your function in the execution time.

For example:

```python
from numba import jit
import numpy as np
import time

x = np.arange(100).reshape(10, 10)

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in machine code
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE EXECUTION TIME!
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %s" % (end - start))

# NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %s" % (end - start))
```

This, for example prints:

```
Elapsed (with compilation) = 0.33030009269714355
Elapsed (after compilation) = 6.67572021484375e-06
```

A good way to measure the impact Numba JIT has on your code is to time execution using the timeit module functions, these measure multiple iterations of execution and, as a result, can be made to accommodate for the compilation time in the first execution.

As a side note, if compilation time is an issue, Numba JIT supports *on-disk caching* of compiled functions and also has an *Ahead-Of-Time* compilation mode.

1.1.5 How fast is it?

Assuming Numba can operate in `nopython` mode, or at least compile some loops, it will target compilation to your specific CPU. Speed up varies depending on application but can be one to two orders of magnitude. Numba has a *performance guide* that covers common options for gaining extra performance.

1.1.6 How does Numba work?

Numba reads the Python bytecode for a decorated function and combines this with information about the types of the input arguments to the function. It analyzes and optimizes your code, and finally uses the LLVM compiler library to generate a machine code version of your function, tailored to your CPU capabilities. This compiled version is then used every time your function is called.

1.1.7 Other things of interest:

Numba has quite a few decorators, we've seen `@jit`, but there's also:

- `@njit` - this is an alias for `@jit(nopython=True)` as it is so commonly used!
- `@vectorize` - produces NumPy `ufunc` s (with all the `ufunc` methods supported). *Docs are here*.
- `@guvectorize` - produces NumPy generalized `ufunc` s. *Docs are here*.
- `@stencil` - declare a function as a kernel for a stencil like operation. *Docs are here*.
- `@jitclass` - for jit aware classes. *Docs are here*.
- `@cfunc` - declare a function for use as a native call back (to be called from C/C++ etc). *Docs are here*.
- `@overload` - register your own implementation of a function for use in nopython mode, e.g. `@overload(scipy.special.j0)`. *Docs are here*.

Extra options available in some decorators:

- `parallel = True` - *enable* the *automatic parallelization* of the function.
- `fastmath = True` - enable *fast-math* behaviour for the function.

ctypes/cffi/cython interoperability:

- `cffi` - The calling of *CFFI* functions is supported in `nopython` mode.
- `ctypes` - The calling of *ctypes* wrapped functions is supported in `nopython` mode. .
- Cython exported functions *are callable*.

GPU targets:

Numba can target Nvidia CUDA and (experimentally) AMD ROC GPUs. You can write a kernel in pure Python and have Numba handle the computation and data movement (or do this explicitly). Click for Numba documentation on *CUDA* or *ROC*.

1.2 Overview

Numba is a compiler for Python array and numerical functions that gives you the power to speed up your applications with high performance functions written directly in Python.

Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba's main features are:

- *on-the-fly code generation* (at import time or runtime, at the user's preference)

- native code generation for the CPU (default) and *GPU hardware*

- integration with the Python scientific software stack (thanks to Numpy)

Here is how a Numba-optimized function, taking a Numpy array as argument, might look like:

```python
@numba.jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```

1.3 Installation

1.3.1 Compatibility

Numba is compatible with Python 3.6 or later, and Numpy versions 1.15 or later.

Our supported platforms are:

- Linux x86 (32-bit and 64-bit)

- Linux ppcle64 (POWER8)

- Windows 7 and later (32-bit and 64-bit)

- OS X 10.9 and later (64-bit)

- NVIDIA GPUs of compute capability 2.0 and later

- AMD ROC dGPUs (linux only and not for AMD Carrizo or Kaveri APU)

- ARMv7 (32-bit little-endian, such as Raspberry Pi 2 and 3)

- ARMv8 (64-bit little-endian, such as the NVIDIA Jetson)

Automatic parallelization with @jit is only available on 64-bit platforms.

1.3.2 Installing using conda on x86/x86_64/POWER Platforms

The easiest way to install Numba and get updates is by using `conda`, a cross-platform package manager and software distribution maintained by Anaconda, Inc. You can either use Anaconda to get the full stack in one download, or Miniconda which will install the minimum packages required for a conda environment.

Once you have conda installed, just type:

```
$ conda install numba
```

or:

```
$ conda update numba
```

Note that Numba, like Anaconda, only supports PPC in 64-bit little-endian mode.

To enable CUDA GPU support for Numba, install the latest graphics drivers from NVIDIA for your platform. (Note that the open source Nouveau drivers shipped by default with many Linux distributions do not support CUDA.) Then install the `cudatoolkit` package:

```
$ conda install cudatoolkit
```

You do not need to install the CUDA SDK from NVIDIA.

1.3.3 Installing using pip on x86/x86_64 Platforms

Binary wheels for Windows, Mac, and Linux are also available from PyPI. You can install Numba using `pip`:

```
$ pip install numba
```

This will download all of the needed dependencies as well. You do not need to have LLVM installed to use Numba (in fact, Numba will ignore all LLVM versions installed on the system) as the required components are bundled into the llvmlite wheel.

To use CUDA with Numba installed by *pip*, you need to install the CUDA SDK from NVIDIA. Please refer to *Setting CUDA Installation Path* for details. Numba can also detect CUDA libraries installed system-wide on Linux.

1.3.4 Enabling AMD ROCm GPU Support

The ROCm Platform allows GPU computing with AMD GPUs on Linux. To enable ROCm support in Numba, conda is required, so begin with an Anaconda or Miniconda installation with Numba 0.40 or later installed. Then:

1. Follow the ROCm installation instructions.

2. Install `roctools` conda package from the `numba` channel:

   ```
   $ conda install -c numba roctools
   ```

See the roc-examples repository for sample notebooks.

1.3.5 Installing on Linux ARMv7 Platforms

Berryconda is a conda-based Python distribution for the Raspberry Pi. We are now uploading packages to the `numba` channel on Anaconda Cloud for 32-bit little-endian, ARMv7-based boards, which currently includes the Raspberry Pi 2 and 3, but not the Pi 1 or Zero. These can be installed using conda from the `numba` channel:

```
$ conda install -c numba numba
```

Berryconda and Numba may work on other Linux-based ARMv7 systems, but this has not been tested.

1.3.6 Installing on Linux ARMv8 (AArch64) Platforms

We build and test conda packages on the NVIDIA Jetson TX2, but they are likely to work for other AArch64 platforms. (Note that while the Raspberry Pi CPU is 64-bit, Raspbian runs it in 32-bit mode, so look at *Installing on Linux ARMv7 Platforms* instead.)

Conda-forge support for AArch64 is still quite experimental and packages are limited, but it does work enough for Numba to build and pass tests. To set up the environment:

- Install conda4aarch64. This will create a minimal conda environment.

- Add the `c4aarch64` and `conda-forge` channels to your conda configuration:

```
$ conda config --add channels c4aarch64
$ conda config --add channels conda-forge
```

- Then you can install Numba from the `numba` channel:

```
$ conda install -c numba numba
```

On CUDA-enabled systems, like the Jetson, the CUDA toolkit should be automatically detected in the environment.

1.3.7 Installing from source

Installing Numba from source is fairly straightforward (similar to other Python packages), but installing llvmlite can be quite challenging due to the need for a special LLVM build. If you are building from source for the purposes of Numba development, see *Build environment* for details on how to create a Numba development environment with conda.

If you are building Numba from source for other reasons, first follow the llvmlite installation guide. Once that is completed, you can download the latest Numba source code from Github:

```
$ git clone git://github.com/numba/numba.git
```

Source archives of the latest release can also be found on PyPI. In addition to `llvmlite`, you will also need:

- A C compiler compatible with your Python installation. If you are using Anaconda, you can use the following conda packages:

  - Linux `x86`: `gcc_linux-32` and `gxx_linux-32`

  - Linux `x86_64`: `gcc_linux-64` and `gxx_linux-64`

  - Linux `POWER`: `gcc_linux-ppc64le` and `gxx_linux-ppc64le`

  - Linux `ARM`: no conda packages, use the system compiler

  - Mac OSX: `clang_osx-64` and `clangxx_osx-64` or the system compiler at `/usr/bin/clang` (Mojave onwards)

  - Windows: a version of Visual Studio appropriate for the Python version in use

- NumPy

Then you can build and install Numba from the top level of the source tree:

```
$ python setup.py install
```

Build time environment variables and configuration of optional components

Below are environment variables that are applicable to altering how Numba would otherwise build by default along with information on configuration options.

NUMBA_DISABLE_OPENMP (default: not set)
> To disable compilation of the OpenMP threading backend set this environment variable to a non-empty string when building. If not set (default):

- For Linux and Windows it is necessary to provide OpenMP C headers and runtime libraries compatible with the compiler tool chain mentioned above, and for these to be accessible to the compiler via standard flags.

- For OSX the conda packages `llvm-openmp` and `intel-openmp` provide suitable C headers and libraries. If the compilation requirements are not met the OpenMP threading backend will not be compiled

NUMBA_DISABLE_TBB (default: not set)
   To disable the compilation of the TBB threading backend set this environment variable to a non-empty string when building. If not set (default) the TBB C headers and libraries must be available at compile time. If building with `conda build` this requirement can be met by installing the `tbb-devel` package. If not building with `conda build` the requirement can be met via a system installation of TBB or through the use of the `TBBROOT` environment variable to provide the location of the TBB installation. For more information about setting `TBBROOT` see the Intel documentation.

1.3.8 Dependency List

Numba has numerous required and optional dependencies which additionally may vary with target operating system and hardware. The following lists them all (as of July 2020).

- Required build time:

    - `setuptools`

    - `numpy`

    - `llvmlite`

    - Compiler toolchain mentioned above

- Required run time:

    - `setuptools`

    - `numpy`

    - `llvmlite`

- Optional build time:

    See *Build time environment variables and configuration of optional components* for more details about additional options for the configuration and specification of these optional components.

    - `llvm-openmp` (OSX) - provides headers for compiling OpenMP support into Numba's threading backend

    - `intel-openmp` (OSX) - provides OpenMP library support for Numba's threading backend.

    - `tbb-devel` - provides TBB headers/libraries for compiling TBB support into Numba's threading backend

- Optional runtime are:

    - `scipy` - provides cython bindings used in Numba's `np.linalg.*` support

    - `tbb` - provides the TBB runtime libraries used by Numba's TBB threading backend

    - `jinja2` - for "pretty" type annotation output (HTML) via the `numba` CLI

    - `cffi` - permits use of CFFI bindings in Numba compiled functions

    - `intel-openmp` - (OSX) provides OpenMP library support for Numba's OpenMP threading backend

    - `ipython` - if in use, caching will use IPython's cache directories/caching still works

- – `pyyaml` - permits the use of a `.numba_config.yaml` file for storing per project configuration options
- – `colorama` - makes error message highlighting work
- – `icc_rt` - (numba channel) allows Numba to use Intel SVML for extra performance
- – `pygments` - for "pretty" type annotation
- – `gdb` as an executable on the `$PATH` - if you would like to use the gdb support
- – Compiler toolchain mentioned above, if you would like to use `pycc` for Ahead-of-Time (AOT) compilation
- – `r2pipe` - required for assembly CFG inspection.
- – `radare2` as an executable on the `$PATH` - required for assembly CFG inspection. See here for information on obtaining and installing.
- – `graphviz` - for some CFG inspection functionality.
- – `pickle5` - provides Python 3.8 pickling features for faster pickling in Python 3.6 and 3.7.
- • To build the documentation:
  - – `sphinx`
  - – `pygments`
  - – `sphinx_rtd_theme`
  - – `numpydoc`
  - – `make` as an executable on the `$PATH`

1.3.9 Checking your installation

You should be able to import Numba from the Python prompt:

```
$ python
Python 3.8.1 (default, Jan 8  2020, 16:15:59)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numba
>>> numba.__version__
'0.48.0'
```

You can also try executing the `numba --sysinfo` (or `numba -s` for short) command to report information about your system capabilities. See *Command line interface* for further information.

```
$ numba -s
System info:
--------------------------------------------------------------------------------
__Time Stamp__
2018-08-28 15:46:24.631054

__Hardware Information__
Machine                           : x86_64
CPU Name                          : haswell
CPU Features                      :
aes avx avx2 bmi bmi2 cmov cx16 f16c fma fsgsbase lzcnt mmx movbe pclmul popcnt
rdrnd sse sse2 sse3 sse4.1 sse4.2 ssse3 xsave xsaveopt
```

(continues on next page)

```
__OS Information__
Platform                               : Darwin-17.6.0-x86_64-i386-64bit
Release                                : 17.6.0
System Name                            : Darwin
Version                                : Darwin Kernel Version 17.6.0: Tue May  8␣
↪15:22:16 PDT 2018; root:xnu-4570.61.1~1/RELEASE_X86_64
OS specific info                       : 10.13.5   x86_64


__Python Information__
Python Compiler                        : GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_
↪401/final)
Python Implementation                  : CPython
Python Version                         : 2.7.15
Python Locale                          : en_US UTF-8


__LLVM information__
LLVM version                           : 6.0.0


__CUDA Information__
Found 1 CUDA devices
id 0        GeForce GT 750M                                  [SUPPORTED]
                    compute capability: 3.0
                         pci device id: 0
                            pci bus id: 1
```

(output truncated due to length)

1.4 Compiling Python code with `@jit`

Numba provides several utilities for code generation, but its central feature is the *numba.jit()* decorator. Using this decorator, you can mark a function for optimization by Numba's JIT compiler. Various invocation modes trigger differing compilation options and behaviours.

1.4.1 Basic usage

Lazy compilation

The recommended way to use the `@jit` decorator is to let Numba decide when and how to optimize:

```python
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

In this mode, compilation will be deferred until the first function execution. Numba will infer the argument types at call time, and generate optimized code based on this information. Numba will also be able to compile separate specializations depending on the input types. For example, calling the `f()` function above with integer or complex numbers will generate different code paths:

```
>>> f(1, 2)
3
>>> f(1j, 2)
(2+1j)
```

Eager compilation

You can also tell Numba the function signature you are expecting. The function `f()` would now look like:

```python
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

`int32(int32, int32)` is the function's signature. In this case, the corresponding specialization will be compiled by the `@jit` decorator, and no other specialization will be allowed. This is useful if you want fine-grained control over types chosen by the compiler (for example, to use single-precision floats).

If you omit the return type, e.g. by writing `(int32, int32)` instead of `int32(int32, int32)`, Numba will try to infer it for you. Function signatures can also be strings, and you can pass several of them as a list; see the `numba.jit()` documentation for more details.

Of course, the compiled function gives the expected results:

```
>>> f(1,2)
3
```

and if we specified `int32` as return type, the higher-order bits get discarded:

```
>>> f(2**31, 2**31 + 1)
1
```

1.4.2 Calling and inlining other functions

Numba-compiled functions can call other compiled functions. The function calls may even be inlined in the native code, depending on optimizer heuristics. For example:

```python
@jit
def square(x):
    return x ** 2

@jit
def hypot(x, y):
    return math.sqrt(square(x) + square(y))
```

The `@jit` decorator *must* be added to any such library function, otherwise Numba may generate much slower code.

1.4.3 Signature specifications

Explicit `@jit` signatures can use a number of types. Here are some common ones:

- `void` is the return type of functions returning nothing (which actually return `None` when called from Python)

- `intp` and `uintp` are pointer-sized integers (signed and unsigned, respectively)

- `intc` and `uintc` are equivalent to C `int` and `unsigned int` integer types

- `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64` are fixed-width integers of the corresponding bit width (signed and unsigned)

- `float32` and `float64` are single- and double-precision floating-point numbers, respectively

- `complex64` and `complex128` are single- and double-precision complex numbers, respectively

- array types can be specified by indexing any numeric type, e.g. `float32[:]` for a one-dimensional single-precision array or `int8[:,:]` for a two-dimensional array of 8-bit integers.

1.4.4 Compilation options

A number of keyword-only arguments can be passed to the `@jit` decorator.

nopython

Numba has two compilation modes: *nopython mode* and *object mode*. The former produces much faster code, but has limitations that can force Numba to fall back to the latter. To prevent Numba from falling back, and instead raise an error, pass `nopython=True`.

```python
@jit(nopython=True)
def f(x, y):
    return x + y
```

See also:

Troubleshooting and tips

nogil

Whenever Numba optimizes Python code to native code that only works on native types and variables (rather than Python objects), it is not necessary anymore to hold Python's global interpreter lock (GIL). Numba will release the GIL when entering such a compiled function if you passed `nogil=True`.

```python
@jit(nogil=True)
def f(x, y):
    return x + y
```

Code running with the GIL released runs concurrently with other threads executing Python or Numba code (either the same compiled function, or another one), allowing you to take advantage of multi-core systems. This will not be possible if the function is compiled in *object mode*.

When using `nogil=True`, you'll have to be wary of the usual pitfalls of multi-threaded programming (consistency, synchronization, race conditions, etc.).

cache

To avoid compilation times each time you invoke a Python program, you can instruct Numba to write the result of function compilation into a file-based cache. This is done by passing `cache=True`:

```
@jit(cache=True)
def f(x, y):
    return x + y
```

parallel

Enables automatic parallelization (and related optimizations) for those operations in the function known to have parallel semantics. For a list of supported operations, see *Automatic parallelization with @jit*. This feature is enabled by passing `parallel=True` and must be used in conjunction with `nopython=True`:

```
@jit(nopython=True, parallel=True)
def f(x, y):
    return x + y
```

See also:

Automatic parallelization with @jit

1.5 Flexible specializations with `@generated_jit`

While the `jit()` decorator is useful for many situations, sometimes you want to write a function that has different implementations depending on its input types. The `generated_jit()` decorator allows the user to control the selection of a specialization at compile-time, while fully retaining runtime execution speed of a JIT function.

1.5.1 Example

Suppose you want to write a function which returns whether a given value is a "missing" value according to certain conventions. For the sake of the example, let's adopt the following definition:

- for floating-point arguments, a missing value is a `NaN`

- for Numpy datetime64 and timedelta64 arguments, a missing value is a `NaT`

- other types don't have the concept of a missing value.

That compile-time logic is easily implemented using the `generated_jit()` decorator:

```
import numpy as np

from numba import generated_jit, types

@generated_jit(nopython=True)
def is_missing(x):
    """
    Return True if the value is missing, False otherwise.
    """
    if isinstance(x, types.Float):
        return lambda x: np.isnan(x)
    elif isinstance(x, (types.NPDatetime, types.NPTimedelta)):
        # The corresponding Not-a-Time value
        missing = x('NaT')
        return lambda x: x == missing
    else:
        return lambda x: False
```

There are several things to note here:

- The decorated function is called with the *Numba types* of the arguments, not their values.

- The decorated function doesn't actually compute a result, it returns a callable implementing the actual definition of the function for the given types.

- It is possible to pre-compute some data at compile-time (the `missing` variable above) to have them reused inside the compiled implementation.

- The function definitions use the same names for arguments as in the decorated function, this is required to ensure passing arguments by name works as expected.

1.5.2 Compilation options

The `generated_jit()` decorator supports the same keyword-only arguments as the `jit()` decorator, for example the `nopython` and `cache` options.

1.6 Creating NumPy universal functions

There are two types of universal functions:

- Those which operate on scalars, these are "universal functions" or *ufuncs* (see `@vectorize` below).

- Those which operate on higher dimensional arrays and scalars, these are "generalized universal functions" or *gufuncs* (`@guvectorize` below).

1.6.1 The `@vectorize` decorator

Numba's vectorize allows Python functions taking scalar input arguments to be used as NumPy ufuncs. Creating a traditional NumPy ufunc is not the most straightforward process and involves writing some C code. Numba makes this easy. Using the `vectorize()` decorator, Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C.

Using `vectorize()`, you write your function as operating over input scalars, rather than arrays. Numba will generate the surrounding loop (or *kernel*) allowing efficient iteration over the actual inputs.

The `vectorize()` decorator has two modes of operation:

- Eager, or decoration-time, compilation: If you pass one or more type signatures to the decorator, you will be building a Numpy universal function (ufunc). The rest of this subsection describes building ufuncs using decoration-time compilation.

- Lazy, or call-time, compilation: When not given any signatures, the decorator will give you a Numba dynamic universal function (*DUFunc*) that dynamically compiles a new kernel when called with a previously unsupported input type. A later subsection, "*Dynamic universal functions*", describes this mode in more depth.

As described above, if you pass a list of signatures to the `vectorize()` decorator, your function will be compiled into a Numpy ufunc. In the basic case, only one signature will be passed:

```python
from numba import vectorize, float64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```

If you pass several signatures, beware that you have to pass most specific signatures before least specific ones (e.g., single-precision floats before double-precision floats), otherwise type-based dispatching will not work as expected:

```
@vectorize([int32(int32, int32),
            int64(int64, int64),
            float32(float32, float32),
            float64(float64, float64)])
def f(x, y):
    return x + y
```

The function will work as expected over the specified array types:

```
>>> a = np.arange(6)
>>> f(a, a)
array([ 0,  2,  4,  6,  8, 10])
>>> a = np.linspace(0, 1, 6)
>>> f(a, a)
array([ 0. ,  0.4,  0.8,  1.2,  1.6,  2. ])
```

but it will fail working on other types:

```
>>> a = np.linspace(0, 1+1j, 6)
>>> f(a, a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ufunc 'ufunc' not supported for the input types, and the inputs could not
→be safely coerced to any supported types according to the casting rule ''safe''
```

You might ask yourself, "why would I go through this instead of compiling a simple iteration loop using the *@jit* decorator?". The answer is that NumPy ufuncs automatically get other features such as reduction, accumulation or broadcasting. Using the example above:

```
>>> a = np.arange(12).reshape(3, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> f.reduce(a, axis=0)
array([12, 15, 18, 21])
>>> f.reduce(a, axis=1)
array([ 6, 22, 38])
>>> f.accumulate(a)
array([[ 0,  1,  2,  3],
       [ 4,  6,  8, 10],
       [12, 15, 18, 21]])
>>> f.accumulate(a, axis=1)
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

See also:

Standard features of ufuncs (NumPy documentation).

Note: Only the broadcasting features of ufuncs are supported in compiled code.

The *vectorize()* decorator supports multiple ufunc targets:

Target	Description
cpu	Single-threaded CPU
parallel	Multi-core CPU
cuda	CUDA GPU **Note:** This creates an *ufunc-like* object. See documentation for CUDA ufunc for detail.

A general guideline is to choose different targets for different data sizes and algorithms. The "cpu" target works well for small data sizes (approx. less than 1KB) and low compute intensity algorithms. It has the least amount of overhead. The "parallel" target works well for medium data sizes (approx. less than 1MB). Threading adds a small delay. The "cuda" target works well for big data sizes (approx. greater than 1MB) and high compute intensity algorithms. Transferring memory to and from the GPU adds significant overhead.

1.6.2 The `@guvectorize` decorator

While *vectorize()* allows you to write ufuncs that work on one element at a time, the *guvectorize()* decorator takes the concept one step further and allows you to write ufuncs that will work on an arbitrary number of elements of input arrays, and take and return arrays of differing dimensions. The typical example is a running median or a convolution filter.

Contrary to *vectorize()* functions, *guvectorize()* functions don't return their result value: they take it as an array argument, which must be filled in by the function. This is because the array is actually allocated by NumPy's dispatch mechanism, which calls into the Numba-generated code.

Here is a very simple example:

```
@guvectorize([(int64[:], int64, int64[:])], '(n),()->(n)')
def g(x, y, res):
    for i in range(x.shape[0]):
        res[i] = x[i] + y
```

The underlying Python function simply adds a given scalar (`y`) to all elements of a 1-dimension array. What's more interesting is the declaration. There are two things there:

- the declaration of input and output *layouts*, in symbolic form: `(n),()->(n)` tells NumPy that the function takes a *n*-element one-dimension array, a scalar (symbolically denoted by the empty tuple `()`) and returns a *n*-element one-dimension array;

- the list of supported concrete *signatures* as per `@vectorize`; here, as in the above example, we demonstrate `int64` arrays.

Note: 1D array type can also receive scalar arguments (those with shape `()`). In the above example, the second argument also could be declared as `int64[:]`. In that case, the value must be read by `y[0]`.

We can now check what the compiled ufunc does, over a simple example:

```
>>> a = np.arange(5)
>>> a
```

(continues on next page)

```
array([0, 1, 2, 3, 4])
>>> g(a, 2)
array([2, 3, 4, 5, 6])
```

The nice thing is that NumPy will automatically dispatch over more complicated inputs, depending on their shapes:

```
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> g(a, 10)
array([[10, 11, 12],
       [13, 14, 15]])
>>> g(a, np.array([10, 20]))
array([[10, 11, 12],
       [23, 24, 25]])
```

Note: Both *vectorize()* and *guvectorize()* support passing `nopython=True` *as in the @jit decorator*. Use it to ensure the generated code does not fallback to *object mode*.

Overwriting input values

In most cases, writing to inputs may also appear to work - however, this behaviour cannot be relied on. Consider the following example function:

```
@guvectorize([(float64[:], float64[:])], '()->()')
def init_values(invals, outvals):
    invals[0] = 6.5
    outvals[0] = 4.2
```

Calling the *init_values* function with an array of *float64* type results in visible changes to the input:

```
>>> invals = np.zeros(shape=(3, 3), dtype=np.float64)
>>> outvals = init_values(invals)
>>> invals
array([[6.5, 6.5, 6.5],
       [6.5, 6.5, 6.5],
       [6.5, 6.5, 6.5]])
>>> outvals
array([[4.2, 4.2, 4.2],
    [4.2, 4.2, 4.2],
    [4.2, 4.2, 4.2]])
```

This works because NumPy can pass the input data directly into the *init_values* function as the data *dtype* matches that of the declared argument. However, it may also create and pass in a temporary array, in which case changes to the input are lost. For example, this can occur when casting is required. To demonstrate, we can use an array of *float32* with the *init_values* function:

```
>>> invals = np.zeros(shape=(3, 3), dtype=np.float32)
>>> outvals = init_values(invals)
>>> invals
array([[0., 0., 0.],
```

```
        [0., 0., 0.],
        [0., 0., 0.]], dtype=float32)
```

In this case, there is no change to the *invals* array because the temporary casted array was mutated instead.

1.6.3 Dynamic universal functions

As described above, if you do not pass any signatures to the `vectorize()` decorator, your Python function will be used to build a dynamic universal function, or *DUFunc*. For example:

```
from numba import vectorize

@vectorize
def f(x, y):
    return x * y
```

The resulting `f()` is a *DUFunc* instance that starts with no supported input types. As you make calls to `f()`, Numba generates new kernels whenever you pass a previously unsupported input type. Given the example above, the following set of interpreter interactions illustrate how dynamic compilation works:

```
>>> f
<numba._DUFunc 'f'>
>>> f.ufunc
<ufunc 'f'>
>>> f.ufunc.types
[]
```

The example above shows that *DUFunc* instances are not ufuncs. Rather than subclass ufunc's, *DUFunc* instances work by keeping a *ufunc* member, and then delegating ufunc property reads and method calls to this member (also known as type aggregation). When we look at the initial types supported by the ufunc, we can verify there are none.

Let's try to make a call to `f()`:

```
>>> f(3,4)
12
>>> f.types    # shorthand for f.ufunc.types
['ll->l']
```

If this was a normal Numpy ufunc, we would have seen an exception complaining that the ufunc couldn't handle the input types. When we call `f()` with integer arguments, not only do we receive an answer, but we can verify that Numba created a loop supporting C `long` integers.

We can add additional loops by calling `f()` with different inputs:

```
>>> f(1.,2.)
2.0
>>> f.types
['ll->l', 'dd->d']
```

We can now verify that Numba added a second loop for dealing with floating-point inputs, `"dd->d"`.

If we mix input types to `f()`, we can verify that Numpy ufunc casting rules are still in effect:

```
>>> f(1,2.)
2.0
```

```
>>> f.types
['ll->l', 'dd->d']
```

This example demonstrates that calling `f()` with mixed types caused Numpy to select the floating-point loop, and cast the integer argument to a floating-point value. Thus, Numba did not create a special `"dl->d"` kernel.

This *DUFunc* behavior leads us to a point similar to the warning given above in "*The @vectorize decorator*" subsection, but instead of signature declaration order in the decorator, call order matters. If we had passed in floating-point arguments first, any calls with integer arguments would be cast to double-precision floating-point values. For example:

```
>>> @vectorize
... def g(a, b): return a / b
...
>>> g(2.,3.)
0.66666666666666663
>>> g(2,3)
0.66666666666666663
>>> g.types
['dd->d']
```

If you require precise support for various type signatures, you should specify them in the `vectorize()` decorator, and not rely on dynamic compilation.

1.7 Compiling Python classes with `@jitclass`

Note: This is a early version of jitclass support. Not all compiling features are exposed or implemented, yet.

Numba supports code generation for classes via the `numba.jitclass()` decorator. A class can be marked for optimization using this decorator along with a specification of the types of each field. We call the resulting class object a *jitclass*. All methods of a jitclass are compiled into nopython functions. The data of a jitclass instance is allocated on the heap as a C-compatible structure so that any compiled functions can have direct access to the underlying data, bypassing the interpreter.

1.7.1 Basic usage

Here's an example of a jitclass:

```
import numpy as np
from numba import int32, float32    # import the types
from numba.experimental import jitclass

spec = [
    ('value', int32),               # a simple scalar field
    ('array', float32[:]),          # an array field
]

@jitclass(spec)
class Bag(object):
    def __init__(self, value):
        self.value = value
        self.array = np.zeros(value, dtype=np.float32)
```

```
    @property
    def size(self):
        return self.array.size

    def increment(self, val):
        for i in range(self.size):
            self.array[i] += val
        return self.array

    @staticmethod
    def add(x, y):
        return x + y

n = 21
mybag = Bag(n)
```

In the above example, a `spec` is provided as a list of 2-tuples. The tuples contain the name of the field and the Numba type of the field. Alternatively, user can use a dictionary (an `OrderedDict` preferably for stable field ordering), which maps field names to types.

The definition of the class requires at least a `__init__` method for initializing each defined fields. Uninitialized fields contains garbage data. Methods and properties (getters and setters only) can be defined. They will be automatically compiled.

1.7.2 Specifying `numba.typed` containers as class members

It is often desirable to use a `numba.typed.Dict` or a `numba.typed.List` as a class member in a `jitclass`. Methods for using these types and various common patterns are presented in the following:

First, using explicit Numba types and explicit construction.

```
from numba import jitclass, types, typed

# key and value types
kv_ty = (types.int64, types.unicode_type)

# A container class with:
# * member 'd' holding a typed dictionary of int64 -> unicode string (kv_ty)
# * member 'l' holding a typed list of float64
@jitclass([('d', types.DictType(*kv_ty)),
           ('l', types.ListType(types.float64))])
class ContainerHolder(object):
    def __init__(self):
        # initialize the containers
        self.d = typed.Dict.empty(*kv_ty)
        self.l = typed.List.empty_list(types.float64)

container = ContainerHolder()
container.d[1] = "apple"
container.d[2] = "orange"
container.l.append(123.)
container.l.append(456.)
print(container.d) # {1: apple, 2: orange}
print(container.l) # [123.0, 456.0]
```

Another useful pattern is to use the `numba.typed` container attribute `_numba_type_` to find the type of a container, this can be accessed directly from an instance of the container in the Python interpreter. The same information can be obtained by calling *numba.typeof()* on the instance. For example:

```python
from numba import jitclass, typed, typeof

d = typed.Dict()
d[1] = "apple"
d[2] = "orange"
l = typed.List()
l.append(123.)
l.append(456.)


@jitclass([('d', typeof(d)), ('l', typeof(l))])
class ContainerInstHolder(object):
    def __init__(self, dict_inst, list_inst):
        self.d = dict_inst
        self.l = list_inst

container = ContainerInstHolder(d, l)
print(container.d) # {1: apple, 2: orange}
print(container.l) # [123.0, 456.0]
```

It is worth noting that the instance of the container in a `jitclass` must be initialized before use, for example, this will cause an invalid memory access as `self.d` is written to without `d` being initialized as a `type.Dict` instance of the type specified.

```python
from numba import jitclass, types

dict_ty = types.DictType(types.int64, types.unicode_type)

@jitclass([('d', dict_ty)])
class NotInitilisingContainer(object):
    def __init__(self):
        self.d[10] = "apple" # this is invalid, `d` is not initialized

NotInitilisingContainer() # segmentation fault/memory access violation
```

1.7.3 Support operations

The following operations of jitclasses work in both the interpreter and Numba compiled functions:

- calling the jitclass class object to construct a new instance (e.g. `mybag = Bag(123)`);
- read/write access to attributes and properties (e.g. `mybag.value`);
- calling methods (e.g. `mybag.increment(3)`);
- calling static methods as instance attributes (e.g. `mybag.add(1, 1)`);
- calling static methods as class attributes (e.g. `Bag.add(1, 2)`);

Using jitclasses in Numba compiled function is more efficient. Short methods can be inlined (at the discretion of LLVM inliner). Attributes access are simply reading from a C structure. Using jitclasses from the interpreter has the same overhead of calling any Numba compiled function from the interpreter. Arguments and return values must be unboxed or boxed between Python objects and native representation. Values encapsulated by a jitclass does not get boxed into Python object when the jitclass instance is handed to the interpreter. It is during attribute access to the field

values that they are boxed. Calling static methods as class attributes is only supported outside of the class definition (i.e. you can't call `Bag.add()` from within another method of `Bag`).

1.7.4 Limitations

- A jitclass class object is treated as a function (the constructor) inside a Numba compiled function.
- `isinstance()` only works in the interpreter.
- Manipulating jitclass instances in the interpreter is not optimized, yet.
- Support for jitclasses are available on CPU only. (Note: Support for GPU devices is planned for a future release.)

1.7.5 The decorator: `@jitclass`

numba.experimental.**jitclass**(*spec*)
> A decorator for creating a jitclass.

> **arguments**:

> - **spec:** Specifies the types of each field on this class. Must be a dictionary or a sequence. With a dictionary, use collections.OrderedDict for stable ordering. With a sequence, it must contain 2-tuples of (fieldname, fieldtype).

> **returns**:

> A callable that takes a class object, which will be compiled.

1.8 Creating C callbacks with `@cfunc`

Interfacing with some native libraries (for example written in C or C++) can necessitate writing native callbacks to provide business logic to the library. The *numba.cfunc()* decorator creates a compiled function callable from foreign C code, using the signature of your choice.

1.8.1 Basic usage

The `@cfunc` decorator has a similar usage to `@jit`, but with an important difference: passing a single signature is mandatory. It determines the visible signature of the C callback:

```python
from numba import cfunc

@cfunc("float64(float64, float64)")
def add(x, y):
    return x + y
```

The C function object exposes the address of the compiled C callback as the *address* attribute, so that you can pass it to any foreign C or C++ library. It also exposes a `ctypes` callback object pointing to that callback; that object is also callable from Python, making it easy to check the compiled code:

```python
@cfunc("float64(float64, float64)")
def add(x, y):
    return x + y

print(add.ctypes(4.0, 5.0))  # prints "9.0"
```

1.8.2 Example

In this example, we are going to be using the `scipy.integrate.quad` function. That function accepts either a regular Python callback or a C callback wrapped in a `ctypes` callback object.

Let's define a pure Python integrand and compile it as a C callback:

```
>>> import numpy as np
>>> from numba import cfunc
>>> def integrand(t):
        return np.exp(-t) / t**2
   ...:
>>> nb_integrand = cfunc("float64(float64)")(integrand)
```

We can pass the `nb_integrand` object's `ctypes` callback to `scipy.integrate.quad` and check that the results are the same as with the pure Python function:

```
>>> import scipy.integrate as si
>>> def do_integrate(func):
        """
        Integrate the given function from 1.0 to +inf.
        """
        return si.quad(func, 1, np.inf)
   ...:
>>> do_integrate(integrand)
(0.14849550677592208, 3.8736750296130505e-10)
>>> do_integrate(nb_integrand.ctypes)
(0.14849550677592208, 3.8736750296130505e-10)
```

Using the compiled callback, the integration function does not invoke the Python interpreter each time it evaluates the integrand. In our case, the integration is made 18 times faster:

```
>>> %timeit do_integrate(integrand)
1000 loops, best of 3: 242 µs per loop
>>> %timeit do_integrate(nb_integrand.ctypes)
100000 loops, best of 3: 13.5 µs per loop
```

1.8.3 Dealing with pointers and array memory

A less trivial use case of C callbacks involves doing operation on some array of data passed by the caller. As C doesn't have a high-level abstraction similar to Numpy arrays, the C callback's signature will pass low-level pointer and size arguments. Nevertheless, the Python code for the callback will expect to exploit the power and expressiveness of Numpy arrays.

In the following example, the C callback is expected to operate on 2-d arrays, with the signature `void(double *input, double *output, int m, int n)`. You can implement such a callback thusly:

```
from numba import cfunc, types, carray

c_sig = types.void(types.CPointer(types.double),
                   types.CPointer(types.double),
                   types.intc, types.intc)

@cfunc(c_sig)
def my_callback(in_, out, m, n):
    in_array = carray(in_, (m, n))
```

(continues on next page)

```
    out_array = carray(out, (m, n))
    for i in range(m):
        for j in range(n):
            out_array[i, j] = 2 * in_array[i, j]
```

The *numba.carray()* function takes as input a data pointer and a shape and returns an array view of the given shape over that data. The data is assumed to be laid out in C order. If the data is laid out in Fortran order, *numba.farray()* should be used instead.

1.8.4 Handling C structures

With CFFI

For applications that have a lot of state, it is useful to pass data in C structures. To simplify the interoperability with C code, numba can convert a `cffi` type into a numba `Record` type using `numba.core.typing.cffi_utils.map_type`:

```python
from numba.core.typing import cffi_utils

nbtype = cffi_utils.map_type(cffi_type, use_record_dtype=True)
```

Note: **use_record_dtype=True** is needed otherwise pointers to C structures are returned as void pointers.

Note: From v0.49 the `numba.cffi_support` module has been phased out in favour of `numba.core.typing.cffi_utils`

For example:

```python
from cffi import FFI

src = """

/* Define the C struct */
typedef struct my_struct {
    int    i1;
    float  f2;
    double d3;
    float  af4[7]; // arrays are supported
} my_struct;

/* Define a callback function */
typedef double (*my_func)(my_struct*, size_t);
"""

ffi = FFI()
ffi.cdef(src)

# Get the function signature from *my_func*
sig = cffi_utils.map_type(ffi.typeof('my_func'), use_record_dtype=True)

# Make the cfunc
```

```python
from numba import cfunc, carray

@cfunc(sig)
def foo(ptr, n):
    base = carray(ptr, n)   # view pointer as an array of my_struct
    tmp = 0
    for i in range(n):
        tmp += base[i].i1 * base[i].f2 / base[i].d3
        tmp += base[i].af4.sum()   # nested arrays are like normal numpy array
    return tmp
```

With `numba.types.Record.make_c_struct`

The `numba.types.Record` type can be created manually to follow a C-structure's layout. To do that, use `Record.make_c_struct`, for example:

```python
my_struct = types.Record.make_c_struct([
    # Provides a sequence of 2-tuples i.e. (name:str, type:Type)
    ('i1', types.int32),
    ('f2', types.float32),
    ('d3', types.float64),
    ('af4', types.NestedArray(dtype=types.float32, shape=(7,))),
])
```

Due to ABI limitations, structures should be passed as pointers using `types.CPointer(my_struct)` as the argument type. Inside the `cfunc` body, the `my_struct*` can be accessed with `carray`.

Full example

See full example in `examples/notebooks/Accessing C Struct Data.ipynb`.

1.8.5 Signature specification

The explicit `@cfunc` signature can use any *Numba types*, but only a subset of them make sense for a C callback. You should generally limit yourself to scalar types (such as `int8` or `float64`) ,pointers to them (for example `types.CPointer(types.int8)`), or pointers to `Record` type.

1.8.6 Compilation options

A number of keyword-only arguments can be passed to the `@cfunc` decorator: `nopython` and `cache`. Their meaning is similar to those in the `@jit` decorator.

1.9 Compiling code ahead of time

While Numba's main use case is *Just-in-Time compilation*, it also provides a facility for *Ahead-of-Time compilation* (AOT).

1.9.1 Overview

Benefits

1. AOT compilation produces a compiled extension module which does not depend on Numba: you can distribute the module on machines which do not have Numba installed (but Numpy is required).

2. There is no compilation overhead at runtime (but see the @jit *cache* option), nor any overhead of importing Numba.

See also:

Compiled extension modules are discussed in the Python packaging user guide.

Limitations

1. AOT compilation only allows for regular functions, not *ufuncs*.

2. You have to specify function signatures explicitly.

3. Each exported function can have only one signature (but you can export several different signatures under different names).

4. AOT compilation produces generic code for your CPU's architectural family (for example "x86-64"), while JIT compilation produces code optimized for your particular CPU model.

1.9.2 Usage

Standalone example

```python
from numba.pycc import CC

cc = CC('my_module')
# Uncomment the following line to print out the compilation steps
#cc.verbose = True

@cc.export('multf', 'f8(f8, f8)')
@cc.export('multi', 'i4(i4, i4)')
def mult(a, b):
    return a * b

@cc.export('square', 'f8(f8)')
def square(a):
    return a ** 2

if __name__ == "__main__":
    cc.compile()
```

If you run this Python script, it will generate an extension module named `my_module`. Depending on your platform, the actual filename may be `my_module.so`, `my_module.pyd`, `my_module.cpython-34m.so`, etc.

The generated module has three functions: `multf`, `multi` and `square`. `multi` operates on 32-bit integers (`i4`), while `multf` and `square` operate on double-precision floats (`f8`):

```
>>> import my_module
>>> my_module.multi(3, 4)
12
>>> my_module.square(1.414)
1.9993959999999997
```

Distutils integration

You can also integrate the compilation step for your extension modules in your `setup.py` script, using distutils or setuptools:

```python
from distutils.core import setup

from source_module import cc

setup(...,
      ext_modules=[cc.distutils_extension()])
```

The `source_module` above is the module defining the `cc` object. Extensions compiled like this will be automatically included in the build files for your Python project, so you can distribute them inside binary packages such as wheels or Conda packages. Note that in the case of using conda, the compilers used for AOT need to be those that are available in the Anaconda distribution.

Signature syntax

The syntax for exported signatures is the same as in the `@jit` decorator. You can read more about it in the *types* reference.

Here is an example of exporting an implementation of the second-order centered difference on a 1d array:

```python
@cc.export('centdiff_1d', 'f8[:](f8[:], f8)')
def centdiff_1d(u, dx):
    D = np.empty_like(u)
    D[0] = 0
    D[-1] = 0
    for i in range(1, len(D) - 1):
        D[i] = (u[i+1] - 2 * u[i] + u[i-1]) / dx**2
    return D
```

You can also omit the return type, which will then be inferred by Numba:

```python
@cc.export('centdiff_1d', '(f8[:], f8)')
def centdiff_1d(u, dx):
    # Same code as above
    ...
```

1.10 Automatic parallelization with `@jit`

Setting the *parallel* option for `jit()` enables a Numba transformation pass that attempts to automatically parallelize and perform other optimizations on (part of) a function. At the moment, this feature only works on CPUs.

Some operations inside a user defined function, e.g. adding a scalar value to an array, are known to have parallel semantics. A user program may contain many such operations and while each operation could be parallelized individually, such an approach often has lackluster performance due to poor cache behavior. Instead, with auto-parallelization, Numba attempts to identify such operations in a user program, and fuse adjacent ones together, to form one or more kernels that are automatically run in parallel. The process is fully automated without modifications to the user program, which is in contrast to Numba's *vectorize()* or *guvectorize()* mechanism, where manual effort is required to create parallel kernels.

1.10.1 Supported Operations

In this section, we give a list of all the array operations that have parallel semantics and for which we attempt to parallelize.

1. All numba array operations that are supported by *Case study: Array Expressions*, which include common arithmetic functions between Numpy arrays, and between arrays and scalars, as well as Numpy ufuncs. They are often called *element-wise* or *point-wise* array operations:

   • unary operators: + - ~

   • binary operators: + - * / /? % | >> ^ << & ** //

   • comparison operators: == != < <= > >=

   • *Numpy ufuncs* that are supported in *nopython mode*.

   • User defined *DUFunc* through *vectorize()*.

2. Numpy reduction functions `sum`, `prod`, `min`, `max`, `argmin`, and `argmax`. Also, array math functions `mean`, `var`, and `std`.

3. Numpy array creation functions `zeros`, `ones`, `arange`, `linspace`, and several random functions (rand, randn, ranf, random_sample, sample, random, standard_normal, chisquare, weibull, power, geometric, exponential, poisson, rayleigh, normal, uniform, beta, binomial, f, gamma, lognormal, laplace, randint, triangular).

4. Numpy `dot` function between a matrix and a vector, or two vectors. In all other cases, Numba's default implementation is used.

5. Multi-dimensional arrays are also supported for the above operations when operands have matching dimension and size. The full semantics of Numpy broadcast between arrays with mixed dimensionality or size is not supported, nor is the reduction across a selected dimension.

6. Array assignment in which the target is an array selection using a slice or a boolean array, and the value being assigned is either a scalar or another selection where the slice range or bitarray are inferred to be compatible.

7. The `reduce` operator of `functools` is supported for specifying parallel reductions on 1D Numpy arrays but the initial value argument is mandatory.

1.10.2 Explicit Parallel Loops

Another feature of the code transformation pass (when `parallel=True`) is support for explicit parallel loops. One can use Numba's `prange` instead of `range` to specify that a loop can be parallelized. The user is required to make sure that the loop does not have cross iteration dependencies except for supported reductions.

A reduction is inferred automatically if a variable is updated by a binary function/operator using its previous value in the loop body. The initial value of the reduction is inferred automatically for the +=, -=, *=, and /= operators. For other functions/operators, the reduction variable should hold the identity value right before entering the `prange` loop. Reductions in this manner are supported for scalars and for arrays of arbitrary dimensions.

The example below demonstrates a parallel loop with a reduction (A is a one-dimensional Numpy array):

```python
from numba import njit, prange

@njit(parallel=True)
def prange_test(A):
    s = 0
    # Without "parallel=True" in the jit-decorator
    # the prange statement is equivalent to range
    for i in prange(A.shape[0]):
        s += A[i]
    return s
```

The following example demonstrates a product reduction on a two-dimensional array:

```python
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def two_d_array_reduction_prod(n):
    shp = (13, 17)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1
```

Care should be taken, however, when reducing into slices or elements of an array if the elements specified by the slice or index are written to simultaneously by multiple parallel threads. The compiler may not detect such cases and then a race condition would occur.

The following example demonstrates such a case where a race condition in the execution of the parallel for-loop results in an incorrect return value:

```python
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[:] += x[i]

    return y
```

as does the following example where the accumulating element is explicitly specified:

```python
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_wrong_result(x):
    n = x.shape[0]
    y = np.zeros(4)
```

(continues on next page)

```
    for i in prange(n):
        # accumulating into the same element of `y` from different
        # parallel iterations of the loop results in a race condition
        y[i % 4] += x[i]

    return y
```

whereas performing a whole array reduction is fine:

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_ok_result_whole_arr(x):
    n = x.shape[0]
    y = np.zeros(4)
    for i in prange(n):
        y += x[i]
    return y
```

as is creating a slice reference outside of the parallel reduction loop:

```
from numba import njit, prange
import numpy as np

@njit(parallel=True)
def prange_ok_result_outer_slice(x):
    n = x.shape[0]
    y = np.zeros(4)
    z = y[:]
    for i in prange(n):
        z += x[i]
    return y
```

1.10.3 Examples

In this section, we give an example of how this feature helps parallelize Logistic Regression:

```
@numba.jit(nopython=True, parallel=True)
def logistic_regression(Y, X, w, iterations):
    for i in range(iterations):
        w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w)))) - 1.0) * Y), X)
    return w
```

We will not discuss details of the algorithm, but instead focus on how this program behaves with auto-parallelization:

1. Input `Y` is a vector of size `N`, `X` is an `N x D` matrix, and `w` is a vector of size `D`.

2. The function body is an iterative loop that updates variable `w`. The loop body consists of a sequence of vector and matrix operations.

3. The inner `dot` operation produces a vector of size `N`, followed by a sequence of arithmetic operations either between a scalar and vector of size `N`, or two vectors both of size `N`.

4. The outer `dot` produces a vector of size `D`, followed by an inplace array subtraction on variable `w`.

5. With auto-parallelization, all operations that produce array of size `N` are fused together to become a single parallel kernel. This includes the inner `dot` operation and all point-wise array operations following it.

6. The outer `dot` operation produces a result array of different dimension, and is not fused with the above kernel.

Here, the only thing required to take advantage of parallel hardware is to set the *parallel* option for `jit()`, with no modifications to the `logistic_regression` function itself. If we were to give an equivalence parallel implementation using `guvectorize()`, it would require a pervasive change that rewrites the code to extract kernel computation that can be parallelized, which was both tedious and challenging.

1.10.4 Diagnostics

Note: At present not all parallel transforms and functions can be tracked through the code generation process. Occasionally diagnostics about some loops or transforms may be missing.

The *parallel* option for `jit()` can produce diagnostic information about the transforms undertaken in automatically parallelizing the decorated code. This information can be accessed in two ways, the first is by setting the environment variable *NUMBA_PARALLEL_DIAGNOSTICS*, the second is by calling *parallel_diagnostics()*, both methods give the same information and print to `STDOUT`. The level of verbosity in the diagnostic information is controlled by an integer argument of value between 1 and 4 inclusive, 1 being the least verbose and 4 the most. For example:

```python
@njit(parallel=True)
def test(x):
    n = x.shape[0]
    a = np.sin(x)
    b = np.cos(a * a)
    acc = 0
    for i in prange(n - 2):
        for j in prange(n - 1):
            acc += b[i] + b[j + 1]
    return acc

test(np.arange(10))

test.parallel_diagnostics(level=4)
```

produces:

```
================================================================================
======= Parallel Accelerator Optimizing:  Function test, example.py (4)   =======
================================================================================


Parallel loop listing for  Function test, example.py (4)
--------------------------------------|loop #ID
@njit(parallel=True)                  |
def test(x):                          |
    n = x.shape[0]                    |
    a = np.sin(x)---------------------| #0
    b = np.cos(a * a)-----------------| #1
    acc = 0                           |
    for i in prange(n - 2):-----------| #3
        for j in prange(n - 1):-------| #2
            acc += b[i] + b[j + 1]    |
    return acc                        |
```

```
------------------------------ Fusing loops ---------------------------------
Attempting fusion of parallel loops (combines loops with similar properties)...
Trying to fuse loops #0 and #1:
    - fusion succeeded: parallel for-loop #1 is fused into for-loop #0.
Trying to fuse loops #0 and #3:
    - fusion failed: loop dimension mismatched in axis 0. slice(0, x_size0.1, 1)
!= slice(0, $40.4, 1)
-------------------------- Before Optimization ------------------------------
Parallel region 0:
+--0 (parallel)
+--1 (parallel)


Parallel region 1:
+--3 (parallel)
+--2 (parallel)



--------------------------------------------------------------------------------
--------------------------- After Optimization ------------------------------
Parallel region 0:
+--0 (parallel, fused with loop(s): 1)


Parallel region 1:
+--3 (parallel)
+--2 (serial)



Parallel region 0 (loop #0) had 1 loop(s) fused.

Parallel region 1 (loop #3) had 0 loop(s) fused and 1 loop(s) serialized as part
of the larger parallel loop (#3).
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

-------------------------Loop invariant code motion--------------------------

Instruction hoisting:
loop #0:
Failed to hoist the following:
    dependency: $arg_out_var.10 = getitem(value=x, index=$parfor__index_5.99)
    dependency: $0.6.11 = getattr(value=$0.5, attr=sin)
    dependency: $expr_out_var.9 = call $0.6.11($arg_out_var.10, func=$0.6.11,␣
↪args=[Var($arg_out_var.10, example.py (7))], kws=(), vararg=None)
    dependency: $arg_out_var.17 = $expr_out_var.9 * $expr_out_var.9
    dependency: $0.10.20 = getattr(value=$0.9, attr=cos)
    dependency: $expr_out_var.16 = call $0.10.20($arg_out_var.17, func=$0.10.20,␣
↪args=[Var($arg_out_var.17, example.py (8))], kws=(), vararg=None)
loop #3:
Has the following hoisted:
    $const58.3 = const(int, 1)
    $58.4 = _n_23 - $const58.3
--------------------------------------------------------------------------------
```

To aid users unfamiliar with the transforms undertaken when the *parallel* option is used, and to assist in the under-

standing of the subsequent sections, the following definitions are provided:

- **Loop fusion** Loop fusion is a technique whereby loops with equivalent bounds may be combined under certain conditions to produce a loop with a larger body (aiming to improve data locality).

- **Loop serialization** Loop serialization occurs when any number of `prange` driven loops are present inside another `prange` driven loop. In this case the outermost of all the `prange` loops executes in parallel and any inner `prange` loops (nested or otherwise) are treated as standard `range` based loops. Essentially, nested parallelism does not occur.

- **Loop invariant code motion** Loop invariant code motion is an optimization technique that analyses a loop to look for statements that can be moved outside the loop body without changing the result of executing the loop, these statements are then "hoisted" out of the loop to save repeated computation.

- **Allocation hoisting** Allocation hoisting is a specialized case of loop invariant code motion that is possible due to the design of some common NumPy allocation methods. Explanation of this technique is best driven by an example:

```python
@njit(parallel=True)
def test(n):
    for i in prange(n):
        temp = np.zeros((50, 50)) # <--- Allocate a temporary array with np.
→zeros()
        for j in range(50):
            temp[j, j] = i

    # ...do something with temp
```

internally, this is transformed to approximately the following:

```python
@njit(parallel=True)
def test(n):
    for i in prange(n):
        temp = np.empty((50, 50)) # <--- np.zeros() is rewritten as np.empty()
        temp[:] = 0               # <--- and then a zero initialisation
        for j in range(50):
            temp[j, j] = i

    # ...do something with temp
```

then after hoisting:

```python
@njit(parallel=True)
def test(n):
    temp = np.empty((50, 50)) # <--- allocation is hoisted as a loop␣
→invariant as `np.empty` is considered pure
    for i in prange(n):
        temp[:] = 0            # <--- this remains as assignment is a side␣
→effect
        for j in range(50):
            temp[j, j] = i

    # ...do something with temp
```

it can be seen that the `np.zeros` allocation is split into an allocation and an assignment, and then the allocation is hoisted out of the loop in `i`, this producing more efficient code as the allocation only occurs once.

The parallel diagnostics report sections

The report is split into the following sections:

1. **Code annotation** This is the first section and contains the source code of the decorated function with loops that have parallel semantics identified and enumerated. The `loop #ID` column on the right of the source code lines up with identified parallel loops. From the example, `#0` is `np.sin`, `#1` is `np.cos` and `#2` and `#3` are `prange()`:

```
Parallel loop listing for  Function test, example.py (4)
--------------------------------------|loop #ID
@njit(parallel=True)                  |
def test(x):                          |
    n = x.shape[0]                    |
    a = np.sin(x)---------------------| #0
    b = np.cos(a * a)-----------------| #1
    acc = 0                           |
    for i in prange(n - 2):-----------| #3
        for j in prange(n - 1):-------| #2
            acc += b[i] + b[j + 1]    |
    return acc                        |
```

It is worth noting that the loop IDs are enumerated in the order they are discovered which is not necessarily the same order as present in the source. Further, it should also be noted that the parallel transforms use a static counter for loop ID indexing. As a consequence it is possible for the loop ID index to not start at 0 due to use of the same counter for internal optimizations/transforms taking place that are invisible to the user.

2. **Fusing loops** This section describes the attempts made at fusing discovered loops noting which succeeded and which failed. In the case of failure to fuse a reason is given (e.g. dependency on other data). From the example:

```
-------------------------------- Fusing loops --------------------------------
↪--
Attempting fusion of parallel loops (combines loops with similar properties)..
↪.
Trying to fuse loops #0 and #1:
    - fusion succeeded: parallel for-loop #1 is fused into for-loop #0.
Trying to fuse loops #0 and #3:
    - fusion failed: loop dimension mismatched in axis 0. slice(0, x_size0.1,␣
↪1)
!= slice(0, $40.4, 1)
```

It can be seen that fusion of loops `#0` and `#1` was attempted and this succeeded (both are based on the same dimensions of x). Following the successful fusion of `#0` and `#1`, fusion was attempted between `#0` (now including the fused `#1` loop) and `#3`. This fusion failed because there is a loop dimension mismatch, `#0` is size `x.shape` whereas `#3` is size `x.shape[0] - 2`.

3. **Before Optimization** This section shows the structure of the parallel regions in the code before any optimization has taken place, but with loops associated with their final parallel region (this is to make before/after optimization output directly comparable). Multiple parallel regions may exist if there are loops which cannot be fused, in this case code within each region will execute in parallel, but each parallel region will run sequentially. From the example:

```
Parallel region 0:
+--0 (parallel)
+--1 (parallel)
```

```
Parallel region 1:
+--3 (parallel)
+--2 (parallel)
```

As alluded to by the *Fusing loops* section, there are necessarily two parallel regions in the code. The first contains loops `#0` and `#1`, the second contains `#3` and `#2`, all loops are marked `parallel` as no optimization has taken place yet.

4. **After Optimization** This section shows the structure of the parallel regions in the code after optimization has taken place. Again, parallel regions are enumerated with their corresponding loops but this time loops which are fused or serialized are noted and a summary is presented. From the example:

```
Parallel region 0:
+--0 (parallel, fused with loop(s): 1)


Parallel region 1:
+--3 (parallel)
   +--2 (serial)

Parallel region 0 (loop #0) had 1 loop(s) fused.

Parallel region 1 (loop #3) had 0 loop(s) fused and 1 loop(s) serialized as␣
↪part
of the larger parallel loop (#3).
```

It can be noted that parallel region 0 contains loop `#0` and, as seen in the *fusing loops* section, loop `#1` is fused into loop `#0`. It can also be noted that parallel region 1 contains loop `#3` and that loop `#2` (the inner `prange()`) has been serialized for execution in the body of loop `#3`.

5. **Loop invariant code motion** This section shows for each loop, after optimization has occurred:

   - the instructions that failed to be hoisted and the reason for failure (dependency/impure).

   - the instructions that were hoisted.

   - any allocation hoisting that may have occurred.

   From the example:

```
Instruction hoisting:
loop #0:
Failed to hoist the following:
    dependency: $arg_out_var.10 = getitem(value=x, index=$parfor__index_5.99)
    dependency: $0.6.11 = getattr(value=$0.5, attr=sin)
    dependency: $expr_out_var.9 = call $0.6.11($arg_out_var.10, func=$0.6.11,␣
↪args=[Var($arg_out_var.10, example.py (7))], kws=(), vararg=None)
    dependency: $arg_out_var.17 = $expr_out_var.9 * $expr_out_var.9
    dependency: $0.10.20 = getattr(value=$0.9, attr=cos)
    dependency: $expr_out_var.16 = call $0.10.20($arg_out_var.17, func=$0.10.
↪20, args=[Var($arg_out_var.17, example.py (8))], kws=(), vararg=None)
loop #3:
Has the following hoisted:
    $const58.3 = const(int, 1)
    $58.4 = _n_23 - $const58.3
```

The first thing to note is that this information is for advanced users as it refers to the *Numba IR* of the function being transformed. As an example, the expression `a * a` in the example source partly translates to the expression `$arg_out_var.17 = $expr_out_var.9 * $expr_out_var.9` in the IR, this clearly cannot be hoisted out of `loop #0` because it is not loop invariant! Whereas in `loop #3`, the expression `$const58.3 = const(int, 1)` comes from the source `b[j + 1]`, the number `1` is clearly a constant and so can be hoisted out of the loop.

See also:

parallel, *Parallel FAQs*

1.11 Using the `@stencil` decorator

Stencils are a common computational pattern in which array elements are updated according to some fixed pattern called the stencil kernel. Numba provides the `@stencil` decorator so that users may easily specify a stencil kernel and Numba then generates the looping code necessary to apply that kernel to some input array. Thus, the stencil decorator allows clearer, more concise code and in conjunction with *the parallel jit option* enables higher performance through parallelization of the stencil execution.

1.11.1 Basic usage

An example use of the `@stencil` decorator:

```python
from numba import stencil

@stencil
def kernel1(a):
    return 0.25 * (a[0, 1] + a[1, 0] + a[0, -1] + a[-1, 0])
```

The stencil kernel is specified by what looks like a standard Python function definition but there are different semantics with respect to array indexing. Stencils produce an output array of the same size and shape as the input array although depending on the kernel definition may have a different type. Conceptually, the stencil kernel is run once for each element in the output array. The return value from the stencil kernel is the value written into the output array for that particular element.

The parameter `a` represents the input array over which the kernel is applied. Indexing into this array takes place with respect to the current element of the output array being processed. For example, if element `(x, y)` is being processed then `a[0, 0]` in the stencil kernel corresponds to `a[x + 0, y + 0]` in the input array. Similarly, `a[-1, 1]` in the stencil kernel corresponds to `a[x - 1, y + 1]` in the input array.

Depending on the specified kernel, the kernel may not be applicable to the borders of the output array as this may cause the input array to be accessed out-of-bounds. The way in which the stencil decorator handles this situation is dependent upon which *func_or_mode* is selected. The default mode is for the stencil decorator to set the border elements of the output array to zero.

To invoke a stencil on an input array, call the stencil as if it were a regular function and pass the input array as the argument. For example, using the kernel defined above:

```python
>>> import numpy as np
>>> input_arr = np.arange(100).reshape((10, 10))
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
```

(continues on next page)

```
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
>>> output_arr = kernel1(input_arr)
array([[  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.],
       [  0.,  11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,   0.],
       [  0.,  21.,  22.,  23.,  24.,  25.,  26.,  27.,  28.,   0.],
       [  0.,  31.,  32.,  33.,  34.,  35.,  36.,  37.,  38.,   0.],
       [  0.,  41.,  42.,  43.,  44.,  45.,  46.,  47.,  48.,   0.],
       [  0.,  51.,  52.,  53.,  54.,  55.,  56.,  57.,  58.,   0.],
       [  0.,  61.,  62.,  63.,  64.,  65.,  66.,  67.,  68.,   0.],
       [  0.,  71.,  72.,  73.,  74.,  75.,  76.,  77.,  78.,   0.],
       [  0.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,  88.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.]])
>>> input_arr.dtype
dtype('int64')
>>> output_arr.dtype
dtype('float64')
```

Note that the stencil decorator has determined that the output type of the specified stencil kernel is `float64` and has thus created the output array as `float64` while the input array is of type `int64`.

1.11.2 Stencil Parameters

Stencil kernel definitions may take any number of arguments with the following provisions. The first argument must be an array. The size and shape of the output array will be the same as that of the first argument. Additional arguments may either be scalars or arrays. For array arguments, those arrays must be at least as large as the first argument (array) in each dimension. Array indexing is relative for all such input array arguments.

1.11.3 Kernel shape inference and border handling

In the above example and in most cases, the array indexing in the stencil kernel will exclusively use `Integer` literals. In such cases, the stencil decorator is able to analyze the stencil kernel to determine its size. In the above example, the stencil decorator determines that the kernel is `3 x 3` in shape since indices `-1` to `1` are used for both the first and second dimensions. Note that the stencil decorator also correctly handles non-symmetric and non-square stencil kernels.

Based on the size of the stencil kernel, the stencil decorator is able to compute the size of the border in the output array. If applying the kernel to some element of input array would cause an index to be out-of-bounds then that element belongs to the border of the output array. In the above example, points `-1` and `+1` are accessed in each dimension and thus the output array has a border of size one in all dimensions.

The parallel mode is able to infer kernel indices as constants from simple expressions if possible. For example:

```python
@njit(parallel=True)
def stencil_test(A):
    c = 2
    B = stencil(
        lambda a, c: 0.3 * (a[-c+1] + a[0] + a[c-1]))(A, c)
    return B
```

1.11.4 Stencil decorator options

Note: The stencil decorator may be augmented in the future to provide additional mechanisms for border handling. At present, only one behaviour is implemented, `"constant"` (see `func_or_mode` below for details).

---

neighborhood

Sometimes it may be inconvenient to write the stencil kernel exclusively with `Integer` literals. For example, let us say we would like to compute the trailing 30-day moving average of a time series of data. One could write `(a[-29] + a[-28] + ... + a[-1] + a[0]) / 30` but the stencil decorator offers a more concise form using the `neighborhood` option:

```python
@stencil(neighborhood = ((-29, 0),))
def kernel2(a):
    cumul = 0
    for i in range(-29, 1):
        cumul += a[i]
    return cumul / 30
```

The neighborhood option is a tuple of tuples. The outer tuple's length is equal to the number of dimensions of the input array. The inner tuple's lengths are always two because each element of the outer tuple corresponds to minimum and maximum index offsets used in the corresponding dimension.

If a user specifies a neighborhood but the kernel accesses elements outside the specified neighborhood, **the behavior is undefined.**

func_or_mode

The optional `func_or_mode` parameter controls how the border of the output array is handled. Currently, there is only one supported value, `"constant"`. In `constant` mode, the stencil kernel is not applied in cases where the kernel would access elements outside the valid range of the input array. In such cases, those elements in the output array are assigned to a constant value, as specified by the `cval` parameter.

cval

The optional cval parameter defaults to zero but can be set to any desired value, which is then used for the border of the output array if the `func_or_mode` parameter is set to `constant`. The cval parameter is ignored in all other modes. The type of the cval parameter must match the return type of the stencil kernel. If the user wishes the output array to be constructed from a particular type then they should ensure that the stencil kernel returns that type.

standard_indexing

By default, all array accesses in a stencil kernel are processed as relative indices as described above. However, sometimes it may be advantageous to pass an auxiliary array (e.g. an array of weights) to a stencil kernel and have that array use standard Python indexing rather than relative indexing. For this purpose, there is the stencil decorator option `standard_indexing` whose value is a collection of strings whose names match those parameters to the stencil function that are to be accessed with standard Python indexing rather than relative indexing:

```
@stencil(standard_indexing=("b",))
def kernel3(a, b):
    return a[-1] * b[0] + a[0] + b[1]
```

1.11.5 `StencilFunc`

The stencil decorator returns a callable object of type `StencilFunc`. `StencilFunc` objects contains a number of attributes but the only one of potential interest to users is the `neighborhood` attribute. If the `neighborhood` option was passed to the stencil decorator then the provided neighborhood is stored in this attribute. Else, upon first execution or compilation, the system calculates the neighborhood as described above and then stores the computed neighborhood into this attribute. A user may then inspect the attribute if they wish to verify that the calculated neighborhood is correct.

1.11.6 Stencil invocation options

Internally, the stencil decorator transforms the specified stencil kernel into a regular Python function. This function will have the same parameters as specified in the stencil kernel definition but will also include the following optional parameter.

out

The optional `out` parameter is added to every stencil function generated by Numba. If specified, the `out` parameter tells Numba that the user is providing their own pre-allocated array to be used for the output of the stencil. In this case, the stencil function will not allocate its own output array. Users should assure that the return type of the stencil kernel can be safely cast to the element-type of the user-specified output array following the Numpy ufunc casting rules.

An example usage is shown below:

```
>>> import numpy as np
>>> input_arr = np.arange(100).reshape((10, 10))
>>> output_arr = np.full(input_arr.shape, 0.0)
>>> kernel1(input_arr, out=output_arr)
```

1.12 Callback into the Python Interpreter from within JIT'ed code

There are rare but real cases when a nopython-mode function needs to callback into the Python interpreter to invoke code that cannot be compiled by Numba. Such cases include:

- logging progress for long running JIT'ed functions;
- use data structures that are not currently supported by Numba;
- debugging inside JIT'ed code using the Python debugger.

When Numba callbacks into the Python interpreter, the following has to happen:

- acquire the GIL;
- convert values in native representation back into Python objects;
- call-back into the Python interpreter;
- convert returned values from the Python-code into native representation;

- release the GIL.

These steps can be expensive. Users **should not** rely on the feature described here on performance-critical paths.

1.12.1 The `objmode` context-manager

> **Warning:** This feature can be easily mis-used. Users should first consider alternative approaches to achieve their intended goal before using this feature.

numba.**objmode**(**args*, ***kwargs*)

> Creates a contextmanager to be used inside jitted functions to enter *object-mode* for using interpreter features. The body of the with-context is lifted into a function that is compiled in *object-mode*. This transformation process is limited and cannot process all possible Python code. However, users can wrap complicated logic in another Python function, which will then be executed by the interpreter.
>
> Use this as a function that takes keyword arguments only. The argument names must correspond to the output variables from the with-block. Their respective values are strings representing the expected types. When exiting the with-context, the output variables are cast to the expected nopython types according to the annotation. This process is the same as passing Python objects into arguments of a nopython function.
>
> Example:

```python
import numpy as np
from numba import njit, objmode

def bar(x):
    # This code is executed by the interpreter.
    return np.asarray(list(reversed(x.tolist())))

@njit
def foo():
    x = np.arange(5)
    y = np.zeros_like(x)
    with objmode(y='intp[:]'):  # annotate return type
        # this region is executed by object-mode.
        y += bar(x)
    return y
```

> **Note:** Known limitations:
>
> - with-block cannot use incoming list objects.
>
> - with-block cannot use incoming function objects.
>
> - with-block cannot `yield`, `break`, `return` or `raise` such that the execution will leave the with-block immediately.
>
> - with-block cannot contain *with* statements.
>
> - random number generator states do not synchronize; i.e. nopython-mode and object-mode uses different RNG states.

> **Note:** When used outside of no-python mode, the context-manager has no effect.

> **Warning:** This feature is experimental. The supported features may change with or without notice.

1.13 Automatic module jitting with `jit_module`

A common usage pattern is to have an entire module containing user-defined functions that all need to be jitted. One option to accomplish this is to manually apply the `@jit` decorator to each function definition. This approach works and is great in many cases. However, for large modules with many functions, manually `jit`-wrapping each function definition can be tedious. For these situations, Numba provides another option, the `jit_module` function, to automatically replace functions declared in a module with their `jit`-wrapped equivalents.

It's important to note the conditions under which `jit_module` will *not* impact a function:

1. Functions which have already been wrapped with a Numba decorator (e.g. `jit`, `vectorize`, `cfunc`, etc.) are not impacted by `jit_module`.

2. Functions which are declared outside the module from which `jit_module` is called are not automatically `jit`-wrapped.

3. Function declarations which occur logically after calling `jit_module` are not impacted.

All other functions in a module will have the `@jit` decorator automatically applied to them. See the following section for an example use case.

> **Note:** This feature is for use by module authors. `jit_module` should not be called outside the context of a module containing functions to be jitted.

1.13.1 Example usage

Let's assume we have a Python module we've created, `mymodule.py` (shown below), which contains several functions. Some of these functions are defined in `mymodule.py` while others are imported from other modules. We wish to have all the functions which are defined in `mymodule.py` jitted using `jit_module`.

```python
# mymodule.py

from numba import jit, jit_module

def inc(x):
    return x + 1

def add(x, y):
    return x + y

import numpy as np
# Use NumPy's mean function
mean = np.mean

@jit(nogil=True)
def mul(a, b):
    return a * b

jit_module(nopython=True, error_model="numpy")
```

(continues on next page)

```
def div(a, b):
    return a / b
```

There are several things to note in the above example:

- Both the `inc` and `add` functions will be replaced with their `jit`-wrapped equivalents with *compilation options* `nopython=True` and `error_model="numpy"`.

- The `mean` function, because it's defined *outside* of `mymodule.py` in NumPy, will not be modified.

- `mul` will not be modified because it has been manually decorated with `jit`.

- `div` will not be automatically `jit`-wrapped because it is declared after `jit_module` is called.

When the above module is imported, we have:

```
>>> import mymodule
>>> mymodule.inc
CPUDispatcher(<function inc at 0x1032f86a8>)
>>> mymodule.mean
<function mean at 0x1096b8950>
```

1.13.2 API

> **Warning:** This feature is experimental. The supported features may change with or without notice.

numba.**jit_module**(***kwargs*)

> Automatically `jit`-wraps functions defined in a Python module
>
> Note that `jit_module` should only be called at the end of the module to be jitted. In addition, only functions which are defined in the module `jit_module` is called from are considered for automatic jit-wrapping. See the Numba documentation for more information about what can/cannot be jitted.
>
> > **Parameters** `kwargs` – Keyword arguments to pass to `jit` such as `nopython` or `error_model`.

1.14 Performance Tips

This is a short guide to features present in Numba that can help with obtaining the best performance from code. Two examples are used, both are entirely contrived and exist purely for pedagogical reasons to motivate discussion. The first is the computation of the trigonometric identity $cos(x)^2 + sin(x)^2$, the second is a simple element wise square root of a vector with reduction over summation. All performance numbers are indicative only and unless otherwise stated were taken from running on an Intel `i7-4790` CPU (4 hardware threads) with an input of `np.arange(1.e7)`.

> **Note:** A reasonably effective approach to achieving high performance code is to profile the code running with real data and use that to guide performance tuning. The information presented here is to demonstrate features, not to act as canonical guidance!

1.14.1 No Python mode vs Object mode

A common pattern is to decorate functions with `@jit` as this is the most flexible decorator offered by Numba. `@jit` essentially encompasses two modes of compilation, first it will try and compile the decorated function in no Python mode, if this fails it will try again to compile the function using object mode. Whilst the use of looplifting in object mode can enable some performance increase, getting functions to compile under no python mode is really the key to good performance. To make it such that only no python mode is used and if compilation fails an exception is raised the decorators `@njit` and `@jit(nopython=True)` can be used (the first is an alias of the second for convenience).

1.14.2 Loops

Whilst NumPy has developed a strong idiom around the use of vector operations, Numba is perfectly happy with loops too. For users familiar with C or Fortran, writing Python in this style will work fine in Numba (after all, LLVM gets a lot of use in compiling C lineage languages). For example:

```
@njit
def ident_np(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2


@njit
def ident_loops(x):
    r = np.empty_like(x)
    n = len(x)
    for i in range(n):
        r[i] = np.cos(x[i]) ** 2 + np.sin(x[i]) ** 2
    return r
```

The above run at almost identical speeds when decorated with `@njit`, without the decorator the vectorized function is a couple of orders of magnitude faster.

Function Name	@njit	Execution time
`ident_np`	No	0.581s
`ident_np`	Yes	0.659s
`ident_loops`	No	25.2s
`ident_loops`	Yes	0.670s

1.14.3 Fastmath

In certain classes of applications strict IEEE 754 compliance is less important. As a result it is possible to relax some numerical rigour with view of gaining additional performance. The way to achieve this behaviour in Numba is through the use of the `fastmath` keyword argument:

```
@njit(fastmath=False)
def do_sum(A):
    acc = 0.
    # without fastmath, this loop must accumulate in strict order
    for x in A:
        acc += np.sqrt(x)
    return acc


@njit(fastmath=True)
def do_sum_fast(A):
    acc = 0.
```

```
    # with fastmath, the reduction can be vectorized as floating point
    # reassociation is permitted.
    for x in A:
        acc += np.sqrt(x)
    return acc
```

Function Name	Execution time
do_sum	35.2 ms
do_sum_fast	17.8 ms

In some cases you may wish to opt-in to only a subset of possible fast-math optimizations. This can be done by supplying a set of LLVM fast-math flags to `fastmath`.:

```
def add_assoc(x, y):
    return (x - y) + y

print(njit(fastmath=False)(add_assoc)(0, np.inf)) # nan
print(njit(fastmath=True) (add_assoc)(0, np.inf)) # 0.0
print(njit(fastmath={'reassoc', 'nsz'})(add_assoc)(0, np.inf)) # 0.0
print(njit(fastmath={'reassoc'})        (add_assoc)(0, np.inf)) # nan
print(njit(fastmath={'nsz'})            (add_assoc)(0, np.inf)) # nan
```

1.14.4 Parallel=True

If code contains operations that are parallelisable (*and supported*) Numba can compile a version that will run in parallel on multiple native threads (no GIL!). This parallelisation is performed automatically and is enabled by simply adding the `parallel` keyword argument:

```
@njit(parallel=True)
def ident_parallel(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2
```

Executions times are as follows:

Function Name	Execution time
ident_parallel	112 ms

The execution speed of this function with `parallel=True` present is approximately 5x that of the NumPy equivalent and 6x that of standard `@njit`.

Numba parallel execution also has support for explicit parallel loop declaration similar to that in OpenMP. To indicate that a loop should be executed in parallel the `numba.prange` function should be used, this function behaves like Python `range` and if `parallel=True` is not set it acts simply as an alias of `range`. Loops induced with `prange` can be used for embarrassingly parallel computation and also reductions.

Revisiting the reduce over sum example, assuming it is safe for the sum to be accumulated out of order, the loop in n can be parallelised through the use of `prange`. Further, the `fastmath=True` keyword argument can be added without concern in this case as the assumption that out of order execution is valid has already been made through the use of `parallel=True` (as each thread computes a partial sum).

```
@njit(parallel=True)
def do_sum_parallel(A):
```

```
    # each thread can accumulate its own partial sum, and then a cross
    # thread reduction is performed to obtain the result to return
    n = len(A)
    acc = 0.
    for i in prange(n):
        acc += np.sqrt(A[i])
    return acc

@njit(parallel=True, fastmath=True)
def do_sum_parallel_fast(A):
    n = len(A)
    acc = 0.
    for i in prange(n):
        acc += np.sqrt(A[i])
    return acc
```

Execution times are as follows, `fastmath` again improves performance.

Function Name	Execution time
`do_sum_parallel`	9.81 ms
`do_sum_parallel_fast`	5.37 ms

1.14.5 Intel SVML

Intel provides a short vector math library (SVML) that contains a large number of optimised transcendental functions available for use as compiler intrinsics. If the `icc_rt` package is present in the environment (or the SVML libraries are simply locatable!) then Numba automatically configures the LLVM back end to use the SVML intrinsic functions where ever possible. SVML provides both high and low accuracy versions of each intrinsic and the version that is used is determined through the use of the `fastmath` keyword. The default is to use high accuracy which is accurate to within `1 ULP`, however if `fastmath` is set to `True` then the lower accuracy versions of the intrinsics are used (answers to within `4 ULP`).

First obtain SVML, using conda for example:

```
conda install -c numba icc_rt
```

Rerunning the identity function example `ident_np` from above with various combinations of options to `@njit` and with/without SVML yields the following performance results (input size `np.arange(1.e8)`). For reference, with just NumPy the function executed in `5.84s`:

`@njit` kwargs	SVML	Execution time
`None`	No	5.95s
`None`	Yes	2.26s
`fastmath=True`	No	5.97s
`fastmath=True`	Yes	1.8s
`parallel=True`	No	1.36s
`parallel=True`	Yes	0.624s
`parallel=True, fastmath=True`	No	1.32s
`parallel=True, fastmath=True`	Yes	0.576s

It is evident that SVML significantly increases the performance of this function. The impact of `fastmath` in the case of SVML not being present is zero, this is expected as there is nothing in the original function that would benefit from relaxing numerical strictness.

1.14.6 Linear algebra

Numba supports most of `numpy.linalg` in no Python mode. The internal implementation relies on a LAPACK and BLAS library to do the numerical work and it obtains the bindings for the necessary functions from SciPy. Therefore, to achieve good performance in `numpy.linalg` functions with Numba it is necessary to use a SciPy built against a well optimised LAPACK/BLAS library. In the case of the Anaconda distribution SciPy is built against Intel's MKL which is highly optimised and as a result Numba makes use of this performance.

1.15 The Threading Layers

This section is about the Numba threading layer, this is the library that is used internally to perform the parallel execution that occurs through the use of the `parallel` targets for CPUs, namely:

- The use of the `parallel=True` kwarg in `@jit` and `@njit`.

- The use of the `target='parallel'` kwarg in `@vectorize` and `@guvectorize`.

Note: If a code base does not use the `threading` or `multiprocessing` modules (or any other sort of parallelism) the defaults for the threading layer that ship with Numba will work well, no further action is required!

1.15.1 Which threading layers are available?

There are three threading layers available and they are named as follows:

- `tbb` - A threading layer backed by Intel TBB.

- `omp` - A threading layer backed by OpenMP.

- `workqueue` -A simple built-in work-sharing task scheduler.

In practice, the only threading layer guaranteed to be present is `workqueue`. The `omp` layer requires the presence of a suitable OpenMP runtime library. The `tbb` layer requires the presence of Intel's TBB libraries, these can be obtained via the conda command:

```
$ conda install tbb
```

If you installed Numba with `pip`, TBB can be enabled by running:

```
$ pip install tbb
```

Due to compatibility issues with manylinux1 and other portability concerns, the OpenMP threading layer is disabled in the Numba binary wheels on PyPI.

Note: The default manner in which Numba searches for and loads a threading layer is tolerant of missing libraries, incompatible runtimes etc.

1.15.2 Setting the threading layer

The threading layer is set via the environment variable `NUMBA_THREADING_LAYER` or through assignment to `numba.config.THREADING_LAYER`. If the programmatic approach to setting the threading layer is used it must occur logically before any Numba based compilation for a parallel target has occurred. There are two approaches

to choosing a threading layer, the first is by selecting a threading layer that is safe under various forms of parallel execution, the second is through explicit selection via the threading layer name (e.g. `tbb`).

Selecting a threading layer for safe parallel execution

Parallel execution is fundamentally derived from core Python libraries in four forms (the first three also apply to code using parallel execution via other means!):

- `threads` from the `threading` module.
- `spawn` ing processes from the `multiprocessing` module via `spawn` (default on Windows, only available in Python 3.4+ on Unix)
- `fork` ing processes from the `multiprocessing` module via `fork` (default on Unix).
- `fork` ing processes from the `multiprocessing` module through the use of a `forkserver` (only available in Python 3 on Unix). Essentially a new process is spawned and then forks are made from this new process on request.

Any library in use with these forms of parallelism must exhibit safe behaviour under the given paradigm. As a result, the threading layer selection methods are designed to provide a way to choose a threading layer library that is safe for a given paradigm in an easy, cross platform and environment tolerant manner. The options that can be supplied to the *setting mechanisms* are as follows:

- `default` provides no specific safety guarantee and is the default.
- `safe` is both fork and thread safe, this requires the `tbb` package (Intel TBB libraries) to be installed.
- `forksafe` provides a fork safe library.
- `threadsafe` provides a thread safe library.

To discover the threading layer that was selected, the function `numba.threading_layer()` may be called after parallel execution. For example, on a Linux machine with no TBB installed:

```python
from numba import config, njit, threading_layer
import numpy as np

# set the threading layer before any parallel target compilation
config.THREADING_LAYER = 'threadsafe'

@njit(parallel=True)
def foo(a, b):
    return a + b

x = np.arange(10.)
y = x.copy()

# this will force the compilation of the function, select a threading layer
# and then execute in parallel
foo(x, y)

# demonstrate the threading layer chosen
print("Threading layer chosen: %s" % threading_layer())
```

which produces:

```
Threading layer chosen: omp
```

and this makes sense as GNU OpenMP, as present on Linux, is thread safe.

Selecting a named threading layer

Advanced users may wish to select a specific threading layer for their use case, this is done by directly supplying the threading layer name to the *setting mechanisms*. The options and requirements are as follows:

Threading Layer Name	Platform	Requirements
tbb	All	The `tbb` package (`$ conda install tbb`)
omp	Linux	GNU OpenMP libraries (very likely this will already exist)
	Windows	MS OpenMP libraries (very likely this will already exist)
	OSX	The `intel-openmp` package (`$ conda install intel-openmp`)
workqueue	All	None

Should the threading layer not load correctly Numba will detect this and provide a hint about how to resolve the problem. It should also be noted that the Numba diagnostic command `numba -s` has a section `__Threading Layer Information__` that reports on the availability of threading layers in the current environment.

1.15.3 Extra notes

The threading layers have fairly complex interactions with CPython internals and system level libraries, some additional things to note:

- The installation of Intel's TBB libraries vastly widens the options available in the threading layer selection process.

- On Linux, the `omp` threading layer is not fork safe due to the GNU OpenMP runtime library (`libgomp`) not being fork safe. If a fork occurs in a program that is using the `omp` threading layer, a detection mechanism is present that will try and gracefully terminate the forked child and print an error message to `STDERR`.

- On OSX, the `intel-openmp` package is required to enable the OpenMP based threading layer.

1.15.4 Setting the Number of Threads

The number of threads used by numba is based on the number of CPU cores available (see *numba.config. NUMBA_DEFAULT_NUM_THREADS*), but it can be overridden with the *NUMBA_NUM_THREADS* environment variable.

The total number of threads that numba launches is in the variable *numba.config.NUMBA_NUM_THREADS*.

For some use cases, it may be desirable to set the number of threads to a lower value, so that numba can be used with higher level parallelism.

The number of threads can be set dynamically at runtime using *numba.set_num_threads()*. Note that *set_num_threads()* only allows setting the number of threads to a smaller value than *NUMBA_NUM_THREADS*. Numba always launches *numba.config.NUMBA_NUM_THREADS* threads, but *set_num_threads()* causes it to mask out unused threads so they aren't used in computations.

The current number of threads used by numba can be accessed with *numba.get_num_threads()*. Both functions work inside of a jitted function.

Example of Limiting the Number of Threads

In this example, suppose the machine we are running on has 8 cores (so *numba.config.NUMBA_NUM_THREADS* would be 8). Suppose we want to run some code with `@njit(parallel=True)`, but we also want to run our code

concurrently in 4 different processes. With the default number of threads, each Python process would run 8 threads, for a total in 4*8 = 32 threads, which is oversubscription for our 8 cores. We should rather limit each process to 2 threads, so that the total will be 4*2 = 8, which matches our number of physical cores.

There are two ways to do this. One is to set the *NUMBA_NUM_THREADS* environment variable to 2.

```
$ NUMBA_NUM_THREADS=2 python ourcode.py
```

However, there are two downsides to this approach:

1. *NUMBA_NUM_THREADS* must be set before Numba is imported, and ideally before Python is launched. As soon as Numba is imported the environment variable is read and that number of threads is locked in as the number of threads Numba launches.

2. If we want to later increase the number of threads used by the process, we cannot. *NUMBA_NUM_THREADS* sets the *maximum* number of threads that are launched for a process. Calling *set_num_threads()* with a value greater than *numba.config.NUMBA_NUM_THREADS* results in an error.

The advantage of this approach is that we can do it from outside of the process without changing the code.

Another approach is to use the *numba.set_num_threads()* function in our code

```python
from numba import njit, set_num_threads

@njit(parallel=True)
def func():
    ...

set_num_threads(2)
func()
```

If we call `set_num_threads(2)` before executing our parallel code, it has the same effect as calling the process with `NUMBA_NUM_THREADS=2`, in that the parallel code will only execute on 2 threads. However, we can later call `set_num_threads(8)` to increase the number of threads back to the default size. And we do not have to worry about setting it before Numba gets imported. It only needs to be called before the parallel function is run.

API Reference

numba.config.**NUMBA_NUM_THREADS**
> The total (maximum) number of threads launched by numba.
>
> Defaults to *numba.config.NUMBA_DEFAULT_NUM_THREADS*, but can be overridden with the *NUMBA_NUM_THREADS* environment variable.

numba.config.**NUMBA_DEFAULT_NUM_THREADS**
> The number of CPU cores on the system (as determined by `multiprocessing.cpu_count()`). This is the default value for *numba.config.NUMBA_NUM_THREADS* unless the *NUMBA_NUM_THREADS* environment variable is set.

numba.**set_num_threads**(*n*)
> Set the number of threads to use for parallel execution.
>
> By default, all *numba.config.NUMBA_NUM_THREADS* threads are used.
>
> This functionality works by masking out threads that are not used. Therefore, the number of threads *n* must be less than or equal to *NUMBA_NUM_THREADS*, the total number of threads that are launched. See its documentation for more details.
>
> This function can be used inside of a jitted function.

>   **Parameters**

>   **n: The number of threads. Must be between 1 and NUMBA_NUM_THREADS.**

>   **See also:**

>   *get_num_threads*, *numba.config.NUMBA_NUM_THREADS*

>   *numba.config.NUMBA_DEFAULT_NUM_THREADS*, *NUMBA_NUM_THREADS*

numba.**get_num_threads**()
>   Get the number of threads used for parallel execution.

>   By default (if *set_num_threads()* is never called), all *numba.config.NUMBA_NUM_THREADS* threads are used.

>   This number is less than or equal to the total number of threads that are launched, *numba.config.NUMBA_NUM_THREADS*.

>   This function can be used inside of a jitted function.

>   **Returns**

>   **The number of threads.**

>   **See also:**

>   *set_num_threads*, *numba.config.NUMBA_NUM_THREADS*

>   *numba.config.NUMBA_DEFAULT_NUM_THREADS*, *NUMBA_NUM_THREADS*

1.16 Command line interface

Numba is a Python package, usually you `import numba` from Python and use the Python application programming interface (API). However, Numba also ships with a command line interface (CLI), i.e. a tool `numba` that is installed when you install Numba.

Currently, the only purpose of the CLI is to allow you to quickly show some information about your system and installation, or to quickly get some debugging information for a Python script using Numba.

1.16.1 Usage

To use the Numba CLI from the terminal, use `numba` followed by the options and arguments like `--help` or `-s`, as explained below.

Sometimes it can happen that you get a "command not found" error when you type `numba`, because your `PATH` isn't configured properly. In that case you can use the equivalent command `python -m numba`. If that still gives "command not found", try to `import numba` as suggested here: *Dependency List*.

The two versions `numba` and `python -m numba` are the same. The first is shorter to type, but if you get a "command not found" error because your `PATH` doesn't contain the location where `numba` is installed, having the `python -m numba` variant is useful.

To use the Numba CLI from IPython or Jupyter, use `!numba`, i.e. prefix the command with an exclamation mark. This is a general IPython/Jupyter feature to execute shell commands, it is not available in the regular `python` terminal.

1.16.2 Help

To see all available options, use `numba --help`:

```
$ numba --help
usage: numba [-h] [--annotate] [--dump-llvm] [--dump-optimized]
             [--dump-assembly] [--dump-cfg] [--dump-ast]
             [--annotate-html ANNOTATE_HTML] [-s]
             [filename]

positional arguments:
  filename              Python source filename

optional arguments:
  -h, --help            show this help message and exit
  --annotate            Annotate source
  --dump-llvm           Print generated llvm assembly
  --dump-optimized      Dump the optimized llvm assembly
  --dump-assembly       Dump the LLVM generated assembly
  --dump-cfg            [Deprecated] Dump the control flow graph
  --dump-ast            [Deprecated] Dump the AST
  --annotate-html ANNOTATE_HTML
                        Output source annotation as html
  -s, --sysinfo         Output system information for bug reporting
```

1.16.3 System information

The `numba -s` (or the equivalent `numba --sysinfo`) command prints a lot of information about your system
and your Numba installation and relevant dependencies.

Remember: you can use `!numba -s` with an exclamation mark to see this information from IPython or Jupyter.

Example output:

```
$ numba -s

System info:
--------------------------------------------------------------------------------
__Time Stamp__
2019-05-07 14:15:39.733994


__Hardware Information__
Machine                                       : x86_64
CPU Name                                      : haswell
CPU count                                     : 8
CPU Features                                  :
aes avx avx2 bmi bmi2 cmov cx16 f16c fma fsgsbase invpcid lzcnt mmx movbe pclmul
popcnt rdrnd sahf sse sse2 sse3 sse4.1 sse4.2 ssse3 xsave xsaveopt


__OS Information__
Platform                                      : Darwin-18.5.0-x86_64-i386-64bit
Release                                       : 18.5.0
System Name                                   : Darwin
Version                                       : Darwin Kernel Version 18.5.0: Mon Mar␣
→11 20:40:32 PDT 2019; root:xnu-4903.251.3~3/RELEASE_X86_64
OS specific info                              : 10.14.4   x86_64
```

(continues on next page)

```
__Python Information__
Python Compiler                                   : Clang 4.0.1 (tags/RELEASE_401/final)
Python Implementation                             : CPython
Python Version                                    : 3.7.3
Python Locale                                     : en_US UTF-8


__LLVM information__
LLVM version                                      : 7.0.0

__CUDA Information__
CUDA driver library cannot be found or no CUDA enabled devices are present.
Error class: <class 'numba.cuda.cudadrv.error.CudaSupportError'>


__ROC Information__
ROC available                                     : False
Error initialising ROC due to                     : No ROC toolchains found.
No HSA Agents found, encountered exception when searching:
Error at driver init:

HSA is not currently supported on this platform (darwin).
:


__SVML Information__
SVML state, config.USING_SVML                     : False
SVML library found and loaded                     : False
llvmlite using SVML patched LLVM                   : True
SVML operational                                  : False


__Threading Layer Information__
TBB Threading layer available                     : False
+--> Disabled due to                              : Unknown import problem.
OpenMP Threading layer available                  : False
+--> Disabled due to                              : Unknown import problem.
Workqueue Threading layer available               : True


__Numba Environment Variable Information__
None set.


__Conda Information__
conda_build_version                               : 3.17.8
conda_env_version                                 : 4.6.14
platform                                          : osx-64
python_version                                    : 3.7.3.final.0
root_writable                                     : True


__Current Conda Env__
(output truncated due to length)
```

1.16.4 Debugging

As shown in the help output above, the `numba` command includes options that can help you to debug Numba compiled code.

To try it out, create an example script called `myscript.py`:

```
import numba

@numba.jit
def f(x):
    return 2 * x

f(42)
```

and then execute one of the following commands:

```
$ numba myscript.py --annotate
$ numba myscript.py --annotate-html myscript.html
$ numba myscript.py --dump-llvm
$ numba myscript.py --dump-optimized
$ numba myscript.py --dump-assembly
```

1.17 Troubleshooting and tips

1.17.1 What to compile

The general recommendation is that you should only try to compile the critical paths in your code. If you have a piece of performance-critical computational code amongst some higher-level code, you may factor out the performance-critical code in a separate function and compile the separate function with Numba. Letting Numba focus on that small piece of performance-critical code has several advantages:

- it reduces the risk of hitting unsupported features;

- it reduces the compilation times;

- it allows you to evolve the higher-level code which is outside of the compiled function much easier.

1.17.2 My code doesn't compile

There can be various reasons why Numba cannot compile your code, and raises an error instead. One common reason is that your code relies on an unsupported Python feature, especially in *nopython mode*. Please see the list of *Supported Python features*. If you find something that is listed there and still fails compiling, please *report a bug*.

When Numba tries to compile your code it first tries to work out the types of all the variables in use, this is so it can generate a type specific implementation of your code that can be compiled down to machine code. A common reason for Numba failing to compile (especially in *nopython mode*) is a type inference failure, essentially Numba cannot work out what the type of all the variables in your code should be.

For example, let's consider this trivial function:

```
@jit(nopython=True)
def f(x, y):
    return x + y
```

If you call it with two numbers, Numba is able to infer the types properly:

```
>>> f(1, 2)
    3
```

If however you call it with a tuple and a number, Numba is unable to say what the result of adding a tuple and number is, and therefore compilation errors out:

```
>>> f(1, (2,))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<path>/numba/numba/dispatcher.py", line 339, in _compile_for_args
    reraise(type(e), e, None)
File "<path>/numba/numba/six.py", line 658, in reraise
    raise value.with_traceback(tb)
numba.errors.TypingError: Failed at nopython (nopython frontend)
Invalid use of + with parameters (int64, tuple(int64 x 1))
Known signatures:
* (int64, int64) -> int64
* (int64, uint64) -> int64
* (uint64, int64) -> int64
* (uint64, uint64) -> uint64
* (float32, float32) -> float32
* (float64, float64) -> float64
* (complex64, complex64) -> complex64
* (complex128, complex128) -> complex128
* (uint16,) -> uint64
* (uint8,) -> uint64
* (uint64,) -> uint64
* (uint32,) -> uint64
* (int16,) -> int64
* (int64,) -> int64
* (int8,) -> int64
* (int32,) -> int64
* (float32,) -> float32
* (float64,) -> float64
* (complex64,) -> complex64
* (complex128,) -> complex128
* parameterized
[1] During: typing of intrinsic-call at <stdin> (3)

File "<stdin>", line 3:
```

The error message helps you find out what went wrong: "Invalid use of + with parameters (int64, tuple(int64 x 1))" is to be interpreted as "Numba encountered an addition of variables typed as integer and 1-tuple of integer, respectively, and doesn't know about any such operation".

Note that if you allow object mode:

```python
@jit
def g(x, y):
    return x + y
```

compilation will succeed and the compiled function will raise at runtime as Python would do:

```python
>>> g(1, (2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'tuple'
```

1.17.3 My code has a type unification problem

Another common reason for Numba not being able to compile your code is that it cannot statically determine the return type of a function. The most likely cause of this is the return type depending on a value that is available only at

runtime. Again, this is most often problematic when using *nopython mode*. The concept of type unification is simply trying to find a type in which two variables could safely be represented. For example a 64 bit float and a 64 bit complex number could both be represented in a 128 bit complex number.

As an example of type unification failure, this function has a return type that is determined at runtime based on the value of *x*:

```
In [1]: from numba import jit

In [2]: @jit(nopython=True)
...: def f(x):
...:     if x > 10:
...:         return (1,)
...:     else:
...:         return 1
...:

In [3]: f(10)
```

Trying to execute this function, errors out as follows:

```
TypingError: Failed at nopython (nopython frontend)
Can't unify return type from the following types: tuple(int64 x 1), int64
Return of: IR name '$8.2', type '(int64 x 1)', location:
File "<ipython-input-2-51ef1cc64bea>", line 4:
def f(x):
    <source elided>
    if x > 10:
        return (1,)
        ^
Return of: IR name '$12.2', type 'int64', location:
File "<ipython-input-2-51ef1cc64bea>", line 6:
def f(x):
    <source elided>
    else:
        return 1
```

The error message "Can't unify return type from the following types: tuple(int64 x 1), int64" should be read as "Numba cannot find a type that can safely represent a 1-tuple of integer and an integer".

1.17.4 My code has an untyped list problem

As *noted previously* the first part of Numba compiling your code involves working out what the types of all the variables are. In the case of lists, a list must contain items that are of the same type or can be empty if the type can be inferred from some later operation. What is not possible is to have a list which is defined as empty and has no inferable type (i.e. an untyped list).

For example, this is using a list of a known type:

```
from numba import jit
@jit(nopython=True)
def f():
    return [1, 2, 3] # this list is defined on construction with `int` type
```

This is using an empty list, but the type can be inferred:

```python
from numba import jit
@jit(nopython=True)
def f(x):
    tmp = [] # defined empty
    for i in range(x):
        tmp.append(i) # list type can be inferred from the type of `i`
    return tmp
```

This is using an empty list and the type cannot be inferred:

```python
from numba import jit
@jit(nopython=True)
def f(x):
    tmp = [] # defined empty
    return (tmp, x) # ERROR: the type of `tmp` is unknown
```

Whilst slightly contrived, if you need an empty list and the type cannot be inferred but you know what type you want the list to be, this "trick" can be used to instruct the typing mechanism:

```python
from numba import jit
import numpy as np
@jit(nopython=True)
def f(x):
    # define empty list, but instruct that the type is np.complex64
    tmp = [np.complex64(x) for x in range(0)]
    return (tmp, x) # the type of `tmp` is known, but it is still empty
```

1.17.5 The compiled code is too slow

The most common reason for slowness of a compiled JIT function is that compiling in *nopython mode* has failed and the Numba compiler has fallen back to *object mode*. *object mode* currently provides little to no speedup compared to regular Python interpretation, and its main point is to allow an internal optimization known as *loop-lifting*: this optimization will allow to compile inner loops in *nopython mode* regardless of what code surrounds those inner loops.

To find out if type inference succeeded on your function, you can use the *inspect_types()* method on the compiled function.

For example, let's take the following function:

```python
@jit
def f(a, b):
    s = a + float(b)
    return s
```

When called with numbers, this function should be fast as Numba is able to convert number types to floating-point numbers. Let's see:

```python
>>> f(1, 2)
3.0
>>> f.inspect_types()
f (int64, int64)
--------------------------------------------------------------------------------
# --- LINE 7 ---

@jit
```

(continues on next page)

```
# --- LINE 8 ---

def f(a, b):

    # --- LINE 9 ---
    # label 0
    #   a.1 = a  :: int64
    #   del a
    #   b.1 = b  :: int64
    #   del b
    #   $0.2 = global(float: <class 'float'>)  :: Function(<class 'float'>)
    #   $0.4 = call $0.2(b.1, )  :: (int64,) -> float64
    #   del b.1
    #   del $0.2
    #   $0.5 = a.1 + $0.4  :: float64
    #   del a.1
    #   del $0.4
    #   s = $0.5  :: float64
    #   del $0.5

    s = a + float(b)

    # --- LINE 10 ---
    #   $0.7 = cast(value=s)  :: float64
    #   del s
    #   return $0.7

    return s
```

Without trying to understand too much of the Numba intermediate representation, it is still visible that all variables and temporary values have had their types inferred properly: for example *a* has the type int64, *$0.5* has the type float64, etc.

However, if *b* is passed as a string, compilation will fall back on object mode as the float() constructor with a string is currently not supported by Numba:

```
>>> f(1, "2")
3.0
>>> f.inspect_types()
[... snip annotations for other signatures, see above ...]
================================================================================
f (int64, str)
--------------------------------------------------------------------------------
# --- LINE 7 ---

@jit

# --- LINE 8 ---

def f(a, b):

    # --- LINE 9 ---
    # label 0
    #   a.1 = a  :: pyobject
    #   del a
    #   b.1 = b  :: pyobject
    #   del b
```

```
    #   $0.2 = global(float: <class 'float'>)  :: pyobject
    #   $0.4 = call $0.2(b.1, )  :: pyobject
    #   del b.1
    #   del $0.2
    #   $0.5 = a.1 + $0.4  :: pyobject
    #   del a.1
    #   del $0.4
    #   s = $0.5  :: pyobject
    #   del $0.5

    s = a + float(b)

    # --- LINE 10 ---
    #   $0.7 = cast(value=s)  :: pyobject
    #   del s
    #   return $0.7

    return s
```

Here we see that all variables end up typed as `pyobject`. This means that the function was compiled in object mode and values are passed around as generic Python objects, without Numba trying to look into them to reason about their raw values. This is a situation you want to avoid when caring about the speed of your code.

If a function fails to compile in `nopython` mode warnings will be emitted with explanation as to why compilation failed. For example with the `f()` function above (slightly edited for documentation purposes):

```
>>> f(1, 2)
3.0
>>> f(1, "2")
example.py:7: NumbaWarning:
Compilation is falling back to object mode WITH looplifting enabled because Function
→"f" failed type inference due to: Invalid use of Function(<class 'float'>) with␣
→argument(s) of type(s): (unicode_type)
* parameterized
In definition 0:
    TypeError: float() only support for numbers
    raised from <path>/numba/typing/builtins.py:880
In definition 1:
    TypeError: float() only support for numbers
    raised from <path>/numba/typing/builtins.py:880
This error is usually caused by passing an argument of a type that is unsupported by␣
→the named function.
[1] During: resolving callee type: Function(<class 'float'>)
[2] During: typing of call at example.py (9)


File "example.py", line 9:
def f(a, b):
    s = a + float(b)
    ^

<path>/numba/compiler.py:722: NumbaWarning: Function "f" was compiled in object mode␣
→without forceobj=True.

File "example.py", line 8:
@jit
def f(a, b):
```

```
^

3.0
```

1.17.6 Disabling JIT compilation

In order to debug code, it is possible to disable JIT compilation, which makes the `jit` decorator (and the `njit` decorator) act as if they perform no operation, and the invocation of decorated functions calls the original Python function instead of a compiled version. This can be toggled by setting the *NUMBA_DISABLE_JIT* enviroment variable to `1`.

When this mode is enabled, the `vectorize` and `guvectorize` decorators will still result in compilation of a ufunc, as there is no straightforward pure Python implementation of these functions.

1.17.7 Debugging JIT compiled code with GDB

Setting the `debug` keyword argument in the `jit` decorator (e.g. `@jit(debug=True)`) enables the emission of debug info in the jitted code. To debug, GDB version 7.0 or above is required. Currently, the following debug info is available:

- Function name will be shown in the backtrace. But, no type information.

- Source location (filename and line number) is available. For example, user can set break point by the absolute filename and line number; e.g. `break /path/to/myfile.py:6`.

- Local variables in the current function can be shown with `info locals`.

- Type of variable with `whatis myvar`.

- Value of variable with `print myvar` or `display myvar`.

  - Simple numeric types, i.e. int, float and double, are shown in their native representation. But, integers are assumed to be signed.

  - Other types are shown as sequence of bytes.

Known issues:

- Stepping depends heavily on optimization level.

  - At full optimization (equivalent to O3), most of the variables are optimized out.

  - With no optimization (e.g. `NUMBA_OPT=0`), source location jumps around when stepping through the code.

  - At O1 optimization (e.g. `NUMBA_OPT=1`), stepping is stable but some variables are optimized out.

- Memory consumption increases significantly with debug info enabled. The compiler emits extra information (DWARF) along with the instructions. The emitted object code can be 2x bigger with debug info.

Internal details:

- Since Python semantics allow variables to bind to value of different types, Numba internally creates multiple versions of the variable for each type. So for code like:

```
x = 1          # type int
x = 2.3        # type float
x = (1, 2, 3)  # type 3-tuple of int
```

Each assignments will store to a different variable name. In the debugger, the variables will be x, x$1 and x$2. (In the Numba IR, they are x, x.1 and x.2.)

- When debug is enabled, inlining of the function is disabled.

Example debug usage

The python source:

```python
from numba import njit

@njit(debug=True)
def foo(a):
    b = a + 1
    c = a * 2.34
    d = (a, b, c)
    print(a, b, c, d)

r= foo(123)
print(r)
```

In the terminal:

```
$ NUMBA_OPT=1 gdb -q python
Reading symbols from python...done.
(gdb) break /home/user/chk_debug.py:5
No source file named /home/user/chk_debug.py.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (/home/user/chk_debug.py:5) pending.
(gdb) run chk_debug.py
Starting program: /home/user/miniconda/bin/python chk_debug.py
...
Breakpoint 1, __main__::foo$241(long long) () at chk_debug.py:5
5           b = a + 1
(gdb) n
6           c = a * 2.34
(gdb) bt
#0  __main__::foo$241(long long) () at chk_debug.py:6
#1  0x00007ffff7fec47c in cpython::__main__::foo$241(long long) ()
#2  0x00007fffeb7976e2 in call_cfunc (locals=0x0, kws=0x0, args=0x7fffeb486198,
...
(gdb) info locals
a = 0
d = <error reading variable d (DWARF-2 expression error: `DW_OP_stack_value'␣
→operations must be used either alone or in conjunction with DW_OP_piece or DW_OP_
→bit_piece.)>
c = 0
b = 124
(gdb) whatis b
type = i64
(gdb) whatis d
type = {i64, i64, double}
(gdb) print b
$2 = 124
```

Globally override debug setting

It is possible to enable debug for the full application by setting environment variable NUMBA_DEBUGINFO=1. This sets the default value of the debug option in jit. Debug can be turned off on individual functions by setting debug=False.

Beware that enabling debug info significantly increases the memory consumption for each compiled function. For large application, this may cause out-of-memory error.

1.17.8 Using Numba's direct `gdb` bindings in `nopython` mode

Numba (version 0.42.0 and later) has some additional functions relating to gdb support for CPUs that make it easier to debug programs. All the gdb related functions described in the following work in the same manner irrespective of whether they are called from the standard CPython interpreter or code compiled in either *nopython mode* or *object mode*.

Note: This feature is experimental!

> **Warning:** This feature does unexpected things if used from Jupyter or alongside the pdb module. It's behaviour is harmless, just hard to predict!

Set up

Numba's gdb related functions make use of a gdb binary, the location and name of this binary can be configured via the *NUMBA_GDB_BINARY* environment variable if desired.

Note: Numba's gdb support requires the ability for gdb to attach to another process. On some systems (notably Ubuntu Linux) default security restrictions placed on ptrace prevent this from being possible. This restriction is enforced at the system level by the Linux security module *Yama*. Documentation for this module and the security implications of making changes to its behaviour can be found in the Linux Kernel documentation. The Ubuntu Linux security documentation discusses how to adjust the behaviour of *Yama* on with regards to ptrace_scope so as to permit the required behaviour.

Basic `gdb` support

> **Warning:** Calling numba.gdb() and/or numba.gdb_init() more than once in the same program is not advisable, unexpected things may happen. If multiple breakpoints are desired within a program, launch gdb once via numba.gdb() or numba.gdb_init() and then use numba.gdb_breakpoint() to register additional breakpoint locations.

The most simple function for adding gdb support is numba.gdb(), which, at the call location, will:

- launch gdb and attach it to the running process.
- create a breakpoint at the site of the numba.gdb() function call, the attached gdb will pause execution here awaiting user input.

---

use of this functionality is best motivated by example, continuing with the example used above:

```python
from numba import njit, gdb


@njit(debug=True)
def foo(a):
    b = a + 1
    gdb() # instruct Numba to attach gdb at this location and pause execution
    c = a * 2.34
    d = (a, b, c)
    print(a, b, c, d)


r= foo(123)
print(r)
```

In the terminal (`...` on a line by itself indicates output that is not presented for brevity):

```
$ NUMBA_OPT=0 python demo_gdb.py
Attaching to PID: 27157
GNU gdb (GDB) Red Hat Enterprise Linux 8.0.1-36.el7
...
Attaching to process 27157
...
Reading symbols from <elided for brevity> ...done.
0x00007f0380c31550 in __nanosleep_nocancel () at ../sysdeps/unix/syscall-template.S:81
81      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
Breakpoint 1 at 0x7f036ac388f0: file numba/_helperlib.c, line 1090.
Continuing.


Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) s
Single stepping until exit from function _ZN5numba7targets8gdb_hook8hook_gdb12
→$3clocals$3e8impl$242E5Tuple,
which has no line number information.
__main__::foo$241(long long) () at demo_gdb.py:7
7           c = a * 2.34
(gdb) l
2
3       @njit(debug=True)
4       def foo(a):
5           b = a + 1
6           gdb() # instruct Numba to attach gdb at this location and pause execution
7           c = a * 2.34
8           d = (a, b, c)
9           print(a, b, c, d)
10
11      r= foo(123)
(gdb) p a
$1 = 123
(gdb) p b
$2 = 124
(gdb) p c
$3 = 0
(gdb) n
8           d = (a, b, c)
(gdb) p c
$4 = 287.81999999999999
```

It can be seen in the above example that execution of the code is paused at the location of the gdb() function call at end of the numba_gdb_breakpoint function (this is the Numba internal symbol registered as breakpoint with gdb). Issuing a step at this point moves to the stack frame of the compiled Python source. From there, it can be seen that the variables a and b have been evaluated but c has not, as demonstrated by printing their values, this is precisely as expected given the location of the gdb() call. Issuing a next then evaluates line 7 and c is assigned a value as demonstrated by the final print.

Running with gdb enabled

The functionality provided by numba.gdb() (launch and attach gdb to the executing process and pause on a breakpoint) is also available as two separate functions:

- numba.gdb_init() this function injects code at the call site to launch and attach gdb to the executing process but does not pause execution.

- numba.gdb_breakpoint() this function injects code at the call site that will call the special numba_gdb_breakpoint function that is registered as a breakpoint in Numba's gdb support. This is demonstrated in the next section.

This functionality enables more complex debugging capabilities. Again, motivated by example, debugging a 'segfault' (memory access violation signalling SIGSEGV):

```python
1   from numba import njit, gdb_init
2   import numpy as np
3
4   @njit(debug=True)
5   def foo(a, index):
6       gdb_init() # instruct Numba to attach gdb at this location, but not to pause
    →execution
7       b = a + 1
8       c = a * 2.34
9       d = c[index] # access an address that is a) invalid b) out of the page
10      print(a, b, c, d)
11
12  bad_index = int(1e9) # this index is invalid
13  z = np.arange(10)
14  r = foo(z, bad_index)
15  print(r)
```

In the terminal (. . . on a line by itself indicates output that is not presented for brevity):

```
$ python demo_gdb_segfault.py
Attaching to PID: 5444
GNU gdb (GDB) Red Hat Enterprise Linux 8.0.1-36.el7
...
Attaching to process 5444
...
Reading symbols from <elided for brevity> ...done.
0x00007f8d8010a550 in __nanosleep_nocancel () at ../sysdeps/unix/syscall-template.S:81
81      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
Breakpoint 1 at 0x7f8d6a1118f0: file numba/_helperlib.c, line 1090.
Continuing.

0x00007fa7b810a41f in __main__::foo$241(Array<long long, 1, C, mutable, aligned>,
→long long) () at demo_gdb_segfault.py:9
9               d = c[index] # access an address that is a) invalid b) out of the page
(gdb) p index
```

(continues on next page)

```
$1 = 1000000000
(gdb) p c
$2 =
→"p\202\017\364\371U\000\000\000\000\000\000\000\000\000\000\n\000\000\000\000\000\000\000\000\b\000\000
→"
(gdb) whatis c
type = {i8*, i8*, i64, i64, double*, [1 x i64], [1 x i64]}
(gdb) x /32xb c
0x7ffd56195068: 0x70    0x82    0x0f    0xf4    0xf9    0x55    0x00    0x00
0x7ffd56195070: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffd56195078: 0x0a    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7ffd56195080: 0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

In the `gdb` output it can be noted that the `numba_gdb_breakpoint` function was registered as a breakpoint (its symbol is in `numba/_helperlib.c`), that a `SIGSEGV` signal was caught, and the line in which the access violation occurred is printed.

Continuing the example as a debugging session demonstration, first `index` can be printed, and it is evidently 1e9. Printing `c` gives a lot of bytes, so the type needs looking up. The type of `c` shows the layout for the array `c` based on its `DataModel` (look in the Numba source `numba.datamodel.models` for the layouts, the `ArrayModel` is presented below for ease).

```python
class ArrayModel(StructModel):
    def __init__(self, dmm, fe_type):
        ndim = fe_type.ndim
        members = [
            ('meminfo', types.MemInfoPointer(fe_type.dtype)),
            ('parent', types.pyobject),
            ('nitems', types.intp),
            ('itemsize', types.intp),
            ('data', types.CPointer(fe_type.dtype)),
            ('shape', types.UniTuple(types.intp, ndim)),
            ('strides', types.UniTuple(types.intp, ndim)),
        ]
```

The type inspected from `gdb` (`type = {i8*, i8*, i64, i64, double*, [1 x i64], [1 x i64]}`) corresponds directly to the members of the `ArrayModel`. Given the segfault came from an invalid access it would be informative to check the number of items in the array and compare that to the index requested.

Examining the memory of `c`, (`x /32xb c`), the first 16 bytes are the two `i8*` corresponding to the `meminfo` pointer and the `parent pyobject`. The next two groups of 8 bytes are `i64/intp` types corresponding to `nitems` and `itemsize` respectively. Evidently their values are `0x0a` and `0x08`, this makes sense as the input array `a` has 10 elements and is of type `int64` which is 8 bytes wide. It's therefore clear that the segfault comes from an invalid access of index `1000000000` in an array containing `10` items.

Adding breakpoints to code

The next example demonstrates using multiple breakpoints that are defined through the invocation of the `numba.gdb_breakpoint()` function:

```python
from numba import njit, gdb_init, gdb_breakpoint


@njit(debug=True)
def foo(a):
    gdb_init() # instruct Numba to attach gdb at this location
```

```
6        b = a + 1
7        gdb_breakpoint() # instruct gdb to break at this location
8        c = a * 2.34
9        d = (a, b, c)
10       gdb_breakpoint() # and to break again at this location
11       print(a, b, c, d)
12
13   r= foo(123)
14   print(r)
```

In the terminal (... on a line by itself indicates output that is not presented for brevity):

```
$ NUMBA_OPT=0 python demo_gdb_breakpoints.py
Attaching to PID: 20366
GNU gdb (GDB) Red Hat Enterprise Linux 8.0.1-36.el7
...
Attaching to process 20366
Reading symbols from <elided for brevity> ...done.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Reading symbols from /lib64/libc.so.6...Reading symbols from /usr/lib/debug/usr/lib64/
↪libc-2.17.so.debug...done.
0x00007f631db5e550 in __nanosleep_nocancel () at ../sysdeps/unix/syscall-template.S:81
81      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
Breakpoint 1 at 0x7f6307b658f0: file numba/_helperlib.c, line 1090.
Continuing.

Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) step
__main__::foo$241(long long) () at demo_gdb_breakpoints.py:8
8            c = a * 2.34
(gdb) l
3        @njit(debug=True)
4        def foo(a):
5            gdb_init() # instruct Numba to attach gdb at this location
6            b = a + 1
7            gdb_breakpoint() # instruct gdb to break at this location
8            c = a * 2.34
9            d = (a, b, c)
10           gdb_breakpoint() # and to break again at this location
11           print(a, b, c, d)
12
(gdb) p b
$1 = 124
(gdb) p c
$2 = 0
(gdb) continue
Continuing.

Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) step
11           print(a, b, c, d)
(gdb) p c
$3 = 287.81999999999999
```

From the `gdb` output it can be seen that execution paused at line 8 as a breakpoint was hit, and after a `continue` was issued, it broke again at line 11 where the next breakpoint was hit.

Debugging in parallel regions

The follow example is quite involved, it executes with `gdb` instrumentation from the outset as per the example above, but it also uses threads and makes use of the breakpoint functionality. Further, the last iteration of the parallel section calls the function `work`, which is actually just a binding to `glibc`'s `free(3)` in this case, but could equally be some involved function that is presenting a segfault for unknown reasons.

```python
from numba import njit, prange, gdb_init, gdb_breakpoint
import ctypes


def get_free():
    lib = ctypes.cdll.LoadLibrary('libc.so.6')
    free_binding = lib.free
    free_binding.argtypes = [ctypes.c_void_p,]
    free_binding.restype = None
    return free_binding


work = get_free()


@njit(debug=True, parallel=True)
def foo():
    gdb_init() # instruct Numba to attach gdb at this location, but not to pause
    ↪execution
    counter = 0
    n = 9
    for i in prange(n):
        if i > 3 and i < 8: # iterations 4, 5, 6, 7 will break here
            gdb_breakpoint()

        if i == 8: # last iteration segfaults
            work(0xBADADD)

        counter += 1
    return counter


r = foo()
print(r)
```

In the terminal (`...` on a line by itself indicates output that is not presented for brevity), note the setting of `NUMBA_NUM_THREADS` to 4 to ensure that there are 4 threads running in the parallel section:

```
$ NUMBA_NUM_THREADS=4 NUMBA_OPT=0 python demo_gdb_threads.py
Attaching to PID: 21462
...
Attaching to process 21462
[New LWP 21467]
[New LWP 21468]
[New LWP 21469]
[New LWP 21470]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
0x00007f59ec31756d in nanosleep () at ../sysdeps/unix/syscall-template.S:81
81      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
```

(continues on next page)

```
Breakpoint 1 at 0x7f59d631e8f0: file numba/_helperlib.c, line 1090.
Continuing.
[Switching to Thread 0x7f59d1fd1700 (LWP 21470)]

Thread 5 "python" hit Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) info threads
Id   Target Id         Frame
1    Thread 0x7f59eca2f740 (LWP 21462) "python" pthread_cond_wait@@GLIBC_2.3.2 ()
     at ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185
2    Thread 0x7f59d37d4700 (LWP 21467) "python" pthread_cond_wait@@GLIBC_2.3.2 ()
     at ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185
3    Thread 0x7f59d2fd3700 (LWP 21468) "python" pthread_cond_wait@@GLIBC_2.3.2 ()
     at ../nptl/sysdeps/unix/sysv/linux/x86_64/pthread_cond_wait.S:185
4    Thread 0x7f59d27d2700 (LWP 21469) "python" numba_gdb_breakpoint () at numba/_
→helperlib.c:1090
* 5    Thread 0x7f59d1fd1700 (LWP 21470) "python" numba_gdb_breakpoint () at numba/_
→helperlib.c:1090
(gdb) thread apply 2-5 info locals

Thread 2 (Thread 0x7f59d37d4700 (LWP 21467)):
No locals.

Thread 3 (Thread 0x7f59d2fd3700 (LWP 21468)):
No locals.

Thread 4 (Thread 0x7f59d27d2700 (LWP 21469)):
No locals.

Thread 5 (Thread 0x7f59d1fd1700 (LWP 21470)):
sched$35 = '\000' <repeats 55 times>
counter__arr = '\000' <repeats 16 times>,
→"\001\000\000\000\000\000\000\000\b\000\000\000\000\000\000\000\370B]\
→"hU\000\000\001", '\000' <repeats 14 times>
counter = 0
(gdb) continue
Continuing.
[Switching to Thread 0x7f59d27d2700 (LWP 21469)]

Thread 4 "python" hit Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) continue
Continuing.
[Switching to Thread 0x7f59d1fd1700 (LWP 21470)]

Thread 5 "python" hit Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) continue
Continuing.
[Switching to Thread 0x7f59d27d2700 (LWP 21469)]

Thread 4 "python" hit Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
(gdb) continue
Continuing.

Thread 5 "python" received signal SIGSEGV, Segmentation fault.
```

```
[Switching to Thread 0x7f59d1fd1700 (LWP 21470)]
__GI___libc_free (mem=0xbadadd) at malloc.c:2935
2935      if (chunk_is_mmapped(p))                        /* release mmapped memory. */
(gdb) bt
#0  __GI___libc_free (mem=0xbadadd) at malloc.c:2935
#1  0x00007f59d37ded84 in $3cdynamic$3e::__numba_parfor_gufunc__0x7ffff80a61ae3e31
↪$244(Array<unsigned long long, 1, C, mutable, aligned>, Array<long long, 1, C,␣
↪mutable, aligned>) () at <string>:24
#2  0x00007f59d17ce326 in __gufunc__._ZN13$3cdynamic$3e45__numba_parfor_gufunc__
↪0x7ffff80a61ae3e31
↪$244E5ArrayIyLi1E1C7mutable7alignedE5ArrayIxLi1E1C7mutable7alignedE ()
#3  0x00007f59d37d7320 in thread_worker ()
from <path>/numba/numba/npyufunc/workqueue.cpython-37m-x86_64-linux-gnu.so
#4  0x00007f59ec626e25 in start_thread (arg=0x7f59d1fd1700) at pthread_create.c:308
#5  0x00007f59ec350bad in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:113
```

In the output it can be seen that there are 4 threads launched and that they all break at the breakpoint, further that
`Thread 5` receives a signal `SIGSEGV` and that back tracing shows that it came from `__GI___libc_free` with
the invalid address in `mem`, as expected.

Using the `gdb` command language

Both the `numba.gdb()` and `numba.gdb_init()` functions accept unlimited string arguments which will be
passed directly to `gdb` as command line arguments when it initializes, this makes it easy to set breakpoints on other
functions and perform repeated debugging tasks without having to manually type them every time. For example,
this code runs with `gdb` attached and sets a breakpoint on `_dgesdd` (say for example the arguments passed to the
LAPACK's double precision divide and conqueror SVD function need debugging).

```python
from numba import njit, gdb
import numpy as np

@njit(debug=True)
def foo(a):
    # instruct Numba to attach gdb at this location and on launch, switch
    # breakpoint pending on , and then set a breakpoint on the function
    # _dgesdd, continue execution, and once the breakpoint is hit, backtrace
    gdb('-ex', 'set breakpoint pending on',
        '-ex', 'b dgesdd_',
        '-ex','c',
        '-ex','bt')
    b = a + 10
    u, s, vh = np.linalg.svd(b)
    return s # just return singular values

z = np.arange(70.).reshape(10, 7)
r = foo(z)
print(r)
```

In the terminal (`...` on a line by itself indicates output that is not presented for brevity), note that no interaction is
required to break and backtrace:

```
$ NUMBA_OPT=0 python demo_gdb_args.py
Attaching to PID: 22300
GNU gdb (GDB) Red Hat Enterprise Linux 8.0.1-36.el7
...
```

```
Attaching to process 22300
Reading symbols from <py_env>/bin/python3.7...done.
0x00007f652305a550 in __nanosleep_nocancel () at ../sysdeps/unix/syscall-template.S:81
81      T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
Breakpoint 1 at 0x7f650d0618f0: file numba/_helperlib.c, line 1090.
Continuing.

Breakpoint 1, numba_gdb_breakpoint () at numba/_helperlib.c:1090
1090    }
Breakpoint 2 at 0x7f65102322e0 (2 locations)
Continuing.

Breakpoint 2, 0x00007f65182be5f0 in mkl_lapack.dgesdd_ ()
from <py_env>/lib/python3.7/site-packages/numpy/core/../../../../libmkl_rt.so
#0  0x00007f65182be5f0 in mkl_lapack.dgesdd_ ()
from <py_env>/lib/python3.7/site-packages/numpy/core/../../../../libmkl_rt.so
#1  0x00007f650d065b71 in numba_raw_rgesdd (kind=kind@entry=100 'd', jobz=<optimized␣
→out>, jobz@entry=65 'A', m=m@entry=10,
    n=n@entry=7, a=a@entry=0x561c6fbb20c0, lda=lda@entry=10, s=0x561c6facf3a0,␣
→u=0x561c6fb680e0, ldu=10, vt=0x561c6fd375c0,
    ldvt=7, work=0x7fff4c926c30, lwork=-1, iwork=0x7fff4c926c40, info=0x7fff4c926c20)␣
→at numba/_lapack.c:1277
#2  0x00007f650d06768f in numba_ez_rgesdd (ldvt=7, vt=0x561c6fd375c0, ldu=10,␣
→u=0x561c6fb680e0, s=0x561c6facf3a0, lda=10,
    a=0x561c6fbb20c0, n=7, m=10, jobz=65 'A', kind=<optimized out>) at numba/_lapack.
→c:1307
#3  numba_ez_gesdd (kind=<optimized out>, jobz=<optimized out>, m=10, n=7,␣
→a=0x561c6fbb20c0, lda=10, s=0x561c6facf3a0,
    u=0x561c6fb680e0, ldu=10, vt=0x561c6fd375c0, ldvt=7) at numba/_lapack.c:1477
#4  0x00007f650a3147a3 in numba::targets::linalg::svd_impl::$3clocals$3e::svd_impl
→$243(Array<double, 2, C, mutable, aligned>, omitted$28default$3d1$29) ()
#5  0x00007f650a1c0489 in __main__::foo$241(Array<double, 2, C, mutable, aligned>) ()␣
→at demo_gdb_args.py:15
#6  0x00007f650a1c2110 in cpython::__main__::foo$241(Array<double, 2, C, mutable,␣
→aligned>) ()
#7  0x00007f650cd096a4 in call_cfunc ()
from <path>/numba/numba/_dispatcher.cpython-37m-x86_64-linux-gnu.so
...
```

How does the `gdb` binding work?

For advanced users and debuggers of Numba applications it's important to know some of the internal implementation details of the outlined `gdb` bindings. The `numba.gdb()` and `numba.gdb_init()` functions work by injecting the following into the function's LLVM IR:

- At the call site of the function first inject a call to `getpid(3)` to get the PID of the executing process and store this for use later, then inject a `fork(3)` call:

  - In the parent:

    * Inject a call `sleep(3)` (hence the pause whilst `gdb` loads).

    * Inject a call to the `numba_gdb_breakpoint` function (only `numba.gdb()` does this).

  - In the child:

    * Inject a call to `execl(3)` with the arguments `numba.config.GDB_BINARY`, the `attach` command and the PID recorded earlier. Numba has a special `gdb` command file that contains instructions

to break on the symbol `numba_gdb_breakpoint` and then `finish`, this is to make sure that the program stops on the breakpoint but the frame it stops in is the compiled Python frame (or one `step` away from, depending on optimisation). This command file is also added to the arguments and finally and any user specified arguments are added.

At the call site of a `numba.gdb_breakpoint()` a call is injected to the special `numba_gdb_breakpoint` symbol, which is already registered and instrumented as a place to break and `finish` immediately.

As a result of this, a e.g. `numba.gdb()` call will cause a fork in the program, the parent will sleep whilst the child launches `gdb` and attaches it to the parent and tells the parent to continue. The launched `gdb` has the `numba_gdb_breakpoint` symbol registered as a breakpoint and when the parent continues and stops sleeping it will immediately call `numba_gdb_breakpoint` on which the child will break. Additional `numba.gdb_breakpoint()` calls create calls to the registered breakpoint hence the program will also break at these locations.

1.17.9 Debugging CUDA Python code

Using the simulator

CUDA Python code can be run in the Python interpreter using the CUDA Simulator, allowing it to be debugged with the Python debugger or with print statements. To enable the CUDA simulator, set the environment variable *NUMBA_ENABLE_CUDASIM* to 1. For more information on the CUDA Simulator, see *the CUDA Simulator documentation*.

Debug Info

By setting the `debug` argument to `cuda.jit` to `True` (`@cuda.jit(debug=True)`), Numba will emit source location in the compiled CUDA code. Unlike the CPU target, only filename and line information are available, but no variable type information is emitted. The information is sufficient to debug memory error with cuda-memcheck.

For example, given the following cuda python code:

```python
import numpy as np
from numba import cuda

@cuda.jit(debug=True)
def foo(arr):
    arr[cuda.threadIdx.x] = 1

arr = np.arange(30)
foo[1, 32](arr)   # more threads than array elements
```

We can use `cuda-memcheck` to find the memory error:

```
$ cuda-memcheck python chk_cuda_debug.py
========= CUDA-MEMCHECK
========= Invalid __global__ write of size 8
=========     at 0x00000148 in /home/user/chk_cuda_debug.py:6:cudapy::__main__::foo
→$241(Array<__int64, int=1, C, mutable, aligned>)
=========     by thread (31,0,0) in block (0,0,0)
=========     Address 0x500a600f8 is out of bounds
...
=========
========= Invalid __global__ write of size 8
=========     at 0x00000148 in /home/user/chk_cuda_debug.py:6:cudapy::__main__::foo
→$241(Array<__int64, int=1, C, mutable, aligned>)
```

(continues on next page)

```
=========      by thread (30,0,0) in block (0,0,0)
=========      Address 0x500a600f0 is out of bounds
...
```

1.18 Frequently Asked Questions

1.18.1 Programming

Can I pass a function as an argument to a jitted function?

As of Numba 0.39, you can, so long as the function argument has also been JIT-compiled:

```python
@jit(nopython=True)
def f(g, x):
    return g(x) + g(-x)

result = f(jitted_g_function, 1)
```

However, dispatching with arguments that are functions has extra overhead. If this matters for your application, you can also use a factory function to capture the function argument in a closure:

```python
def make_f(g):
    # Note: a new f() is created each time make_f() is called!
    @jit(nopython=True)
    def f(x):
        return g(x) + g(-x)
    return f

f = make_f(jitted_g_function)
result = f(1)
```

Improving the dispatch performance of functions in Numba is an ongoing task.

Numba doesn't seem to care when I modify a global variable

Numba considers global variables as compile-time constants. If you want your jitted function to update itself when you have modified a global variable's value, one solution is to recompile it using the *recompile()* method. This is a relatively slow operation, though, so you may instead decide to rearchitect your code and turn the global variable into a function argument.

Can I debug a jitted function?

Calling into `pdb` or other such high-level facilities is currently not supported from Numba-compiled code. However, you can temporarily disable compilation by setting the *NUMBA_DISABLE_JIT* environment variable.

How can I create a Fortran-ordered array?

Numba currently doesn't support the `order` argument to most Numpy functions such as *numpy.empty()* (because of limitations in the *type inference* algorithm). You can work around this issue by creating a C-ordered array and then transposing it. For example:

```
a = np.empty((3, 5), order='F')
b = np.zeros(some_shape, order='F')
```

can be rewritten as:

```
a = np.empty((5, 3)).T
b = np.zeros(some_shape[::-1]).T
```

How can I increase integer width?

By default, Numba will generally use machine integer width for integer variables. On a 32-bit machine, you may sometimes need the magnitude of 64-bit integers instead. You can simply initialize relevant variables as `np.int64` (for example `np.int64(0)` instead of `0`). It will propagate to all computations involving those variables.

How can I tell if `parallel=True` worked?

If the `parallel=True` transformations failed for a function decorated as such, a warning will be displayed. See also *Diagnostics* for information about parallel diagnostics.

1.18.2 Performance

Does Numba inline functions?

Numba gives enough information to LLVM so that functions short enough can be inlined. This only works in *nopython mode*.

Does Numba vectorize array computations (SIMD)?

Numba doesn't implement such optimizations by itself, but it lets LLVM apply them.

Why my loop is not vectorized?

Numba enables the loop-vectorize optimization in LLVM by default. While it is a powerful optimization, not all loops are applicable. Sometimes, loop-vectorization may fail due to subtle details like memory access pattern. To see additional diagnostic information from LLVM, add the following lines:

```
import llvmlite.binding as llvm
llvm.set_option('', '--debug-only=loop-vectorize')
```

This tells LLVM to print debug information from the **loop-vectorize** pass to stderr. Each function entry looks like:

```
LV: Checking a loop in "<low-level symbol name>" from <function name>
LV: Loop hints: force=? width=0 unroll=0
...
LV: Vectorization is possible but not beneficial.
LV: Interleaving is not beneficial.
```

Each function entry is separated by an empty line. The reason for rejecting the vectorization is usually at the end of the entry. In the example above, LLVM rejected the vectorization because doing so will not speedup the loop. In this case, it can be due to memory access pattern. For instance, the array being looped over may not be in contiguous layout.

When memory access pattern is non-trivial such that it cannot determine the access memory region, LLVM may reject with the following message:

```
LV: Can't vectorize due to memory conflicts
```

Another common reason is:

```
LV: Not vectorizing: loop did not meet vectorization requirements.
```

In this case, vectorization is rejected because the vectorized code may behave differently. This is a case to try turning on `fastmath=True` to allow fastmath instructions.

Does Numba automatically parallelize code?

It can, in some cases:

- Ufuncs and gufuncs with the `target="parallel"` option will run on multiple threads.

- The `parallel=True` option to `@jit` will attempt to optimize array operations and run them in parallel. It also adds support for `prange()` to explicitly parallelize a loop.

You can also manually run computations on multiple threads yourself and use the `nogil=True` option (see *releasing the GIL*). Numba can also target parallel execution on GPU architectures using its CUDA and HSA backends.

Can Numba speed up short-running functions?

Not significantly. New users sometimes expect to JIT-compile such functions:

```python
def f(x, y):
    return x + y
```

and get a significant speedup over the Python interpreter. But there isn't much Numba can improve here: most of the time is probably spent in CPython's function call mechanism, rather than the function itself. As a rule of thumb, if a function takes less than 10 µs to execute: leave it.

The exception is that you *should* JIT-compile that function if it is called from another jitted function.

There is a delay when JIT-compiling a complicated function, how can I improve it?

Try to pass `cache=True` to the `@jit` decorator. It will keep the compiled version on disk for later use.

A more radical alternative is *ahead-of-time compilation*.

1.18.3 GPU Programming

How do I work around the `CUDA intialized before forking` error?

On Linux, the `multiprocessing` module in the Python standard library defaults to using the `fork` method for creating new processes. Because of the way process forking duplicates state between the parent and child processes, CUDA will not work correctly in the child process if the CUDA runtime was initialized *prior* to the fork. Numba detects this and raises a `CudaDriverError` with the message `CUDA initialized before forking`.

One approach to avoid this error is to make all calls to `numba.cuda` functions inside the child processes or after the process pool is created. However, this is not always possible, as you might want to query the number of available GPUs before starting the process pool. In Python 3, you can change the process start method, as described in the

multiprocessing documentation. Switching from `fork` to `spawn` or `forkserver` will avoid the CUDA initalization issue, although the child processes will not inherit any global variables from their parent.

1.18.4 Integration with other utilities

Can I "freeze" an application which uses Numba?

If you're using PyInstaller or a similar utility to freeze an application, you may encounter issues with llvmlite. llvmlite needs a non-Python DLL for its working, but it won't be automatically detected by freezing utilities. You have to inform the freezing utility of the DLL's location: it will usually be named `llvmlite/binding/libllvmlite.so` or `llvmlite/binding/llvmlite.dll`, depending on your system.

I get errors when running a script twice under Spyder

When you run a script in a console under Spyder, Spyder first tries to reload existing modules. This doesn't work well with Numba, and can produce errors like `TypeError: No matching definition for argument type(s)`.

There is a fix in the Spyder preferences. Open the "Preferences" window, select "Console", then "Advanced Settings", click the "Set UMR excluded modules" button, and add `numba` inside the text box that pops up.

To see the setting take effect, be sure to restart the IPython console or kernel.

Why does Numba complain about the current locale?

If you get an error message such as the following:

```
RuntimeError: Failed at nopython (nopython mode backend)
LLVM will produce incorrect floating-point code in the current locale
```

it means you have hit a LLVM bug which causes incorrect handling of floating-point constants. This is known to happen with certain third-party libraries such as the Qt backend to matplotlib.

To work around the bug, you need to force back the locale to its default value, for example:

```python
import locale
locale.setlocale(locale.LC_NUMERIC, 'C')
```

1.18.5 Miscellaneous

Where does the project name "Numba" come from?

"Numba" is a combination of "NumPy" and "Mamba". Mambas are some of the fastest snakes in the world, and Numba makes your Python code fast.

How do I reference/cite/acknowledge Numba in other work?

For academic use, the best option is to cite our ACM Proceedings: Numba: a LLVM-based Python JIT compiler. You can also find the sources on github, including a pre-print pdf, in case you don't have access to the ACM site but would like to read the paper.

Other related papers

A paper describing ParallelAccelerator technology, that is activated when the `parallel=True` jit option is used, can be found here.

1.19 Examples

1.19.1 Mandelbrot

Listing 1: from `test_mandelbrot` of `numba/tests/doc_examples/test_examples.py`

```python
from timeit import default_timer as timer
try:
    from matplotlib.pylab import imshow, show
    have_mpl = True
except ImportError:
    have_mpl = False
import numpy as np
from numba import jit


@jit(nopython=True)
def mandel(x, y, max_iters):
    """
    Given the real and imaginary parts of a complex number,
    determine if it is a candidate for membership in the Mandelbrot
    set given a fixed number of iterations.
    """
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z * z + c
        if (z.real * z.real + z.imag * z.imag) >= 4:
            return i

    return 255


@jit(nopython=True)
def create_fractal(min_x, max_x, min_y, max_y, image, iters):
    height = image.shape[0]
    width = image.shape[1]

    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    for x in range(width):
        real = min_x + x * pixel_size_x
        for y in range(height):
            imag = min_y + y * pixel_size_y
            color = mandel(real, imag, iters)
            image[y, x] = color

    return image

image = np.zeros((500 * 2, 750 * 2), dtype=np.uint8)
```

(continues on next page)

```
44  s = timer()
45  create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
46  e = timer()
47  print(e - s)
48  if have_mpl:
49      imshow(image)
50      show()
```

1.19.2 Moving average

Listing 2: from `test_moving_average` of `numba/tests/doc_examples/test_examples.py`

```python
1  import numpy as np
2
3  from numba import guvectorize
4
5  @guvectorize(['void(float64[:], intp[:], float64[:])'],
6              '(n),()->(n)')
7  def move_mean(a, window_arr, out):
8      window_width = window_arr[0]
9      asum = 0.0
10     count = 0
11     for i in range(window_width):
12         asum += a[i]
13         count += 1
14         out[i] = asum / count
15     for i in range(window_width, len(a)):
16         asum += a[i] - a[i - window_width]
17         out[i] = asum / count
18
19  arr = np.arange(20, dtype=np.float64).reshape(2, 10)
20  print(arr)
21  print(move_mean(arr, 3))
```

1.19.3 Multi-threading

The code below showcases the potential performance improvement when using the *nogil* feature. For example, on a 4-core machine, the following results were printed:

```
numpy (1 thread)        145 ms
numba (1 thread)        128 ms
numba (4 threads)        35 ms
```

Note: If preferred it's possible to use the standard concurrent.futures module rather than spawn threads and dispatch tasks by hand.

Listing 3: from `test_no_gil` of `numba/tests/doc_examples/test_examples.py`

```python
import math
import threading
from timeit import repeat

import numpy as np
from numba import jit

nthreads = 4
size = 10**6

def func_np(a, b):
    """
    Control function using Numpy.
    """
    return np.exp(2.1 * a + 3.2 * b)

@jit('void(double[:], double[:], double[:])', nopython=True,
     nogil=True)
def inner_func_nb(result, a, b):
    """
    Function under test.
    """
    for i in range(len(result)):
        result[i] = math.exp(2.1 * a[i] + 3.2 * b[i])

def timefunc(correct, s, func, *args, **kwargs):
    """
    Benchmark *func* and print out its runtime.
    """
    print(s.ljust(20), end=" ")
    # Make sure the function is compiled before the benchmark is
    # started
    res = func(*args, **kwargs)
    if correct is not None:
        assert np.allclose(res, correct), (res, correct)
    # time it
    print('{:>5.0f} ms'.format(min(repeat(
        lambda: func(*args, **kwargs), number=5, repeat=2)) * 1000))
    return res

def make_singlethread(inner_func):
    """
    Run the given function inside a single thread.
    """
    def func(*args):
        length = len(args[0])
        result = np.empty(length, dtype=np.float64)
        inner_func(result, *args)
        return result
    return func

def make_multithread(inner_func, numthreads):
    """
    Run the given function inside *numthreads* threads, splitting
```

(continues on next page)

```
55        its arguments into equal-sized chunks.
56        """
57     def func_mt(*args):
58         length = len(args[0])
59         result = np.empty(length, dtype=np.float64)
60         args = (result,) + args
61         chunklen = (length + numthreads - 1) // numthreads
62         # Create argument tuples for each input chunk
63         chunks = [[arg[i * chunklen:(i + 1) * chunklen] for arg in
64                    args] for i in range(numthreads)]
65         # Spawn one thread per chunk
66         threads = [threading.Thread(target=inner_func, args=chunk)
67                    for chunk in chunks]
68         for thread in threads:
69             thread.start()
70         for thread in threads:
71             thread.join()
72         return result
73     return func_mt

75 func_nb = make_singlethread(inner_func_nb)
76 func_nb_mt = make_multithread(inner_func_nb, nthreads)

78 a = np.random.rand(size)
79 b = np.random.rand(size)

81 correct = timefunc(None, "numpy (1 thread)", func_np, a, b)
82 timefunc(correct, "numba (1 thread)", func_nb, a, b)
83 timefunc(correct, "numba (%d threads)" % nthreads, func_nb_mt, a, b)
```

1.20 Talks and Tutorials

Note: This is a selection of talks and tutorials that have been given by members of the Numba team as well as Numba users. If you know of a Numba-related talk that should be included on this list, please open an issue.

1.20.1 Talks on Numba

- AnacondaCON 2018 - Accelerating Scientific Workloads with Numba - Siu Kwan Lam (Video)
- DIANA-HEP Meeting, 23 April 2018 - Overview of Numba - Stan Seibert

1.20.2 Talks on Applications of Numba

- GPU Technology Conference 2016 - Accelerating a Spectral Algorithm for Plasma Physics with Python/Numba on GPU - Manuel Kirchen & Rémi Lehe (Slides)
- DIANA-HEP Meeting, 23 April 2018 - Use of Numba in XENONnT - Chris Tunnell
- DIANA-HEP Meeting, 23 April 2018 - Extending Numba for HEP data types - Jim Pivarski

- STAC Summit, Nov 1 2017 - Scaling High-Performance Python with Minimal Effort - Ehsan Totoni (Video, Slides)
- SciPy 2018 - UMAP: Uniform Manifold Approximation and Projection for Dimensional Reduction - Leland McInnes (Video, Github)
- PyData Berlin 2018 - Extending Pandas using Apache Arrow and Numba - Uwe L. Korn (Video, Blog)
- FOSDEM 2019 - Extending Numba - Joris Geessels (Video, Slides & Examples)
- PyCon India 2019 - Real World Numba: Taking the Path of Least Resistance - Ankit Mahato (Video)
- SciPy 2019 - How to Accelerate an Existing Codebase with Numba - Siu Kwan Lam & Stanley Seibert (Video)
- SciPy 2019 - Real World Numba: Creating a Skeleton Analysis Library - Juan Nunez-Iglesias (Video)
- SciPy 2019 - Fast Gradient Boosting Decision Trees with PyGBM and Numba - Nicholas Hug (Video)

1.20.3 Tutorials

- SciPy 2017 - Numba: Tell those C++ Bullies to Get Lost - Gil Forsyth & Lorena Barba (Video, Notebooks)
- GPU Technology Conference 2018 - GPU Computing in Python with Numba - Stan Seibert (Notebooks)
- PyData Amsterdam 2019 - Create CUDA kernels from Python using Numba and CuPy - Valentin Haenel (Video)

Reference Manual

2.1 Types and signatures

2.1.1 Rationale

As an optimizing compiler, Numba needs to decide on the type of each variable to generate efficient machine code. Python's standard types are not precise enough for that, so we had to develop our own fine-grained type system.

You will encounter Numba types mainly when trying to inspect the results of Numba's type inference, for *debugging* or *educational* purposes. However, you need to use types explicitly if compiling code *ahead-of-time*.

2.1.2 Signatures

A signature specifies the type of a function. Exactly which kind of signature is allowed depends on the context (*AOT* or *JIT* compilation), but signatures always involve some representation of Numba types to specify the concrete types for the function's arguments and, if required, the function's return type.

An example function signature would be the string `"f8(i4, i4)"` (or the equivalent `"float64(int32, int32)"`) which specifies a function taking two 32-bit integers and returning a double-precision float.

2.1.3 Basic types

The most basic types can be expressed through simple expressions. The symbols below refer to attributes of the main `numba` module (so if you read "boolean", it means that symbol can be accessed as `numba.boolean`). Many types are available both as a canonical name and a shorthand alias, following Numpy's conventions.

Numbers

The following table contains the elementary numeric types currently defined by Numba and their aliases.

Type name(s)	Shorthand	Comments
boolean	b1	represented as a byte
uint8, byte	u1	8-bit unsigned byte
uint16	u2	16-bit unsigned integer
uint32	u4	32-bit unsigned integer
uint64	u8	64-bit unsigned integer
int8, char	i1	8-bit signed byte
int16	i2	16-bit signed integer
int32	i4	32-bit signed integer
int64	i8	64-bit signed integer
intc	–	C int-sized integer
uintc	–	C int-sized unsigned integer
intp	–	pointer-sized integer
uintp	–	pointer-sized unsigned integer
float32	f4	single-precision floating-point number
float64, double	f8	double-precision floating-point number
complex64	c8	single-precision complex number
complex128	c16	double-precision complex number

Arrays

The easy way to declare array types is to subscript an elementary type according to the number of dimensions. For example a 1-dimension single-precision array:

```
>>> numba.float32[:]
array(float32, 1d, A)
```

or a 3-dimension array of the same underlying type:

```
>>> numba.float32[:, :, :]
array(float32, 3d, A)
```

This syntax defines array types with no particular layout (producing code that accepts both non-contiguous and contiguous arrays), but you can specify a particular contiguity by using the `::1` index either at the beginning or the end of the index specification:

```
>>> numba.float32[::1]
array(float32, 1d, C)
>>> numba.float32[:, :, ::1]
array(float32, 3d, C)
>>> numba.float32[::1, :, :]
array(float32, 3d, F)
```

Functions

> **Warning:** The feature of considering functions as first-class type objects is under development.

Functions are often considered as certain transformations of input arguments to output values. Within Numba *JIT* compiled functions, the functions can also be considered as objects, that is, functions can be passed around as arguments or return values, or used as items in sequences, in addition to being callable.

First-class function support is enabled for all Numba *JIT* compiled functions and Numba `cfunc` compiled functions except when:

- using a non-CPU compiler,

- the compiled function is a Python generator,

- the compiled function has Omitted arguments,

- or the compiled function returns Optional value.

To disable first-class function support, use `no_cfunc_wrapper=True` decorator option.

For instance, consider an example where the Numba *JIT* compiled function applies user-specified functions as a composition to an input argument:

```
>>> @numba.njit
... def composition(funcs, x):
...     r = x
...     for f in funcs[::-1]:
...         r = f(r)
...     return r
...
>>> @numba.cfunc("double(double)")
... def a(x):
...     return x + 1.0
...
>>> @numba.njit
... def b(x):
...     return x * x
...
>>> composition((a, b), 0.5), 0.5 ** 2 + 1
(1.25, 1.25)
>>> composition((b, a, b, b, a), 0.5), b(a(b(b(a(0.5)))))
(36.75390625, 36.75390625)
```

Here, `cfunc` compiled functions `a` and `b` are considered as first-class function objects because these are passed in to the Numba *JIT* compiled function `composition` as arguments, that is, the `composition` is *JIT* compiled independently from its argument function objects (that are collected in the input argument `funcs`).

Currently, first-class function objects can be Numba `cfunc` compiled functions, *JIT* compiled functions, and objects that implement the Wrapper Address Protocol (WAP, see below) with the following restrictions:

Context	JIT compiled	cfunc compiled	WAP objects
Can be used as arguments	yes	yes	yes
Can be called	yes	yes	yes
Can be used as items	yes*	yes	yes
Can be returned	yes	yes	yes
Namespace scoping	yes	yes	yes
Automatic overload	yes	no	no

* at least one of the items in a sequence of first-class function objects must have a precise type.

Wrapper Address Protocol - WAP

Wrapper Address Protocol provides an API for making any Python object a first-class function for Numba *JIT* compiled functions. This assumes that the Python object represents a compiled function that can be called via its memory

address (function pointer value) from Numba *JIT* compiled functions. The so-called WAP objects must define the following two methods:

__wrapper_address__(*self*) → int
> Return the memory address of a first-class function. This method is used when a Numba *JIT* compiled function tries to call the given WAP instance.

signature(*self*) → numba.typing.Signature
> Return the signature of the given first-class function. This method is used when passing in the given WAP instance to a Numba *JIT* compiled function.

In addition, the WAP object may implement the __call__ method. This is necessary when calling WAP objects from Numba *JIT* compiled functions in *object mode*.

As an example, let us call the standard math library function cos within a Numba *JIT* compiled function. The memory address of cos can be established after loading the math library and using the ctypes package:

```
>>> import numba, ctypes, ctypes.util, math
>>> libm = ctypes.cdll.LoadLibrary(ctypes.util.find_library('m'))
>>> class LibMCos(numba.types.WrapperAddressProtocol):
...     def __wrapper_address__(self):
...         return ctypes.cast(libm.cos, ctypes.c_voidp).value
...     def signature(self):
...         return numba.float64(numba.float64)
...
>>> @numba.njit
... def foo(f, x):
...     return f(x)
...
>>> foo(LibMCos(), 0.0)
1.0
>>> foo(LibMCos(), 0.5), math.cos(0.5)
(0.8775825618903728, 0.8775825618903728)
```

Miscellaneous Types

There are some non-numerical types that do not fit into the other categories.

Type name(s)	Comments
pyobject	generic Python object
voidptr	raw pointer, no operations can be performed on it

2.1.4 Advanced types

For more advanced declarations, you have to explicitly call helper functions or classes provided by Numba.

> **Warning:** The APIs documented here are not guaranteed to be stable. Unless necessary, it is recommended to let Numba infer argument types by using the *signature-less variant of @jit*.

Inference

numba.**typeof**(*value*)
> Create a Numba type accurately describing the given Python *value*. ValueError is raised if the value isn't

supported in *nopython mode*.

```
>>> numba.typeof(np.empty(3))
array(float64, 1d, C)
>>> numba.typeof((1, 2.0))
(int64, float64)
>>> numba.typeof([0])
reflected list(int64)
```

Numpy scalars

Instead of using `typeof()`, non-trivial scalars such as structured types can also be constructed programmatically.

numba.**from_dtype**(*dtype*)

Create a Numba type corresponding to the given Numpy *dtype*:

```
>>> struct_dtype = np.dtype([('row', np.float64), ('col', np.float64)])
>>> ty = numba.from_dtype(struct_dtype)
>>> ty
Record([('row', '<f8'), ('col', '<f8')])
>>> ty[:, :]
unaligned array(Record([('row', '<f8'), ('col', '<f8')]), 2d, A)
```

class numba.types.**NPDatetime**(*unit*)

Create a Numba type for Numpy datetimes of the given *unit*. *unit* should be a string amongst the codes recognized by Numpy (e.g. Y, M, D, etc.).

class numba.types.**NPTimedelta**(*unit*)

Create a Numba type for Numpy timedeltas of the given *unit*. *unit* should be a string amongst the codes recognized by Numpy (e.g. Y, M, D, etc.).

See also:

Numpy datetime units.

Arrays

class numba.types.**Array**(*dtype*, *ndim*, *layout*)

Create an array type. *dtype* should be a Numba type. *ndim* is the number of dimensions of the array (a positive integer). *layout* is a string giving the layout of the array: A means any layout, C means C-contiguous and F means Fortran-contiguous.

Optional types

class numba.**optional**(*typ*)

Create an optional type based on the underlying Numba type *typ*. The optional type will allow any value of either *typ* or None.

```
>>> @jit((optional(intp),))
... def f(x):
...     return x is not None
...
>>> f(0)
True
```

(continues on next page)

```
>>> f(None)
False
```

2.2 Just-in-Time compilation

2.2.1 JIT functions

@numba.**jit**(*signature=None*, *nopython=False*, *nogil=False*, *cache=False*, *forceobj=False*, *parallel=False*,
        error_model='python', *fastmath=False*, *locals={}*, *boundscheck=False*)
 Compile the decorated function on-the-fly to produce efficient machine code. All parameters are optional.

If present, the *signature* is either a single signature or a list of signatures representing the expected *Types and signatures* of function arguments and return values. Each signature can be given in several forms:

- A tuple of *Types and signatures* arguments (for example (numba.int32, numba.double)) representing the types of the function's arguments; Numba will then infer an appropriate return type from the arguments.

- A call signature using *Types and signatures*, specifying both return type and argument types. This can be given in intuitive form (for example numba.void(numba.int32, numba.double)).

- A string representation of one of the above, for example "void(int32, double)". All type names used in the string are assumed to be defined in the numba.types module.

nopython and *nogil* are boolean flags. *locals* is a mapping of local variable names to *Types and signatures*.

This decorator has several modes of operation:

- If one or more signatures are given in *signature*, a specialization is compiled for each of them. Calling the decorated function will then try to choose the best matching signature, and raise a TypeError if no appropriate conversion is available for the function arguments. If converting succeeds, the compiled machine code is executed with the converted arguments and the return value is converted back according to the signature.

- If no *signature* is given, the decorated function implements lazy compilation. Each call to the decorated function will try to re-use an existing specialization if it exists (for example, a call with two integer arguments may re-use a specialization for argument types (numba.int64, numba.int64)). If no suitable specialization exists, a new specialization is compiled on-the-fly, stored for later use, and executed with the converted arguments.

If true, *nopython* forces the function to be compiled in *nopython mode*. If not possible, compilation will raise an error.

If true, *forceobj* forces the function to be compiled in *object mode*. Since object mode is slower than nopython mode, this is mostly useful for testing purposes.

If true, *nogil* tries to release the global interpreter lock inside the compiled function. The GIL will only be released if Numba can compile the function in *nopython mode*, otherwise a compilation warning will be printed.

If true, *cache* enables a file-based cache to shorten compilation times when the function was already compiled in a previous invocation. The cache is maintained in the __pycache__ subdirectory of the directory containing the source file; if the current user is not allowed to write to it, though, it falls back to a platform-specific user-wide cache directory (such as $HOME/.cache/numba on Unix platforms).

If true, *parallel* enables the automatic parallelization of a number of common Numpy constructs as well as the fusion of adjacent parallel operations to maximize cache locality.

The *error_model* option controls the divide-by-zero behavior. Setting it to 'python' causes divide-by-zero to raise exception like CPython. Setting it to 'numpy' causes divide-by-zero to set the result to *+/-inf* or *nan*.

Not all functions can be cached, since some functionality cannot be always persisted to disk. When a function cannot be cached, a warning is emitted.

If true, *fastmath* enables the use of otherwise unsafe floating point transforms as described in the LLVM documentation. Further, if *Intel SVML* is installed faster but less accurate versions of some math intrinsics are used (answers to within `4 ULP`).

If True, `boundscheck` enables bounds checking for array indices. Out of bounds accesses will raise Index-Error. The default is to not do bounds checking. If bounds checking is disabled, out of bounds accesses can produce garbage results or segfaults. However, enabling bounds checking will slow down typical functions, so it is recommended to only use this flag for debugging. You can also set the *NUMBA_BOUNDSCHECK* environment variable to 0 or 1 to globally override this flag.

The *locals* dictionary may be used to force the *Types and signatures* of particular local variables, for example if you want to force the use of single precision floats at some point. In general, we recommend you let Numba's compiler infer the types of local variables by itself.

Here is an example with two signatures:

```python
@jit(["int32(int32)", "float32(float32)"], nopython=True)
def f(x): ...
```

Not putting any parentheses after the decorator is equivalent to calling the decorator without any arguments, i.e.:

```python
@jit
def f(x): ...
```

is equivalent to:

```python
@jit()
def f(x): ...
```

The decorator returns a `Dispatcher` object.

Note: If no *signature* is given, compilation errors will be raised when the actual compilation occurs, i.e. when the function is first called with some given argument types.

Note: Compilation can be influenced by some dedicated *Environment variables*.

2.2.2 Generated JIT functions

@numba.**generated_jit**(*nopython=False*, *nogil=False*, *cache=False*, *forceobj=False*, *locals={}*)
Like the `jit()` decorator, but calls the decorated function at compile-time, passing the *types* of the function's arguments. The decorated function must return a callable which will be compiled as the function's implementation for those types, allowing flexible kinds of specialization.

The `generated_jit()` decorator returns a `Dispatcher` object.

2.2.3 Dispatcher objects

class Dispatcher

The class of objects created by calling *jit()* or *generated_jit()*. You shouldn't try to create such an object in any other way. Calling a Dispatcher object calls the compiled specialization for the arguments with which it is called, letting it act as an accelerated replacement for the Python function which was compiled.

In addition, Dispatcher objects have the following methods and attributes:

py_func

The pure Python function which was compiled.

inspect_types(*file=None*, *pretty=False*)

Print out a listing of the function source code annotated line-by-line with the corresponding Numba IR, and the inferred types of the various variables. If *file* is specified, printing is done to that file object, otherwise to sys.stdout. If *pretty* is set to True then colored ANSI will be produced in a terminal and HTML in a notebook.

See also:

Numba architecture

inspect_llvm(*signature=None*)

Return a dictionary keying compiled function signatures to the human readable LLVM IR generated for the function. If the signature keyword is specified a string corresponding to that individual signature is returned.

inspect_asm(*signature=None*)

Return a dictionary keying compiled function signatures to the human-readable native assembly code for the function. If the signature keyword is specified a string corresponding to that individual signature is returned.

inspect_cfg(*signature=None*, *show_wrapped*)

Return a dictionary keying compiled function signatures to the control-flow graph objects for the function. If the signature keyword is specified a string corresponding to that individual signature is returned.

The control-flow graph objects can be stringified (`str` or `repr`) to get the textual representation of the graph in DOT format. Or, use its `.display(filename=None, view=False)` method to plot the graph. The *filename* option can be set to a specific path for the rendered output to write to. If *view* option is True, the plot is opened by the system default application for the image format (PDF). In IPython notebook, the returned object can be plot inlined.

Usage:

```python
@jit
def foo():
    ...

# opens the CFG in system default application
foo.inspect_cfg(foo.signatures[0]).display(view=True)
```

inspect_disasm_cfg(*signature=None*)

Return a dictionary keying compiled function signatures to the control-flow graph of the disassembly of the underlying compiled `ELF` object. If the signature keyword is specified a control-flow graph corresponding to that individual signature is returned. This function is execution environment aware and will produce SVG output in Jupyter notebooks and ASCII in terminals.

Example:

```
@njit
def foo(x):
    if x < 3:
        return x + 1
    return x + 2

foo(10)

print(foo.inspect_disasm_cfg(signature=foo.signatures[0]))
```

Gives:

```
[0x08000040]>  # method.__main__.foo_241_long_long (int64_t arg1, int64_t␣
↪arg3);
 ┌─────────────────────────────────────────────────────────────────────────┐
 │  0x8000040                                                                │
 │ ; arg3 ; [02] -r-x section size 279 named .text                          │
 │   ;-- section..text:                                                      │
 │   ;-- .text:                                                              │
 │   ;-- __main__::foo$241(long long):                                       │
 │   ;-- rip:                                                                │
 │ 25: method.__main__.foo_241_long_long (int64_t arg1, int64_t arg3);       │
 │ ; arg int64_t arg1 @ rdi                                                  │
 │ ; arg int64_t arg3 @ rdx                                                  │
 │ ; 2                                                                       │
 │ cmp rdx, 2                                                                │
 │ jg 0x800004f                                                             │
 └─────────────────────────────────────────────────────────────────────────┘
          f t
          │ │
      ┌───┘ └──────────────────────────────────┐
      │                                          │
      │                                          │
 ┌────────────────────────┐       ┌────────────────────────┐
 │  0x8000046             │       │  0x800004f             │
 │ ; arg3                 │       │ ; arg3                 │
 │ inc rdx                │       │ add rdx, 2             │
 │ ; arg3                 │       │ ; arg3                 │
 │ mov qword [rdi], rdx   │       │ mov qword [rdi], rdx   │
 │ xor eax, eax           │       │ xor eax, eax           │
 │ ret                    │       │ ret                    │
 └────────────────────────┘       └────────────────────────┘
```
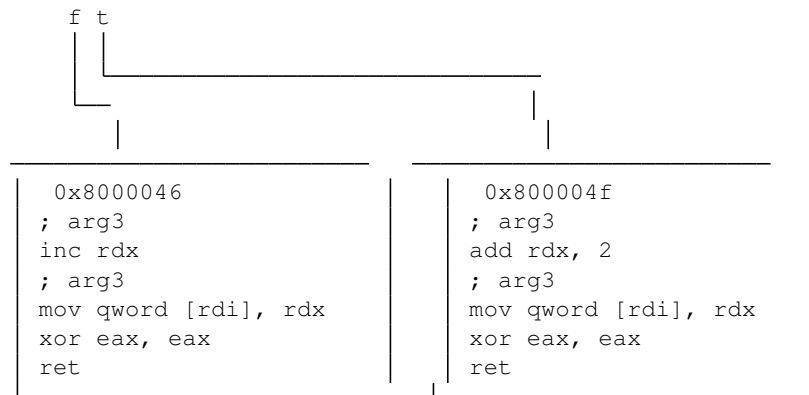
recompile()

Recompile all existing signatures. This can be useful for example if a global or closure variable was frozen by your function and its value in Python has changed. Since compiling isn't cheap, this is mainly for testing and interactive use.

parallel_diagnostics(*signature=None*, *level=1*)

Print parallel diagnostic information for the given signature. If no signature is present it is printed for all known signatures. `level` is used to adjust the verbosity, `level=1` (default) is minimum verbosity, levels 2, 3, and 4 provide increasing levels of verbosity.

get_metadata(*signature=None*)

Obtain the compilation metadata for a given signature. This is useful for developers of Numba and Numba extensions.

2.2.4 Vectorized functions (ufuncs and DUFuncs)

@numba.**vectorize**(*, *signatures=[]*, *identity=None*, *nopython=True*, *target='cpu'*, *forceobj=False*,
             cache=False, *locals={}*)

Compile the decorated function and wrap it either as a Numpy ufunc or a Numba *DUFunc*. The optional *nopython*, *forceobj* and *locals* arguments have the same meaning as in `numba.jit()`.

signatures is an optional list of signatures expressed in the same form as in the `numba.jit()` *signature* argument. If *signatures* is non-empty, then the decorator will compile the user Python function into a Numpy ufunc. If no *signatures* are given, then the decorator will wrap the user Python function in a *DUFunc* instance, which will compile the user function at call time whenever Numpy can not find a matching loop for the input arguments. *signatures* is required if *target* is `"parallel"`.

identity is the identity (or unit) value of the function being implemented. Possible values are 0, 1, None, and the string `"reorderable"`. The default is None. Both None and `"reorderable"` mean the function has no identity value; `"reorderable"` additionally specifies that reductions along multiple axes can be reordered.

If there are several *signatures*, they must be ordered from the more specific to the least specific. Otherwise, Numpy's type-based dispatching may not work as expected. For example, the following is wrong:

```python
@vectorize(["float64(float64)", "float32(float32)"])
def f(x): ...
```

as running it over a single-precision array will choose the `float64` version of the compiled function, leading to much less efficient execution. The correct invocation is:

```python
@vectorize(["float32(float32)", "float64(float64)"])
def f(x): ...
```

target is a string for backend target; Available values are "cpu", "parallel", and "cuda". To use a multithreaded version, change the target to "parallel" (which requires signatures to be specified):

```python
@vectorize(["float64(float64)", "float32(float32)"], target='parallel')
def f(x): ...
```

For the CUDA target, use "cuda":

```python
@vectorize(["float64(float64)", "float32(float32)"], target='cuda')
def f(x): ...
```

The compiled function can be cached to reduce future compilation time. It is enabled by setting *cache* to True. Only the "cpu" and "parallel" targets support caching.

@numba.**guvectorize**(*signatures*, *layout*, *, *identity=None*, *nopython=True*, *target='cpu'*,
                forceobj=False, *cache=False*, *locals={}*)

Generalized version of `numba.vectorize()`. While `numba.vectorize()` will produce a simple ufunc whose core functionality (the function you are decorating) operates on scalar operands and returns a scalar value, `numba.guvectorize()` allows you to create a Numpy ufunc whose core function takes array arguments of various dimensions.

The additional argument *layout* is a string specifying, in symbolic form, the dimensionality and size relationship of the argument types and return types. For example, a matrix multiplication will have a layout string of `"(m, n),(n,p)->(m,p)"`. Its definition might be (function body omitted):

```python
@guvectorize(["void(float64[:,:], float64[:,:], float64[:,:])"],
             "(m,n),(n,p)->(m,p)")
def f(a, b, result):
    """Fill-in *result* matrix such as result := a * b"""
    ...
```

If one of the arguments should be a scalar, the corresponding layout specification is `()` and the argument will really be given to you as a zero-dimension array (you have to dereference it to get the scalar value). For example, a *one-dimension moving average* with a parameterable window width may have a layout string of `"(n),()->(n)"`.

Note that any output will be given to you preallocated as an additional function argument: your code has to fill it with the appropriate values for the function you are implementing.

If your function doesn't take an output array, you should omit the "arrow" in the layout string (e.g. `"(n), (n)"`). When doing this, it is important to be aware that changes to the input arrays cannot always be relied on to be visible outside the execution of the ufunc, as NumPy may pass in temporary arrays as inputs (for example, if a cast is required).

See also:

Specification of the layout string as supported by Numpy. Note that Numpy uses the term "signature", which we unfortunately use for something else.

The compiled function can be cached to reduce future compilation time. It is enabled by setting *cache* to True. Only the "cpu" and "parallel" targets support caching.

class numba.**DUFunc**

The class of objects created by calling `numba.vectorize()` with no signatures.

DUFunc instances should behave similarly to Numpy `ufunc` objects with one important difference: call-time loop generation. When calling a ufunc, Numpy looks at the existing loops registered for that ufunc, and will raise a `TypeError` if it cannot find a loop that it cannot safely cast the inputs to suit. When calling a DUFunc, Numba delegates the call to Numpy. If the Numpy ufunc call fails, then Numba attempts to build a new loop for the given input types, and calls the ufunc again. If this second call attempt fails or a compilation error occurs, then DUFunc passes along the exception to the caller.

See also:

The "*Dynamic universal functions*" section in the user's guide demonstrates the call-time behavior of *DUFunc*, and discusses the impact of call order on how Numba generates the underlying `ufunc`.

ufunc

The actual Numpy `ufunc` object being built by the *DUFunc* instance. Note that the *DUFunc* object maintains several important data structures required for proper ufunc functionality (specifically the dynamically compiled loops). Users should not pass the `ufunc` value around without ensuring the underlying *DUFunc* will not be garbage collected.

nin

The number of DUFunc (ufunc) inputs. See ufunc.nin.

nout

The number of DUFunc outputs. See ufunc.nout.

nargs

The total number of possible DUFunc arguments (should be *nin* + *nout*). See ufunc.nargs.

ntypes

The number of input types supported by the DUFunc. See ufunc.ntypes.

types

A list of the supported types given as strings. See ufunc.types.

identity

The identity value when using the ufunc as a reduction. See ufunc.identity.

reduce(*A*, ***, *axis*, *dtype*, *out*, *keepdims*)

Reduces *A*'s dimension by one by applying the DUFunc along one axis. See ufunc.reduce.

accumulate (*A*, *, *axis*, *dtype*, *out*)

Accumulate the result of applying the operator to all elements. See ufunc.accumulate.

reduceat (*A*, *indices*, *, *axis*, *dtype*, *out*)

Performs a (local) reduce with specified slices over a single axis. See ufunc.reduceat.

outer (*A*, *B*)

Apply the ufunc to all pairs (*a*, *b*) with *a* in *A*, and *b* in *B*. See ufunc.outer.

at (*A*, *indices*, *, *B*)

Performs unbuffered in place operation on operand *A* for elements specified by *indices*. If you are using Numpy 1.7 or earlier, this method will not be present. See ufunc.at.

Note: Vectorized functions can, in rare circumstances, show *unexpected warnings or errors*.

2.2.5 C callbacks

@numba.**cfunc** (*signature*, *nopython=False*, *cache=False*, *locals={}*)

Compile the decorated function on-the-fly to produce efficient machine code. The compiled code is wrapped in a thin C callback that makes it callable using the natural C ABI.

The *signature* is a single signature representing the signature of the C callback. It must have the same form as in *jit()*. The decorator does not check that the types in the signature have a well-defined representation in C.

nopython and *cache* are boolean flags. *locals* is a mapping of local variable names to *Types and signatures*. They all have the same meaning as in *jit()*.

The decorator returns a *CFunc* object.

Note: C callbacks currently do not support *object mode*.

---

class CFunc

The class of objects created by *cfunc()*. *CFunc* objects expose the following attributes and methods:

address

The address of the compiled C callback, as an integer.

cffi

A cffi function pointer instance, to be passed as an argument to cffi-wrapped functions. The pointer's type is void *, so only minimal type checking will happen when passing it to cffi.

ctypes

A ctypes callback instance, as if it were created using ctypes.CFUNCTYPE().

native_name

The name of the compiled C callback.

inspect_llvm()

Return the human-readable LLVM IR generated for the C callback. *native_name* is the name under which this callback is defined in the IR.

2.3 Ahead-of-Time compilation

class numba.pycc.**CC**(*extension_name*, *source_module=None*)

An object used to generate compiled extensions from Numba-compiled Python functions. *extension_name* is the name of the extension to be generated. *source_module* is the Python module containing the functions; if None, it is inferred by examining the call stack.

CC instances have the following attributes and methods:

name

(read-only attribute) The name of the extension module to be generated.

output_dir

(read-write attribute) The directory the extension module will be written into. By default it is the directory the *source_module* is located in.

output_file

(read-write attribute) The name of the file the extension module will be written to. By default this follows the Python naming convention for the current platform.

target_cpu

(read-write attribute) The name of the CPU model to generate code for. This will select the appropriate instruction set extensions. By default, a generic CPU is selected in order to produce portable code.

Recognized names for this attribute depend on the current architecture and LLVM version. If you have LLVM installed, llc -mcpu=help will give you a list. Examples on x86-64 are "ivybridge", "haswell", "skylake" or "broadwell". You can also give the value "host" which will select the current host CPU.

verbose

(read-write attribute) If true, print out information while compiling the extension. False by default.

@**export**(*exported_name*, *sig*)

Mark the decorated function for compilation with the signature *sig*. The compiled function will be exposed as *exported_name* in the generated extension module.

All exported names within a given *CC* instance must be distinct, otherwise an exception is raised.

compile()

Compile all exported functions and generate the extension module as specified by *output_dir* and *output_file*.

distutils_extension(***kwargs*)

Return a distutils.core.Extension instance allowing to integrate generation of the extension module in a conventional setup.py-driven build process. The optional *kwargs* let you pass optional parameters to the Extension constructor.

In this mode of operation, it is not necessary to call *compile()* yourself. Also, *output_dir* and *output_file* will be ignored.

2.4 Utilities

2.4.1 Dealing with pointers

These functions can be called from pure Python as well as in *nopython mode*.

numba.**carray**(*ptr*, *shape*, *dtype=None*)

Return a Numpy array view over the data pointed to by *ptr* with the given *shape*, in C order. If *dtype* is given,

it is used as the array's dtype, otherwise the array's dtype is inferred from *ptr*'s type. As the returned array is a view, not a copy, writing to it will modify the original data.

ptr should be a ctypes pointer object (either a typed pointer as created using `POINTER()`, or a `c_void_p`).

shape should be an integer or a tuple of integers.

dtype should be a Numpy dtype or scalar class (i.e. both `np.dtype('int8')` and `np.int8` are accepted).

numba.**farray**(*ptr*, *shape*, *dtype=None*)
    Same as *carray()*, but the data is assumed to be laid out in Fortran order, and the array view is constructed accordingly.

2.5 Environment variables

Note: This section relates to environment variables that impact Numba's runtime, for compile time environment variables see *Build time environment variables and configuration of optional components*.

Numba allows its behaviour to be changed through the use of environment variables. Unless otherwise mentioned, those variables have integer values and default to zero.

For convenience, Numba also supports the use of a configuration file to persist configuration settings. Note: To use this feature `pyyaml` must be installed.

The configuration file must be named `.numba_config.yaml` and be present in the directory from which the Python interpreter is invoked. The configuration file, if present, is read for configuration settings before the environment variables are searched. This means that the environment variable settings will override the settings obtained from a configuration file (the configuration file is for setting permanent preferences whereas the environment variables are for ephemeral preferences).

The format of the configuration file is a dictionary in `YAML` format that maps the environment variables below (without the `NUMBA_` prefix) to a desired value. For example, to permanently switch on developer mode (`NUMBA_DEVELOPER_MODE` environment variable) and control flow graph printing (`NUMBA_DUMP_CFG` environment variable), create a configuration file with the contents:

```
developer_mode: 1
dump_cfg: 1
```

This can be especially useful in the case of wanting to use a set color scheme based on terminal background color. For example, if the terminal background color is black, the `dark_bg` color scheme would be well suited and can be set for permanent use by adding:

```
color_scheme: dark_bg
```

2.5.1 Jit flags

These variables globally override flags to the *jit()* decorator.

NUMBA_BOUNDSCHECK
    If set to 0 or 1, globally disable or enable bounds checking, respectively. The default if the variable is not set or set to an empty string is to use the `boundscheck` flag passed to the *jit()* decorator for a given function. See the documentation of *@jit* for more information.

Note, due to limitations in numba, the bounds checking currently produces exception messages that do not match those from NumPy. If you set `NUMBA_FULL_TRACEBACKS=1`, the full exception message with the axis, index, and shape information will be printed to the terminal.

2.5.2 Debugging

These variables influence what is printed out during compilation of *JIT functions*.

NUMBA_DEVELOPER_MODE
> If set to non-zero, developer mode produces full tracebacks and disables help instructions. Default is zero.

NUMBA_FULL_TRACEBACKS
> If set to non-zero, enable full tracebacks when an exception occurs. Defaults to the value set by *NUMBA_DEVELOPER_MODE*.

NUMBA_SHOW_HELP
> If set to non-zero, show resources for getting help. Default is zero.

NUMBA_DISABLE_ERROR_MESSAGE_HIGHLIGHTING
> If set to non-zero error message highlighting is disabled. This is useful for running the test suite on CI systems.

NUMBA_COLOR_SCHEME
> Alters the color scheme used in error reporting (requires the `colorama` package to be installed to work). Valid values are:
>
> - `no_color` No color added, just bold font weighting.
>
> - `dark_bg` Suitable for terminals with a dark background.
>
> - `light_bg` Suitable for terminals with a light background.
>
> - `blue_bg` Suitable for terminals with a blue background.
>
> - `jupyter_nb` Suitable for use in Jupyter Notebooks.
>
> *Default value:* `no_color`. The type of the value is `string`.

NUMBA_HIGHLIGHT_DUMPS
> If set to non-zero and `pygments` is installed, syntax highlighting is applied to Numba IR, LLVM IR and assembly dumps. Default is zero.

NUMBA_DISABLE_PERFORMANCE_WARNINGS
> If set to non-zero the issuing of performance warnings is disabled. Default is zero.

NUMBA_DEBUG
> If set to non-zero, print out all possible debugging information during function compilation. Finer-grained control can be obtained using other variables below.

NUMBA_DEBUG_FRONTEND
> If set to non-zero, print out debugging information during operation of the compiler frontend, up to and including generation of the Numba Intermediate Representation.

NUMBA_DEBUGINFO
> If set to non-zero, enable debug for the full application by setting the default value of the `debug` option in `jit`. Beware that enabling debug info significantly increases the memory consumption for each compiled function. Default value equals to the value of *NUMBA_ENABLE_PROFILING*.

NUMBA_GDB_BINARY
> Set the `gdb` binary for use in Numba's `gdb` support, this takes the form of a path and full name of the binary, for example: `/path/from/root/to/binary/name_of_gdb_binary` This is to permit the use of a `gdb` from a non-default location with a non-default name. If not set `gdb` is assumed to reside at `/usr/bin/gdb`.

NUMBA_DEBUG_TYPEINFER
> If set to non-zero, print out debugging information about type inference.

NUMBA_ENABLE_PROFILING
> Enables JIT events of LLVM in order to support profiling of jitted functions. This option is automatically enabled under certain profilers.

NUMBA_TRACE
> If set to non-zero, trace certain function calls (function entry and exit events, including arguments and return values).

NUMBA_DUMP_BYTECODE
> If set to non-zero, print out the Python *bytecode* of compiled functions.

NUMBA_DUMP_CFG
> If set to non-zero, print out information about the Control Flow Graph of compiled functions.

NUMBA_DUMP_IR
> If set to non-zero, print out the Numba Intermediate Representation of compiled functions.

NUMBA_DUMP_SSA
> If set to non-zero, print out the Numba Intermediate Representation of compiled functions after conversion to Static Single Assignment (SSA) form.

NUMBA_DEBUG_PRINT_AFTER
> Dump the Numba IR after declared pass(es). This is useful for debugging IR changes made by given passes. Accepted values are:
>
> - Any pass name (as given by the `.name()` method on the class)
>
> - Multiple pass names as a comma separated list, i.e. `"foo_pass,bar_pass"`
>
> - The token `"all"`, which will print after all passes.
>
> The default value is `"none"` so as to prevent output.

NUMBA_DUMP_ANNOTATION
> If set to non-zero, print out types annotations for compiled functions.

NUMBA_DUMP_LLVM
> Dump the unoptimized LLVM assembly source of compiled functions. Unoptimized code is usually very verbose; therefore, *NUMBA_DUMP_OPTIMIZED* is recommended instead.

NUMBA_DUMP_FUNC_OPT
> Dump the LLVM assembly source after the LLVM "function optimization" pass, but before the "module optimization" pass. This is useful mostly when developing Numba itself, otherwise use *NUMBA_DUMP_OPTIMIZED*.

NUMBA_DUMP_OPTIMIZED
> Dump the LLVM assembly source of compiled functions after all optimization passes. The output includes the raw function as well as its CPython-compatible wrapper (whose name begins with `wrapper.`). Note that the function is often inlined inside the wrapper, as well.

NUMBA_DEBUG_ARRAY_OPT
> Dump debugging information related to the processing associated with the `parallel=True` jit decorator option.

NUMBA_DEBUG_ARRAY_OPT_RUNTIME
> Dump debugging information related to the runtime scheduler associated with the `parallel=True` jit decorator option.

NUMBA_DEBUG_ARRAY_OPT_STATS

Dump statistics about how many operators/calls are converted to parallel for-loops and how many are fused together, which are associated with the `parallel=True` jit decorator option.

NUMBA_PARALLEL_DIAGNOSTICS

If set to an integer value between 1 and 4 (inclusive) diagnostic information about parallel transforms undertaken by Numba will be written to STDOUT. The higher the value set the more detailed the information produced.

NUMBA_DUMP_ASSEMBLY

Dump the native assembly code of compiled functions.

See also:

Troubleshooting and tips and *Numba architecture*.

2.5.3 Compilation options

NUMBA_OPT

The optimization level; this option is passed straight to LLVM.

Default value: 3

NUMBA_LOOP_VECTORIZE

If set to non-zero, enable LLVM loop vectorization.

Default value: 1 (except on 32-bit Windows)

NUMBA_ENABLE_AVX

If set to non-zero, enable AVX optimizations in LLVM. This is disabled by default on Sandy Bridge and Ivy Bridge architectures as it can sometimes result in slower code on those platforms.

NUMBA_DISABLE_INTEL_SVML

If set to non-zero and Intel SVML is available, the use of SVML will be disabled.

NUMBA_COMPATIBILITY_MODE

If set to non-zero, compilation of JIT functions will never entirely fail, but instead generate a fallback that simply interprets the function. This is only to be used if you are migrating a large codebase from an old Numba version (before 0.12), and want to avoid breaking everything at once. Otherwise, please don't use this.

NUMBA_DISABLE_JIT

Disable JIT compilation entirely. The *jit()* decorator acts as if it performs no operation, and the invocation of decorated functions calls the original Python function instead of a compiled version. This can be useful if you want to run the Python debugger over your code.

NUMBA_CPU_NAME

NUMBA_CPU_FEATURES

Override CPU and CPU features detection. By setting `NUMBA_CPU_NAME=generic`, a generic CPU model is picked for the CPU architecture and the feature list (`NUMBA_CPU_FEATURES`) defaults to empty. CPU features must be listed with the format `+feature1,-feature2` where + indicates enable and − indicates disable. For example, `+sse,+sse2,-avx,-avx2` enables SSE and SSE2, and disables AVX and AVX2.

These settings are passed to LLVM for configuring the compilation target. To get a list of available options, use the `llc` commandline tool from LLVM, for example:

```
llc -march=x86 -mattr=help
```

Tip: To force all caching functions (`@jit(cache=True)`) to emit portable code (portable within the same architecture and OS), simply set `NUMBA_CPU_NAME=generic`.

NUMBA_FUNCTION_CACHE_SIZE

Override the size of the function cache for retaining recently deserialized functions in memory. In systems like Dask, it is common for functions to be deserialized multiple times. Numba will cache functions as long as there is a reference somewhere in the interpreter. This cache size variable controls how many functions that are no longer referenced will also be retained, just in case they show up in the future. The implementation of this is not a true LRU, but the large size of the cache should be sufficient for most situations.

Note: this is unrelated to the compilation cache.

Default value: 128

2.5.4 Caching options

Options for the compilation cache.

NUMBA_DEBUG_CACHE

If set to non-zero, print out information about operation of the *JIT compilation cache*.

NUMBA_CACHE_DIR

Override the location of the cache directory. If defined, this should be a valid directory path.

If not defined, Numba picks the cache directory in the following order:

1. In-tree cache. Put the cache next to the corresponding source file under a `__pycache__` directory following how `.pyc` files are stored.

2. User-wide cache. Put the cache in the user's application directory using `appdirs.user_cache_dir` from the Appdirs package.

3. IPython cache. Put the cache in an IPython specific application directory. Stores are made under the `numba_cache` in the directory returned by `IPython.paths.get_ipython_cache_dir()`.

Also see *docs on cache sharing* and *docs on cache clearing*

2.5.5 GPU support

NUMBA_DISABLE_CUDA

If set to non-zero, disable CUDA support.

NUMBA_FORCE_CUDA_CC

If set, force the CUDA compute capability to the given version (a string of the type `major.minor`), regardless of attached devices.

NUMBA_CUDA_DEFAULT_PTX_CC

The default compute capability (a string of the type `major.minor`) to target when compiling to PTX using `cuda.compile_ptx`. The default is 5.2, which is the lowest non-deprecated compute capability in the most recent version of the CUDA toolkit supported (10.2 at present).

NUMBA_ENABLE_CUDASIM

If set, don't compile and execute code for the GPU, but use the CUDA Simulator instead. For debugging purposes.

2.5.6 Threading Control

NUMBA_NUM_THREADS

> If set, the number of threads in the thread pool for the parallel CPU target will take this value. Must be greater than zero. This value is independent of `OMP_NUM_THREADS` and `MKL_NUM_THREADS`.
>
> *Default value:* The number of CPU cores on the system as determined at run time. This can be accessed via `numba.config.NUMBA_DEFAULT_NUM_THREADS`.
>
> See also the section on *Setting the Number of Threads* for information on how to set the number of threads at runtime.

NUMBA_THREADING_LAYER

> This environment variable controls the library used for concurrent execution for the CPU parallel targets (`@vectorize(target='parallel')`, `@guvectorize(target='parallel')` and `@njit(parallel=True)`). The variable type is string and by default is `default` which will select a threading layer based on what is available in the runtime. The valid values are (for more information about these see *the threading layer documentation*):
>
> - `default` - select a threading layer based on what is available in the current runtime.
> - `safe` - select a threading layer that is both fork and thread safe (requires the TBB package).
> - `forksafe` - select a threading layer that is fork safe.
> - `threadsafe` - select a threading layer that is thread safe.
> - `tbb` - A threading layer backed by Intel TBB.
> - `omp` - A threading layer backed by OpenMP.
> - `workqueue` - A simple built-in work-sharing task scheduler.

2.6 Supported Python features

Apart from the *Language* part below, which applies to both *object mode* and *nopython mode*, this page only lists the features supported in *nopython mode*.

> **Warning:** Numba behavior differs from Python semantics in some situations. We strongly advise reviewing *Deviations from Python Semantics* to become familiar with these differences.

2.6.1 Language

Constructs

Numba strives to support as much of the Python language as possible, but some language features are not available inside Numba-compiled functions. Below is a quick reference for the support level of Python constructs.

Supported constructs:

- conditional branch: `if .. elif .. else`
- loops: `while`, `for .. in`, `break`, `continue`
- basic generator: `yield`
- assertion: `assert`

Partially supported constructs:

- exceptions: `try .. except`, `raise`, `else` and `finally` (See details in this *section*)

- context manager: `with` (only support *numba.objmode()*)

- list comprehension (see details in this *section*)

Unsupported constructs:

- async features: `async with`, `async for` and `async def`

- class definition: `class` (except for *@jitclass*)

- set, dict and generator comprehensions

- generator delegation: `yield from`

Functions

Function calls

Numba supports function calls using positional and named arguments, as well as arguments with default values and `*args` (note the argument for `*args` can only be a tuple, not a list). Explicit `**kwargs` are not supported.

Function calls to locally defined inner functions are supported as long as they can be fully inlined.

Functions as arguments

Functions can be passed as argument into another function. But, they cannot be returned. For example:

```python
from numba import jit

@jit
def add1(x):
    return x + 1

@jit
def bar(fn, x):
    return fn(x)

@jit
def foo(x):
    return bar(add1, x)

# Passing add1 within numba compiled code.
print(foo(1))
# Passing add1 into bar from interpreted code
print(bar(add1, 1))
```

Note: Numba does not handle function objects as real objects. Once a function is assigned to a variable, the variable cannot be re-assigned to a different function.

Inner function and closure

Numba now supports inner functions as long as they are non-recursive and only called locally, but not passed as argument or returned as result. The use of closure variables (variables defined in outer scopes) within an inner function is also supported.

Recursive calls

Most recursive call patterns are supported. The only restriction is that the recursive callee must have a control-flow path that returns without recursing. Numba is able to type-infer recursive functions without specifying the function type signature (which is required in numba 0.28 and earlier). Recursive calls can even call into a different overload of the function.

Generators

Numba supports generator functions and is able to compile them in *object mode* and *nopython mode*. The returned generator can be used both from Numba-compiled code and from regular Python code.

Coroutine features of generators are not supported (i.e. the `generator.send()`, `generator.throw()`, `generator.close()` methods).

Exception handling

`raise` statement

The `raise` statement is only supported in the following forms:

- `raise SomeException`

- `raise SomeException(<arguments>)`: in *nopython mode*, constructor arguments must be *compile-time constants*

It is currently unsupported to re-raise an exception created in compiled code.

`try .. except`

The `try .. except` construct is partially supported. The following forms of are supported:

- the *bare* except that captures all exceptions:

```
try:
    ...
except:
    ...
```

- using exactly the `Exception` class in the `except` clause:

```
try:
    ...
except Exception:
    ...
```

    This will match any exception that is a subclass of `Exception` as expected. Currently, instances of `Exception` and it's subclasses are the only kind of exception that can be raised in compiled code.

> **Warning:** Numba currently masks signals like `KeyboardInterrupt` and `SystemExit`. These signaling exceptions are ignored during the execution of Numba compiled code. The Python interpreter will handle them as soon as the control is returned to it.

Currently, exception objects are not materialized inside compiled functions. As a result, it is not possible to store an exception object into a user variable or to re-raise an exception. With this limitation, the only realistic use-case would look like:

```python
try:
    do_work()
except Exception:
    handle_error_case()
    return error_code
```

try .. except .. else .. finally

The `else` block and the `finally` block of a `try .. except` are supported:

```python
>>> @jit(nopython=True)
... def foo():
...     try:
...         print('main block')
...     except Exception:
...         print('handler block')
...     else:
...         print('else block')
...     finally:
...         print('final block')
...
>>> foo()
main block
else block
final block
```

The `try .. finally` construct without the `except` clause is also supported.

2.6.2 Built-in types

int, bool

Arithmetic operations as well as truth values are supported.

The following attributes and methods are supported:

- `.conjugate()`
- `.real`
- `.imag`

float, complex

Arithmetic operations as well as truth values are supported.

The following attributes and methods are supported:

- `.conjugate()`
- `.real`
- `.imag`

str

Numba supports (Unicode) strings in Python 3. Strings can be passed into *nopython mode* as arguments, as well as constructed and returned from *nopython mode*. As in Python, slices (even of length 1) return a new, reference counted string. Optimized code paths for efficiently accessing single characters may be introduced in the future.

The in-memory representation is the same as was introduced in Python 3.4, with each string having a tag to indicate whether the string is using a 1, 2, or 4 byte character width in memory. When strings of different encodings are combined (as in concatenation), the resulting string automatically uses the larger character width of the two input strings. String slices also use the same character width as the original string, even if the slice could be represented with a narrower character width. (These details are invisible to the user, of course.)

The following constructors, functions, attributes and methods are currently supported:

- `str(int)`
- `len()`
- + (concatenation of strings)
- * (repetition of strings)
- `in`, `.contains()`
- ==, <, <=, >, >= (comparison)
- `.capitalize()`
- `.casefold()`
- `.center()`
- `.count()`
- `.endswith()`
- `.endswith()`
- `.expandtabs()`
- `.find()`
- `.index()`
- `.isalnum()`
- `.isalpha()`
- `.isdecimal()`
- `.isdigit()`
- `.isidentifier()`
- `.islower()`
- `.isnumeric()`
- `.isprintable()`

2.6. Supported Python features 105

- `.isspace()`
- `.istitle()`
- `.isupper()`
- `.join()`
- `.ljust()`
- `.lower()`
- `.lstrip()`
- `.partition()`
- `.replace()`
- `.rfind()`
- `.rindex()`
- `.rjust()`
- `.rpartition()`
- `.rsplit()`
- `.rstrip()`
- `.split()`
- `.splitlines()`
- `.startswith()`
- `.strip()`
- `.swapcase()`
- `.title()`
- `.upper()`
- `.zfill()`

Additional operations as well as support for Python 2 strings / Python 3 bytes will be added in a future version of Numba. Python 2 Unicode objects will likely never be supported.

> **Warning:** The performance of some operations is known to be slower than the CPython implementation. These include substring search (`in`, `.contains()` and `find()`) and string creation (like `.split()`). Improving the string performance is an ongoing task, but the speed of CPython is unlikely to be surpassed for basic string operation in isolation. Numba is most successfully used for larger algorithms that happen to involve strings, where basic string operations are not the bottleneck.

tuple

Tuple support is categorised into two categories based on the contents of a tuple. The first category is homogeneous tuples, these are tuples where the type of all the values in the tuple are the same, the second is heterogeneous tuples, these are tuples where the types of the values are different.

> **Note:** The `tuple()` constructor itself is NOT supported.

homogeneous tuples

An example of a homogeneous tuple:

```
homogeneous_tuple = (1, 2, 3, 4)
```

The following operations are supported on homogeneous tuples:

- Tuple construction.
- Tuple unpacking.
- Comparison between tuples.
- Iteration and indexing.
- Addition (concatenation) between tuples.
- Slicing tuples with a constant slice.
- The index method on tuples.

heterogeneous tuples

An example of a heterogeneous tuple:

```
heterogeneous_tuple = (1, 2j, 3.0, "a")
```

The following operations are supported on heterogeneous tuples:

- Comparison between tuples.
- Indexing using an index value that is a compile time constant e.g. `mytuple[7]`, where 7 is evidently a constant.
- Iteration over a tuple (requires experimental `literal_unroll()` feature, see below).

> **Warning:** The following feature (`literal_unroll()`) is experimental and was added in version 0.47.

To permit iteration over a heterogeneous tuple the special function `numba.literal_unroll()` must be used. This function has no effect other than to act as a token to permit the use of this feature. Example use:

```python
from numba import njit, literal_unroll

@njit
def foo()
    heterogeneous_tuple = (1, 2j, 3.0, "a")
    for i in literal_unroll(heterogeneous_tuple):
        print(i)
```

> **Warning:** The following restrictions apply to the use of `literal_unroll()`:
>
> - This feature is only available for Python versions >= 3.6.
> - `literal_unroll()` can only be used on tuples and constant lists of compile time constants, e.g. `[1, 2j, 3, "a"]` and the list not being mutated.
> - The only supported use pattern for `literal_unroll()` is loop iteration.

> - Only one `literal_unroll()` call is permitted per loop nest (i.e. nested heterogeneous tuple iteration loops are forbidden).
>
> - The usual type inference/stability rules still apply.

A more involved use of `literal_unroll()` might be type specific dispatch, recall that string and integer literal values are considered their own type, for example:

```python
from numba import njit, types, literal_unroll
from numba.extending import overload

def dt(x):
    # dummy function to overload
    pass

@overload(dt, inline='always')
def ol_dt(li):
    if isinstance(li, types.StringLiteral):
        value = li.literal_value
        if value == "apple":
            def impl(li):
                return 1
        elif value == "orange":
            def impl(li):
                return 2
        elif value == "banana":
            def impl(li):
                return 3
        return impl
    elif isinstance(li, types.IntegerLiteral):
        value = li.literal_value
        if value == 0xca11ab1e:
            def impl(li):
                # capture the dispatcher literal value
                return 0x5ca1ab1e + value
            return impl

@njit
def foo():
    acc = 0
    for t in literal_unroll(('apple', 'orange', 'banana', 3390155550)):
        acc += dt(t)
    return acc

print(foo())
```

list

> **Warning:** As of version 0.45.x the internal implementation for the list datatype in Numba is changing. Until recently, only a single implementation of the list datatype was available, the so-called *reflected-list* (see below). However, it was scheduled for deprecation from version 0.44.0 onwards due to its limitations. As of version 0.45.0 a new implementation, the so-called *typed-list* (see below), is available as an experimental feature. For more information, please see: *Deprecation Notices*.

Creating and returning lists from JIT-compiled functions is supported, as well as all methods and operations. Lists must be strictly homogeneous: Numba will reject any list containing objects of different types, even if the types are compatible (for example, `[1, 2.5]` is rejected as it contains a `int` and a `float`).

For example, to create a list of arrays:

```
In [1]: from numba import njit

In [2]: import numpy as np

In [3]: @njit
   ...: def foo(x):
   ...:     lst = []
   ...:     for i in range(x):
   ...:         lst.append(np.arange(i))
   ...:     return lst
   ...:

In [4]: foo(4)
Out[4]: [array([], dtype=int64), array([0]), array([0, 1]), array([0, 1, 2])]
```

List Reflection

In nopython mode, Numba does not operate on Python objects. `list` are compiled into an internal representation. Any `list` arguments must be converted into this representation on the way in to nopython mode and their contained elements must be restored in the original Python objects via a process called *reflection*. Reflection is required to maintain the same semantics as found in regular Python code. However, the reflection process can be expensive for large lists and it is not supported for lists that contain reflected data types. Users cannot use list-of-list as an argument because of this limitation.

Note: When passing a list into a JIT-compiled function, any modifications made to the list will not be visible to the Python interpreter until the function returns. (A limitation of the reflection process.)

> **Warning:** List sorting currently uses a quicksort algorithm, which has different performance characterics than the algorithm used by Python.

Initial Values

> **Warning:** This is an experimental feature!

Lists that:

- Are constructed using the square braces syntax
- Have values of a literal type

will have their initial value stored in the `.initial_value` property on the type so as to permit inspection of these values at compile time. If required, to force value based dispatch the *literally* function will accept such a list.

Example:

Listing 1: from `test_ex_initial_value_list_compile_time_consts`
of `numba/tests/doc_examples/`
`test_literal_container_usage.py`

```python
from numba import njit, literally
from numba.extending import overload

# overload this function
def specialize(x):
    pass

@overload(specialize)
def ol_specialize(x):
    iv = x.initial_value
    if iv is None:
        return lambda x: literally(x) # Force literal dispatch
    assert iv == [1, 2, 3] # INITIAL VALUE
    return lambda x: x

@njit
def foo():
    l = [1, 2, 3]
    l[2] = 20 # no impact on .initial_value
    l.append(30) # no impact on .initial_value
    return specialize(l)

result = foo()
print(result) # [1, 2, 20, 30] # NOT INITIAL VALUE!
```

Typed List

Note: `numba.typed.List` is an experimental feature, if you encounter any bugs in functionality or suffer from unexpectedly bad performance, please report this, ideally by opening an issue on the Numba issue tracker.

As of version 0.45.0 a new implementation of the list data type is available, the so-called *typed-list*. This is compiled library backed, type-homogeneous list data type that is an improvement over the *reflected-list* mentioned above. Additionally, lists can now be arbitrarily nested. Since the implementation is considered experimental, you will need to import it explicitly from the *numba.typed* module:

```python
In [1]: from numba.typed import List

In [2]: from numba import njit

In [3]: @njit
...: def foo(l):
...:     l.append(23)
...:     return l
...:

In [4]: mylist = List()

In [5]: mylist.append(1)
```

(continues on next page)

```
In [6]: foo(mylist)
Out[6]: ListType[int64]([1, 23])
```

Note: As the typed-list stabilizes it will fully replace the reflected-list and the constructors *[]* and *list()* will create a typed-list instead of a reflected one.

Here's an example using `List()` to create `numba.typed.List` inside a jit-compiled function and letting the compiler infer the item type:

Listing 2: from `ex_inferred_list_jit` of `numba/tests/doc_examples/test_typed_list_usage.py`

```python
from numba import njit
from numba.typed import List


@njit
def foo():
    # Instantiate a typed-list
    l = List()
    # Append a value to it, this will set the type to int32/int64
    # (depending on platform)
    l.append(42)
    # The usual list operations, getitem, pop and length are
    # supported
    print(l[0])    # 42
    l[0] = 23
    print(l[0])    # 23
    print(len(l)) # 1
    l.pop()
    print(len(l)) # 0
    return l


foo()

```

Here's an example of using `List()` to create a `numba.typed.List` outside of a jit-compiled function and then using it as an argument to a jit-compiled function:

Listing 3: from `ex_inferred_list` of `numba/tests/doc_examples/test_typed_list_usage.py`

```python
from numba import njit
from numba.typed import List


@njit
def foo(mylist):
    for i in range(10, 20):
        mylist.append(i)
    return mylist


# Instantiate a typed-list, outside of a jit context
l = List()
# Append a value to it, this will set the type to int32/int64
# (depending on platform)
```

```
14  l.append(42)
15  # The usual list operations, getitem, pop and length are supported
16  print(l[0])   # 42
17  l[0] = 23
18  print(l[0])   # 23
19  print(len(l)) # 1
20  l.pop()
21  print(len(l)) # 0
22
23  # And you can use the typed-list as an argument for a jit compiled
24  # function
25  l = foo(l)
26  print(len(l)) # 10
27
```

Finally, here's an example of using a nested *List()*:

Listing 4: from `ex_nested_list` of `numba/tests/ doc_examples/test_typed_list_usage.py`

```
1   from numba.typed import List
2
3   # typed-lists can be nested in typed-lists
4   mylist = List()
5   for i in range(10):
6       l = List()
7       for i in range(10):
8           l.append(i)
9       mylist.append(l)
10  # mylist is now a list of 10 lists, each containing 10 integers
11  print(mylist)
12
```

Literal List

> **Warning:** This is an experimental feature!

Numba supports the use of literal lists containing any values, for example:

```
l = ['a', 1, 2j, np.zeros(5,)]
```

the predominant use of these lists is for use as a configuration object. The lists appear as a `LiteralList` type which inherits from `Literal`, as a result the literal values of the list items are available at compile time. For example:

Listing 5: from `test_ex_literal_list` of `numba/tests/ doc_examples/test_literal_container_usage.py`

```
1   from numba import njit
2   from numba.extending import overload
3
4   # overload this function
5   def specialize(x):
```

```
6        pass
7
8  @overload(specialize)
9  def ol_specialize(x):
10       l = x.literal_value
11       const_expr = []
12       for v in l:
13           const_expr.append(str(v))
14       const_strings = tuple(const_expr)
15
16       def impl(x):
17           return const_strings
18       return impl
19
20  @njit
21  def foo():
22       const_list = ['a', 10, 1j, ['another', 'list']]
23       return specialize(const_list)
24
25  result = foo()
26  print(result) # ('Literal[str](a)', 'Literal[int](10)', 'complex128', 'list(unicode_
   ↪type)') # noqa E501
```

Important things to note about these kinds of lists:

1. They are immutable, use of mutating methods e.g. `.pop()` will result in compilation failure. Read-only static access and read only methods are supported e.g. `len()`.

2. Dynamic access of items is not possible, e.g. `some_list[x]`, for a value `x` which is not a compile time constant. This is because it's impossible statically determine the type of the item being accessed.

3. Inside the compiler, these lists are actually just tuples with some extra things added to make them look like they are lists.

4. They cannot be returned to the interpreter from a compiled function.

List comprehension

Numba supports list comprehension. For example:

```
In [1]: from numba import njit

In [2]: @njit
   ...: def foo(x):
   ...:     return [[i for i in range(n)] for n in range(x)]
   ...:

In [3]: foo(3)
Out[3]: [[], [0], [0, 1]]
```

Note: Prior to version 0.39.0, Numba did not support the creation of nested lists.

Numba also supports "array comprehension" that is a list comprehension followed immediately by a call to *numpy.array()*. The following is an example that produces a 2D Numpy array:

```
from numba import jit
import numpy as np

@jit(nopython=True)
def f(n):
  return np.array([ [ x * y for x in range(n) ] for y in range(n) ])
```

In this case, Numba is able to optimize the program to allocate and initialize the result array directly without allocating intermediate list objects. Therefore, the nesting of list comprehension here is not a problem since a multi-dimensional array is being created here instead of a nested list.

Additionally, Numba supports parallel array comprehension when combined with the *parallel* option on CPUs.

set

All methods and operations on sets are supported in JIT-compiled functions.

Sets must be strictly homogeneous: Numba will reject any set containing objects of different types, even if the types are compatible (for example, {1, 2.5} is rejected as it contains a `int` and a `float`).

Note: When passing a set into a JIT-compiled function, any modifications made to the set will not be visible to the Python interpreter until the function returns.

Typed Dict

Warning: `numba.typed.Dict` is an experimental feature. The API may change in the future releases.

Note: `dict()` was not supported in versions prior to 0.44. Currently, calling `dict()` translates to calling `numba.typed.Dict()`.

Numba only supports the use of `dict()` without any arguments. Such use is semantically equivalent to `{}` and `numba.typed.Dict()`. It will create an instance of `numba.typed.Dict` where the key-value types will be later inferred by usage.

Numba does not fully support the Python `dict` because it is an untyped container that can have any Python types as members. To generate efficient machine code, Numba needs the keys and the values of the dictionary to have fixed types, declared in advance. To achieve this, Numba has a typed dictionary, `numba.typed.Dict`, for which the type-inference mechanism must be able to infer the key-value types by use, or the user must explicitly declare the key-value type using the `Dict.empty()` constructor method. This typed dictionary has the same API as the Python `dict`, it implements the `collections.MutableMapping` interface and is usable in both interpreted Python code and JIT-compiled Numba functions. Because the typed dictionary stores keys and values in Numba's native, unboxed data layout, passing a Numba dictionary into nopython mode has very low overhead. However, this means that using a typed dictionary from the Python interpreter is slower than a regular dictionary because Numba has to box and unbox key and value objects when getting or setting items.

An important difference of the typed dictionary in comparison to Python's `dict` is that **implicit casting** occurs when a key or value is stored. As a result the *setitem* operation may fail should the type-casting fail.

It should be noted that the Numba typed dictionary is implemented using the same algorithm as the CPython 3.7 dictionary. As a consequence, the typed dictionary is ordered and has the same collision resolution as the CPython implementation.

Further to the above in relation to type specification, there are limitations placed on the types that can be used as keys and/or values in the typed dictionary, most notably the Numba `Set` and `List` types are currently unsupported. Acceptable key/value types include but are not limited to: unicode strings, arrays (value only), scalars, tuples. It is expected that these limitations will be relaxed as Numba continues to improve.

Here's an example of using `dict()` and `{}` to create `numba.typed.Dict` instances and letting the compiler infer the key-value types:

Listing 6: from `test_ex_inferred_dict_njit` of `numba/`
`tests/doc_examples/test_typed_dict_usage.py`

```
1   from numba import njit
2   import numpy as np
3
4   @njit
5   def foo():
6       d = dict()
7       k = {1: np.arange(1), 2: np.arange(2)}
8       # The following tells the compiler what the key type and the
9       # value
10      # type are for `d`.
11      d[3] = np.arange(3)
12      d[5] = np.arange(5)
13      return d, k
14
15  d, k = foo()
16  print(d)      # {3: [0 1 2], 5: [0 1 2 3 4]}
17  print(k)      # {1: [0], 2: [0 1]}
```

Here's an example of creating a `numba.typed.Dict` instance from interpreted code and using the dictionary in jit code:

Listing 7: from `test_ex_typed_dict_from_cpython` of
`numba/tests/doc_examples/test_typed_dict_usage.`
`py`

```
1   import numpy as np
2   from numba import njit
3   from numba.core import types
4   from numba.typed import Dict
5
6   # The Dict.empty() constructs a typed dictionary.
7   # The key and value typed must be explicitly declared.
8   d = Dict.empty(
9       key_type=types.unicode_type,
10      value_type=types.float64[:],
11  )
12
13  # The typed-dict can be used from the interpreter.
14  d['posx'] = np.asarray([1, 0.5, 2], dtype='f8')
15  d['posy'] = np.asarray([1.5, 3.5, 2], dtype='f8')
16  d['velx'] = np.asarray([0.5, 0, 0.7], dtype='f8')
17  d['vely'] = np.asarray([0.2, -0.2, 0.1], dtype='f8')
18
```

(continues on next page)

```python
19  # Here's a function that expects a typed-dict as the argument
20  @njit
21  def move(d):
22      # inplace operations on the arrays
23      d['posx'] += d['velx']
24      d['posy'] += d['vely']
25
26  print('posx: ', d['posx'])  # Out: posx:  [1.   0.5 2. ]
27  print('posy: ', d['posy'])  # Out: posy:  [1.5 3.5 2. ]
28
29  # Call move(d) to inplace update the arrays in the typed-dict.
30  move(d)
31
32  print('posx: ', d['posx'])  # Out: posx:  [1.5 0.5 2.7]
33  print('posy: ', d['posy'])  # Out: posy:  [1.7 3.3 2.1]
```

Here's an example of creating a `numba.typed.Dict` instance from jit code and using the dictionary in interpreted code:

Listing 8: from `test_ex_typed_dict_njit` of `numba/tests/ doc_examples/test_typed_dict_usage.py`

```python
1   import numpy as np
2   from numba import njit
3   from numba.core import types
4   from numba.typed import Dict
5
6   # Make array type.  Type-expression is not supported in jit
7   # functions.
8   float_array = types.float64[:]
9
10  @njit
11  def foo():
12      # Make dictionary
13      d = Dict.empty(
14          key_type=types.unicode_type,
15          value_type=float_array,
16      )
17      # Fill the dictionary
18      d["posx"] = np.arange(3).astype(np.float64)
19      d["posy"] = np.arange(3, 6).astype(np.float64)
20      return d
21
22  d = foo()
23  # Print the dictionary
24  print(d)  # Out: {posx: [0. 1. 2.], posy: [3. 4. 5.]}
```

It should be noted that `numba.typed.Dict` is not thread-safe. Specifically, functions which modify a dictionary from multiple threads will potentially corrupt memory, causing a range of possible failures. However, the dictionary can be safely read from multiple threads as long as the contents of the dictionary do not change during the parallel access.

Initial Values

> **Warning:** This is an experimental feature!

Typed dictionaries that:

- Are constructed using the curly braces syntax

- Have literal string keys

- Have values of a literal type

will have their initial value stored in the `.initial_value` property on the type so as to permit inspection of these values at compile time. If required, to force value based dispatch the *literally* function will accept a typed dictionary.

Example:

> Listing 9: from `test_ex_initial_value_dict_compile_time_consts` of `numba/tests/doc_examples/` `test_literal_container_usage.py`

```python
from numba import njit, literally
from numba.extending import overload

# overload this function
def specialize(x):
    pass

@overload(specialize)
def ol_specialize(x):
    iv = x.initial_value
    if iv is None:
        return lambda x: literally(x) # Force literal dispatch
    assert iv == {'a': 1, 'b': 2, 'c': 3} # INITIAL VALUE
    return lambda x: literally(x)

@njit
def foo():
    d = {'a': 1, 'b': 2, 'c': 3}
    d['c'] = 20 # no impact on .initial_value
    d['d'] = 30 # no impact on .initial_value
    return specialize(d)

result = foo()
print(result) # {a: 1, b: 2, c: 20, d: 30} # NOT INITIAL VALUE!
```

Heterogeneous Literal String Key Dictionary

> **Warning:** This is an experimental feature!

Numba supports the use of statically declared string key to any value dictionaries, for example:

```python
d = {'a': 1, 'b': 'data', 'c': 2j}
```

the predominant use of these dictionaries is to orchestrate advanced compilation dispatch or as a container for use as a configuration object. The dictionaries appear as a `LiteralStrKeyDict` type which inherits from `Literal`, as a result the literal values of the keys and the types of the items are available at compile time. For example:

Listing 10: from `test_ex_literal_dict_compile_time_consts` of `numba/tests/doc_examples/` `test_literal_container_usage.py`

```python
1   import numpy as np
2   from numba import njit, types
3   from numba.extending import overload
4
5   # overload this function
6   def specialize(x):
7       pass
8
9   @overload(specialize)
10  def ol_specialize(x):
11      ld = x.literal_value
12      const_expr = []
13      for k, v in ld.items():
14          if isinstance(v, types.Literal):
15              lv = v.literal_value
16              if lv == 'cat':
17                  const_expr.append("Meow!")
18              elif lv == 'dog':
19                  const_expr.append("Woof!")
20              elif isinstance(lv, int):
21                  const_expr.append(k.literal_value * lv)
22          else: # it's an array
23              const_expr.append("Array(dim={dim}".format(dim=v.ndim))
24      const_strings = tuple(const_expr)
25
26      def impl(x):
27          return const_strings
28      return impl
29
30  @njit
31  def foo():
32      pets_ints_and_array = {'a': 1,
33                             'b': 2,
34                             'c': 'cat',
35                             'd': 'dog',
36                             'e': np.ones(5,)}
37      return specialize(pets_ints_and_array)
38
39  result = foo()
40  print(result) # ('a', 'bb', 'Meow!', 'Woof!', 'Array(dim=1')
```

Important things to note about these kinds of dictionaries:

1. They are immutable, use of mutating methods e.g. `.pop()` will result in compilation failure. Read-only static access and read only methods are supported e.g. `len()`.

2. Dynamic access of items is not possible, e.g. `some_dictionary[x]`, for a value `x` which is not a compile time constant. This is because it's impossible statically determine the type of the item being accessed.

3. Inside the compiler, these dictionaries are actually just named tuples with some extra things added to make them look like they are dictionaries.

4. They cannot be returned to the interpreter from a compiled function.

5. The `.keys()`, `.values()` and `.items()` methods all functionally operate but return tuples opposed to iterables.

None

The None value is supported for identity testing (when using an *optional* type).

bytes, bytearray, memoryview

The `bytearray` type and, on Python 3, the `bytes` type support indexing, iteration and retrieving the len().

The `memoryview` type supports indexing, slicing, iteration, retrieving the len(), and also the following attributes:

- `contiguous`
- `c_contiguous`
- `f_contiguous`
- `itemsize`
- `nbytes`
- `ndim`
- `readonly`
- `shape`
- `strides`

2.6.3 Built-in functions

The following built-in functions are supported:

- `abs()`
- `bool`
- `chr()`
- `complex`
- `divmod()`
- `enumerate()`
- `filter()`
- `float`
- `hash()` (see *Hashing* below)
- `int`: only the one-argument form
- `iter()`: only the one-argument form
- `len()`
- `min()`
- `map()`

- `max()`
- `next()`: only the one-argument form
- `ord()`
- `print()`: only numbers and strings; no `file` or `sep` argument
- `range`: The only permitted use of range is as a callable function (cannot pass range as an argument to a jitted function or return a range from a jitted function).
- `round()`
- `sorted()`: the `key` argument is not supported
- `type()`: only the one-argument form, and only on some types (e.g. numbers and named tuples)
- `zip()`

Hashing

The `hash()` built-in is supported and produces hash values for all supported hashable types with the following Python version specific behavior:

Under Python 3, hash values computed by Numba will exactly match those computed in CPython under the condition that the `sys.hash_info.algorithm` is `siphash24` (default).

The `PYTHONHASHSEED` environment variable influences the hashing behavior in precisely the manner described in the CPython documentation.

2.6.4 Standard library modules

array

Limited support for the `array.array` type is provided through the buffer protocol. Indexing, iteration and taking the len() is supported. All type codes are supported except for `"u"`.

cmath

The following functions from the `cmath` module are supported:

- `cmath.acos()`
- `cmath.acosh()`
- `cmath.asin()`
- `cmath.asinh()`
- `cmath.atan()`
- `cmath.atanh()`
- `cmath.cos()`
- `cmath.cosh()`
- `cmath.exp()`
- `cmath.isfinite()`
- `cmath.isinf()`

- *cmath.isnan()*
- *cmath.log()*
- *cmath.log10()*
- *cmath.phase()*
- *cmath.polar()*
- *cmath.rect()*
- *cmath.sin()*
- *cmath.sinh()*
- *cmath.sqrt()*
- *cmath.tan()*
- *cmath.tanh()*

collections

Named tuple classes, as returned by `collections.namedtuple()`, are supported in the same way regular tuples are supported. Attribute access and named parameters in the constructor are also supported.

Creating a named tuple class inside Numba code is *not* supported; the class must be created at the global level.

ctypes

Numba is able to call ctypes-declared functions with the following argument and return types:

- `ctypes.c_int8`
- `ctypes.c_int16`
- `ctypes.c_int32`
- `ctypes.c_int64`
- `ctypes.c_uint8`
- `ctypes.c_uint16`
- `ctypes.c_uint32`
- `ctypes.c_uint64`
- `ctypes.c_float`
- `ctypes.c_double`
- `ctypes.c_void_p`

enum

Both `enum.Enum` and `enum.IntEnum` subclasses are supported.

math

The following functions from the `math` module are supported:

- *math.acos()*
- *math.acosh()*
- *math.asin()*
- *math.asinh()*
- *math.atan()*
- *math.atan2()*
- *math.atanh()*
- *math.ceil()*
- *math.copysign()*
- *math.cos()*
- *math.cosh()*
- *math.degrees()*
- *math.erf()*
- *math.erfc()*
- *math.exp()*
- *math.expm1()*
- *math.fabs()*
- *math.floor()*
- *math.frexp()*
- *math.gamma()*
- *math.gcd()*
- *math.hypot()*
- *math.isfinite()*
- *math.isinf()*
- *math.isnan()*
- *math.ldexp()*
- *math.lgamma()*
- *math.log()*
- *math.log10()*
- *math.log1p()*
- *math.pow()*
- *math.radians()*
- *math.sin()*
- *math.sinh()*

- *math.sqrt()*
- *math.tan()*
- *math.tanh()*
- *math.trunc()*

operator

The following functions from the operator module are supported:

- operator.add()
- operator.and_()
- operator.eq()
- operator.floordiv()
- operator.ge()
- operator.gt()
- operator.iadd()
- operator.iand()
- operator.ifloordiv()
- operator.ilshift()
- operator.imatmul() (Python 3.5 and above)
- operator.imod()
- operator.imul()
- operator.invert()
- operator.ior()
- operator.ipow()
- operator.irshift()
- operator.isub()
- operator.itruediv()
- operator.ixor()
- operator.le()
- operator.lshift()
- operator.lt()
- operator.matmul() (Python 3.5 and above)
- operator.mod()
- operator.mul()
- operator.ne()
- operator.neg()
- operator.not_()

- `operator.or_()`
- `operator.pos()`
- `operator.pow()`
- `operator.rshift()`
- `operator.sub()`
- `operator.truediv()`
- `operator.xor()`

functools

The `functools.reduce()` function is supported but the *initializer* argument is required.

random

Numba supports top-level functions from the `random` module, but does not allow you to create individual Random instances. A Mersenne-Twister generator is used, with a dedicated internal state. It is initialized at startup with entropy drawn from the operating system.

- `random.betavariate()`
- `random.expovariate()`
- `random.gammavariate()`
- `random.gauss()`
- `random.getrandbits()`: number of bits must not be greater than 64
- `random.lognormvariate()`
- `random.normalvariate()`
- `random.paretovariate()`
- `random.randint()`
- `random.random()`
- `random.randrange()`
- `random.seed()`: with an integer argument only
- `random.shuffle()`: the sequence argument must be a one-dimension Numpy array or buffer-providing object (such as a `bytearray` or `array.array`); the second (optional) argument is not supported
- `random.uniform()`
- `random.triangular()`
- `random.vonmisesvariate()`
- `random.weibullvariate()`

Note: Calling `random.seed()` from non-Numba code (or from *object mode* code) will seed the Python random generator, not the Numba random generator.

Note: Since version 0.28.0, the generator is thread-safe and fork-safe. Each thread and each process will produce independent streams of random numbers.

See also:

Numba also supports most additional distributions from the *Numpy random module*.

`heapq`

The following functions from the `heapq` module are supported:

- `heapq.heapify()`
- `heapq.heappop()`
- `heapq.heappush()`
- `heapq.heappushpop()`
- `heapq.heapreplace()`
- `heapq.nlargest()` : first two arguments only
- `heapq.nsmallest()` : first two arguments only

Note: the heap must be seeded with at least one value to allow its type to be inferred; heap items are assumed to be homogeneous in type.

2.6.5 Third-party modules

`cffi`

Similarly to ctypes, Numba is able to call into cffi-declared external functions, using the following C types and any derived pointer types:

- `char`
- `short`
- `int`
- `long`
- `long long`
- `unsigned char`
- `unsigned short`
- `unsigned int`
- `unsigned long`
- `unsigned long long`
- `int8_t`
- `uint8_t`
- `int16_t`
- `uint16_t`

- `int32_t`

- `uint32_t`

- `int64_t`

- `uint64_t`

- `float`

- `double`

- `ssize_t`

- `size_t`

- `void`

The `from_buffer()` method of `cffi.FFI` and `CompiledFFI` objects is supported for passing Numpy arrays and other buffer-like objects. Only *contiguous* arguments are accepted. The argument to `from_buffer()` is converted to a raw pointer of the appropriate C type (for example a `double *` for a `float64` array).

Additional type mappings for the conversion from a buffer to the appropriate C type may be registered with Numba. This may include struct types, though it is only permitted to call functions that accept pointers to structs - passing a struct by value is unsupported. For registering a mapping, use:

`numba.core.typing.cffi_utils.`**`register_type`**(*cffi_type*, *numba_type*)

Out-of-line cffi modules must be registered with Numba prior to the use of any of their functions from within Numba-compiled functions:

`numba.core.typing.cffi_utils.`**`register_module`**(*mod*)
    Register the cffi out-of-line module `mod` with Numba.

Inline cffi modules require no registration.

2.7 Supported NumPy features

One objective of Numba is having a seamless integration with NumPy. NumPy arrays provide an efficient storage method for homogeneous sets of data. NumPy dtypes provide type information useful when compiling, and the regular, structured storage of potentially large amounts of data in memory provides an ideal memory layout for code generation. Numba excels at generating code that executes on top of NumPy arrays.

NumPy support in Numba comes in many forms:

- Numba understands calls to NumPy ufuncs and is able to generate equivalent native code for many of them.

- NumPy arrays are directly supported in Numba. Access to Numpy arrays is very efficient, as indexing is lowered to direct memory accesses when possible.

- Numba is able to generate ufuncs and gufuncs. This means that it is possible to implement ufuncs and gufuncs within Python, getting speeds comparable to that of ufuncs/gufuncs implemented in C extension modules using the NumPy C API.

The following sections focus on the Numpy features supported in *nopython mode*, unless otherwise stated.

2.7.1 Scalar types

Numba supports the following Numpy scalar types:

- **Integers**: all integers of either signedness, and any width up to 64 bits

- **Booleans**

- **Real numbers:** single-precision (32-bit) and double-precision (64-bit) reals

- **Complex numbers:** single-precision (2x32-bit) and double-precision (2x64-bit) complex numbers

- **Datetimes and timestamps:** of any unit

- **Character sequences** (but no operations are available on them)

- **Structured scalars:** structured scalars made of any of the types above and arrays of the types above

The following scalar types and features are not supported:

- **Arbitrary Python objects**

- **Half-precision and extended-precision** real and complex numbers

- **Nested structured scalars** the fields of structured scalars may not contain other structured scalars

The operations supported on NumPy scalars are almost the same as on the equivalent built-in types such as `int` or `float`. You can use a type's constructor to convert from a different type or width. In addition you can use the `view(np.<dtype>)` method to bitcast all `int` and `float` types within the same width. However, you must define the scalar using a NumPy constructor within a jitted function. For example, the following will work:

```
>>> import numpy as np
>>> from numba import njit
>>> @njit
... def bitcast():
...     i = np.int64(-1)
...     print(i.view(np.uint64))
...
>>> bitcast()
18446744073709551615
```

Whereas the following will not work:

```
>>> import numpy as np
>>> from numba import njit
>>> @njit
... def bitcast(i):
...     print(i.view(np.uint64))
...
>>> bitcast(np.int64(-1))
---------------------------------------------------------------------------
TypingError                               Traceback (most recent call last)
    ...
TypingError: Failed in nopython mode pipeline (step: ensure IR is legal prior to
→lowering)
'view' can only be called on NumPy dtypes, try wrapping the variable with 'np.<dtype>
→()'

File "<ipython-input-3-fc40aaab84c4>", line 3:
def bitcast(i):
    print(i.view(np.uint64))
```

Structured scalars support attribute getting and setting, as well as member lookup using constant strings. Strings stored in a local or global tuple are considered constant strings and can be used for member lookup.

```
import numpy as np
from numba import njit
```

(continues on next page)

```python
arr = np.array([(1, 2)], dtype=[('a1', 'f8'), ('a2', 'f8')])
fields_gl = ('a1', 'a2')

@njit
def get_field_sum(rec):
    fields_lc = ('a1', 'a2')
    field_name1 = fields_lc[0]
    field_name2 = fields_gl[1]
    return rec[field_name1] + rec[field_name2]

get_field_sum(arr[0])   # returns 3
```

It is also possible to use local or global tuples together with `literal_unroll`:

```python
import numpy as np
from numba import njit, literal_unroll

arr = np.array([(1, 2)], dtype=[('a1', 'f8'), ('a2', 'f8')])
fields_gl = ('a1', 'a2')

@njit
def get_field_sum(rec):
    out = 0
    for f in literal_unroll(fields_gl):
        out += rec[f]
    return out

get_field_sum(arr[0])    # returns 3
```

See also:

Numpy scalars reference.

2.7.2 Array types

Numpy arrays of any of the scalar types above are supported, regardless of the shape or layout.

Array access

Arrays support normal iteration. Full basic indexing and slicing is supported. A subset of advanced indexing is also supported: only one advanced index is allowed, and it has to be a one-dimensional array (it can be combined with an arbitrary number of basic indices as well).

See also:

Numpy indexing reference.

Attributes

The following attributes of Numpy arrays are supported:

- `dtype`
- `flags`

- `flat`

- `itemsize`

- `ndim`

- `shape`

- `size`

- `strides`

- `T`

- `real`

- `imag`

The `flags` object

The object returned by the `flags` attribute supports the `contiguous`, `c_contiguous` and `f_contiguous` attributes.

The `flat` object

The object returned by the `flat` attribute supports iteration and indexing, but be careful: indexing is very slow on non-C-contiguous arrays.

The `real` and `imag` attributes

Numpy supports these attributes regardless of the dtype but Numba chooses to limit their support to avoid potential user error. For numeric dtypes, Numba follows Numpy's behavior. The `real` attribute returns a view of the real part of the complex array and it behaves as an identity function for other numeric dtypes. The `imag` attribute returns a view of the imaginary part of the complex array and it returns a zero array with the same shape and dtype for other numeric dtypes. For non-numeric dtypes, including all structured/record dtypes, using these attributes will result in a compile-time (*TypingError*) error. This behavior differs from Numpy's but it is chosen to avoid the potential confusion with field names that overlap these attributes.

Calculation

The following methods of Numpy arrays are supported in their basic form (without any optional arguments):

- `all()`

- `any()`

- `argmax()`

- `argmin()`

- `conj()`

- `conjugate()`

- `cumprod()`

- `cumsum()`

- `max()`

- `mean()`
- `min()`
- `nonzero()`
- `prod()`
- `std()`
- `take()`
- `var()`

The corresponding top-level Numpy functions (such as *`numpy.prod()`*) are similarly supported.

Other methods

The following methods of Numpy arrays are supported:

- `argsort()` (`kind` key word argument supported for values `'quicksort'` and `'mergesort'`)
- `astype()` (only the 1-argument form)
- `copy()` (without arguments)
- `dot()` (only the 1-argument form)
- `flatten()` (no order argument; 'C' order only)
- `item()` (without arguments)
- `itemset()` (only the 1-argument form)
- `ravel()` (no order argument; 'C' order only)
- `repeat()` (no axis argument)
- `reshape()` (only the 1-argument form)
- `sort()` (without arguments)
- `sum()` (with or without the `axis` and/or `dtype` arguments.)
    - `axis` only supports `integer` values.
    - If the `axis` argument is a compile-time constant, all valid values are supported. An out-of-range value will result in a `LoweringError` at compile-time.
    - If the `axis` argument is not a compile-time constant, only values from 0 to 3 are supported. An out-of-range value will result in a runtime exception.
    - All numeric `dtypes` are supported in the `dtype` parameter. `timedelta` arrays can be used as input arrays but `timedelta` is not supported as `dtype` parameter.
    - When a `dtype` is given, it determines the type of the internal accumulator. When it is not, the selection is made automatically based on the input array's `dtype`, mostly following the same rules as NumPy. However, on 64-bit Windows, Numba uses a 64-bit accumulator for integer inputs (`int64` for `int32` inputs and `uint64` for `uint32` inputs), while NumPy would use a 32-bit accumulator in those cases.
- `transpose()`
- `view()` (only the 1-argument form)

> **Warning:** Sorting may be slightly slower than Numpy's implementation.

2.7.3 Functions

Linear algebra

Basic linear algebra is supported on 1-D and 2-D contiguous arrays of floating-point and complex numbers:

- *numpy.dot()*
- *numpy.kron()* ('C' and 'F' order only)
- *numpy.outer()*
- *numpy.trace()* (only the first argument).
- *numpy.vdot()*
- On Python 3.5 and above, the matrix multiplication operator from **PEP 465** (i.e. `a @ b` where `a` and `b` are 1-D or 2-D arrays).
- *numpy.linalg.cholesky()*
- *numpy.linalg.cond()* (only non string values in `p`).
- *numpy.linalg.det()*
- *numpy.linalg.eig()* (only running with data that does not cause a domain change is supported e.g. real input -> real output, complex input -> complex output).
- *numpy.linalg.eigh()* (only the first argument).
- *numpy.linalg.eigvals()* (only running with data that does not cause a domain change is supported e.g. real input -> real output, complex input -> complex output).
- *numpy.linalg.eigvalsh()* (only the first argument).
- *numpy.linalg.inv()*
- *numpy.linalg.lstsq()*
- *numpy.linalg.matrix_power()*
- *numpy.linalg.matrix_rank()*
- *numpy.linalg.norm()* (only the 2 first arguments and only non string values in `ord`).
- *numpy.linalg.pinv()*
- *numpy.linalg.qr()* (only the first argument).
- *numpy.linalg.slogdet()*
- *numpy.linalg.solve()*
- *numpy.linalg.svd()* (only the 2 first arguments).

> **Note:** The implementation of these functions needs SciPy to be installed.

Reductions

The following reduction functions are supported:

- *numpy.diff()* (only the 2 first arguments)
- *numpy.median()* (only the first argument)
- *numpy.nancumprod()* (only the first argument, requires NumPy >= 1.12))
- *numpy.nancumsum()* (only the first argument, requires NumPy >= 1.12))
- *numpy.nanmax()* (only the first argument)
- *numpy.nanmean()* (only the first argument)
- *numpy.nanmedian()* (only the first argument)
- *numpy.nanmin()* (only the first argument)
- *numpy.nanpercentile()* (only the 2 first arguments, requires NumPy >= 1.11, complex dtypes unsupported)
- *numpy.nanquantile()* (only the 2 first arguments, requires NumPy >= 1.15, complex dtypes unsupported)
- *numpy.nanprod()* (only the first argument)
- *numpy.nanstd()* (only the first argument)
- *numpy.nansum()* (only the first argument)
- *numpy.nanvar()* (only the first argument)
- *numpy.percentile()* (only the 2 first arguments, requires NumPy >= 1.10, complex dtypes unsupported)
- *numpy.quantile()* (only the 2 first arguments, requires NumPy >= 1.15, complex dtypes unsupported)

Other functions

The following top-level functions are supported:

- *numpy.append()*
- *numpy.arange()*
- *numpy.argsort()* (kind key word argument supported for values 'quicksort' and 'mergesort')
- *numpy.argwhere()*
- *numpy.array()* (only the 2 first arguments)
- *numpy.array_equal()*
- *numpy.asarray()* (only the 2 first arguments)
- *numpy.asfortranarray()* (only the first argument)
- *numpy.atleast_1d()*
- *numpy.atleast_2d()*
- *numpy.atleast_3d()*
- *numpy.bartlett()*
- *numpy.bincount()*
- *numpy.blackman()*

- *numpy.column_stack()*
- *numpy.concatenate()*
- *numpy.convolve()* (only the 2 first arguments)
- *numpy.copy()* (only the first argument)
- *numpy.corrcoef()* (only the 3 first arguments, requires SciPy)
- *numpy.correlate()* (only the 2 first arguments)
- *numpy.count_nonzero()* (axis only supports scalar values)
- *numpy.cov()* (only the 5 first arguments)
- *numpy.cross()* (only the 2 first arguments; at least one of the input arrays should have shape[-1] == 3)
    - If shape[-1] == 2 for both inputs, please replace your *numpy.cross()* call with numba.np. extensions.cross2d().
- *numpy.delete()* (only the 2 first arguments)
- *numpy.diag()*
- *numpy.digitize()*
- *numpy.dstack()*
- numpy.dtype() (only the first argument)
- *numpy.ediff1d()*
- *numpy.empty()* (only the 2 first arguments)
- *numpy.empty_like()* (only the 2 first arguments)
- *numpy.expand_dims()*
- *numpy.extract()*
- *numpy.eye()*
- *numpy.fill_diagonal()*
- numpy.flatten() (no order argument; 'C' order only)
- *numpy.flatnonzero()*
- *numpy.flip()* (no axis argument)
- *numpy.fliplr()*
- *numpy.flipud()*
- *numpy.frombuffer()* (only the 2 first arguments)
- *numpy.full()* (only the 3 first arguments)
- *numpy.full_like()* (only the 3 first arguments)
- *numpy.hamming()*
- *numpy.hanning()*
- *numpy.histogram()* (only the 3 first arguments)
- *numpy.hstack()*
- *numpy.identity()*

- *numpy.kaiser()*
- *numpy.interp()* (only the 3 first arguments; requires NumPy >= 1.10)
- *numpy.linspace()* (only the 3-argument form)
- numpy.ndenumerate
- numpy.ndindex
- numpy.nditer (only the first argument)
- *numpy.ones()* (only the 2 first arguments)
- *numpy.ones_like()* (only the 2 first arguments)
- *numpy.partition()* (only the 2 first arguments)
- *numpy.ptp()* (only the first argument)
- *numpy.ravel()* (no order argument; 'C' order only)
- *numpy.repeat()* (no axis argument)
- *numpy.reshape()* (no order argument; 'C' order only)
- *numpy.roll()* (only the 2 first arguments; second argument shift must be an integer)
- *numpy.roots()*
- *numpy.round_()*
- *numpy.searchsorted()* (only the 3 first arguments)
- *numpy.select()* (only using homogeneous lists or tuples for the first two arguments, condlist and choicelist). Additionally, these two arguments can only contain arrays (unlike Numpy that also accepts tuples).
- *numpy.shape()*
- *numpy.sinc()*
- *numpy.sort()* (no optional arguments)
- *numpy.stack()*
- *numpy.take()* (only the 2 first arguments)
- *numpy.transpose()*
- *numpy.trapz()* (only the 3 first arguments)
- *numpy.tri()* (only the 3 first arguments; third argument k must be an integer)
- *numpy.tril()* (second argument k must be an integer)
- *numpy.tril_indices()* (all arguments must be integer)
- *numpy.tril_indices_from()* (second argument k must be an integer)
- *numpy.triu()* (second argument k must be an integer)
- *numpy.triu_indices()* (all arguments must be integer)
- *numpy.triu_indices_from()* (second argument k must be an integer)
- *numpy.unique()* (only the first argument)
- *numpy.vander()*
- *numpy.vstack()*

- *numpy.where()*
- *numpy.zeros()* (only the 2 first arguments)
- *numpy.zeros_like()* (only the 2 first arguments)

The following constructors are supported, both with a numeric input (to construct a scalar) or a sequence (to construct an array):

- numpy.bool_
- numpy.complex64
- numpy.complex128
- numpy.float32
- numpy.float64
- numpy.int8
- numpy.int16
- numpy.int32
- numpy.int64
- numpy.intc
- numpy.intp
- numpy.uint8
- numpy.uint16
- numpy.uint32
- numpy.uint64
- numpy.uintc
- numpy.uintp

The following machine parameter classes are supported, with all purely numerical attributes:

- numpy.iinfo
- numpy.finfo (machar attribute not supported)
- numpy.MachAr (with no arguments to the constructor)

Literal arrays

Neither Python nor Numba has actual array literals, but you can construct arbitrary arrays by calling *numpy. array()* on a nested tuple:

```
a = numpy.array(((a, b, c), (d, e, f)))
```

(nested lists are not yet supported by Numba)

2.7.4 Modules

`random`

Numba supports top-level functions from the numpy.random module, but does not allow you to create individual RandomState instances. The same algorithms are used as for *the standard random module* (and therefore the same notes apply), but with an independent internal state: seeding or drawing numbers from one generator won't affect the other.

The following functions are supported.

Initialization

- *numpy.random.seed()*: with an integer argument only

Simple random data

- *numpy.random.rand()*
- *numpy.random.randint()* (only the first two arguments)
- *numpy.random.randn()*
- *numpy.random.random()*
- *numpy.random.random_sample()*
- *numpy.random.ranf()*
- *numpy.random.sample()*

Permutations

- *numpy.random.choice()*: the optional *p* argument (probabilities array) is not supported
- *numpy.random.permutation()*
- *numpy.random.shuffle()*: the sequence argument must be a one-dimension Numpy array or buffer-providing object (such as a `bytearray` or `array.array`)

Distributions

> **Warning:** The *size* argument is not supported in the following functions.

- *numpy.random.beta()*
- *numpy.random.binomial()*
- *numpy.random.chisquare()*
- *numpy.random.exponential()*
- *numpy.random.f()*
- *numpy.random.gamma()*

- *numpy.random.geometric()*
- *numpy.random.gumbel()*
- *numpy.random.hypergeometric()*
- *numpy.random.laplace()*
- *numpy.random.logistic()*
- *numpy.random.lognormal()*
- *numpy.random.logseries()*
- *numpy.random.multinomial()*
- *numpy.random.negative_binomial()*
- *numpy.random.normal()*
- *numpy.random.pareto()*
- *numpy.random.poisson()*
- *numpy.random.power()*
- *numpy.random.rayleigh()*
- *numpy.random.standard_cauchy()*
- *numpy.random.standard_exponential()*
- *numpy.random.standard_gamma()*
- *numpy.random.standard_normal()*
- *numpy.random.standard_t()*
- *numpy.random.triangular()*
- *numpy.random.uniform()*
- *numpy.random.vonmises()*
- *numpy.random.wald()*
- *numpy.random.weibull()*
- *numpy.random.zipf()*

Note: Calling *numpy.random.seed()* from non-Numba code (or from *object mode* code) will seed the Numpy random generator, not the Numba random generator.

Note: Since version 0.28.0, the generator is thread-safe and fork-safe. Each thread and each process will produce independent streams of random numbers.

stride_tricks

The following function from the numpy.lib.stride_tricks module is supported:

- *as_strided()* (the *strides* argument is mandatory, the *subok* argument is not supported)

2.7.5 Standard ufuncs

One objective of Numba is having all the standard ufuncs in NumPy understood by Numba. When a supported ufunc is found when compiling a function, Numba maps the ufunc to equivalent native code. This allows the use of those ufuncs in Numba code that gets compiled in *nopython mode*.

Limitations

Right now, only a selection of the standard ufuncs work in *nopython mode*. Following is a list of the different standard ufuncs that Numba is aware of, sorted in the same way as in the NumPy documentation.

Math operations

UFUNC	MODE	
name	object mode	nopython mode
add	Yes	Yes
subtract	Yes	Yes
multiply	Yes	Yes
divide	Yes	Yes
logaddexp	Yes	Yes
logaddexp2	Yes	Yes
true_divide	Yes	Yes
floor_divide	Yes	Yes
negative	Yes	Yes
power	Yes	Yes
remainder	Yes	Yes
mod	Yes	Yes
fmod	Yes	Yes
divmod (*)	Yes	Yes
abs	Yes	Yes
absolute	Yes	Yes
fabs	Yes	Yes
rint	Yes	Yes
sign	Yes	Yes
conj	Yes	Yes
exp	Yes	Yes
exp2	Yes	Yes
log	Yes	Yes
log2	Yes	Yes
log10	Yes	Yes
expm1	Yes	Yes
log1p	Yes	Yes
sqrt	Yes	Yes
square	Yes	Yes
reciprocal	Yes	Yes
conjugate	Yes	Yes
gcd	Yes	Yes
lcm	Yes	Yes

(*) not supported on timedelta types

Trigonometric functions

UFUNC	MODE	
name	object mode	nopython mode
sin	Yes	Yes
cos	Yes	Yes
tan	Yes	Yes
arcsin	Yes	Yes
arccos	Yes	Yes
arctan	Yes	Yes
arctan2	Yes	Yes
hypot	Yes	Yes
sinh	Yes	Yes
cosh	Yes	Yes
tanh	Yes	Yes
arcsinh	Yes	Yes
arccosh	Yes	Yes
arctanh	Yes	Yes
deg2rad	Yes	Yes
rad2deg	Yes	Yes
degrees	Yes	Yes
radians	Yes	Yes

Bit-twiddling functions

UFUNC	MODE	
name	object mode	nopython mode
bitwise_and	Yes	Yes
bitwise_or	Yes	Yes
bitwise_xor	Yes	Yes
bitwise_not	Yes	Yes
invert	Yes	Yes
left_shift	Yes	Yes
right_shift	Yes	Yes

Comparison functions

UFUNC	MODE	
name	object mode	nopython mode
greater	Yes	Yes
greater_equal	Yes	Yes
less	Yes	Yes
less_equal	Yes	Yes
not_equal	Yes	Yes
equal	Yes	Yes
logical_and	Yes	Yes
logical_or	Yes	Yes
logical_xor	Yes	Yes
logical_not	Yes	Yes
maximum	Yes	Yes
minimum	Yes	Yes
fmax	Yes	Yes
fmin	Yes	Yes

Floating functions

UFUNC	MODE	
name	object mode	nopython mode
isfinite	Yes	Yes
isinf	Yes	Yes
isnan	Yes	Yes
signbit	Yes	Yes
copysign	Yes	Yes
nextafter	Yes	Yes
modf	Yes	No
ldexp	Yes (*)	Yes
frexp	Yes	No
floor	Yes	Yes
ceil	Yes	Yes
trunc	Yes	Yes
spacing	Yes	Yes

(*) not supported on windows 32 bit

Datetime functions

UFUNC	MODE	
name	object mode	nopython mode
isnat	Yes	Yes

2.8 Deviations from Python Semantics

2.8.1 Bounds Checking

By default, instead of causing an `IndexError`, accessing an out-of-bound index of an array in a Numba-compiled function will return invalid values or lead to an access violation error (it's reading from invalid memory locations). Bounds checking can be enabled on a specific function via the *boundscheck* option of the jit decorator. Additionally, the *NUMBA_BOUNDSCHECK* can be set to 0 or 1 to globally override this flag.

Note: Bounds checking will slow down typical functions so it is recommended to only use this flag for debugging purposes.

2.8.2 Exceptions and Memory Allocation

Due to limitations in the current compiler when handling exceptions, memory allocated (almost always NumPy arrays) within a function that raises an exception will **leak**. This is a known issue that will be fixed, but in the meantime, it is best to do memory allocation outside of functions that can also raise exceptions.

2.8.3 Integer width

While Python has arbitrary-sized integers, integers in Numba-compiled functions get a fixed size through *type inference* (usually, the size of a machine integer). This means that arithmetic operations can wrapround or produce undefined results or overflow.

Type inference can be overridden by an explicit type specification, if fine-grained control of integer width is desired.

See also:

Enhancement proposal 1: Changes in integer typing

2.8.4 Boolean inversion

Calling the bitwise complement operator (the ~ operator) on a Python boolean returns an integer, while the same operator on a Numpy boolean returns another boolean:

```
>>> ~True
-2
>>> ~np.bool_(True)
False
```

Numba follows the Numpy semantics.

2.8.5 Global and closure variables

In *nopython mode*, global and closure variables are *frozen* by Numba: a Numba-compiled function sees the value of those variables at the time the function was compiled. Also, it is not possible to change their values from the function.

Numba **may or may not** copy global variables referenced inside a compiled function. Small global arrays are copied for potential compiler optimization with immutability assumption. However, large global arrays are not copied to conserve memory. The definition of "small" and "large" may change.

Todo: This document needs completing.

2.9 Floating-point pitfalls

2.9.1 Precision and accuracy

For some operations, Numba may use a different algorithm than Python or Numpy. The results may not be bit-by-bit compatible. The difference should generally be small and within reasonable expectations. However, small accumulated differences might produce large differences at the end, especially if a divergent function is involved.

Math library implementations

Numba supports a variety of platforms and operating systems, each of which has its own math library implementation (referred to as `libm` from here in). The majority of math functions included in `libm` have specific requirements as set out by the IEEE 754 standard (like `sin()`, `exp()` etc.), but each implementation may have bugs. Thus, on some platforms Numba has to exercise special care in order to workaround known `libm` issues.

Another typical problem is when an operating system's `libm` function set is incomplete and needs to be supplemented by additional functions. These are provided with reference to the IEEE 754 and C99 standards and are often implemented in Numba in a manner similar to equivalent CPython functions.

Linear algebra

Numpy forces some linear algebra operations to run in double-precision mode even when a `float32` input is given. Numba will always observe the input's precision, and invoke single-precision linear algebra routines when all inputs are `float32` or `complex64`.

The implementations of the `numpy.linalg` routines in Numba only support the floating point types that are used in the LAPACK functions that provide the underlying core functionality. As a result only `float32`, `float64`, `complex64` and `complex128` types are supported. If a user has e.g. an `int32` type, an appropriate type conversion must be performed to a floating point type prior to its use in these routines. The reason for this decision is to essentially avoid having to replicate type conversion choices made in Numpy and to also encourage the user to choose the optimal floating point type for the operation they are undertaking.

Mixed-types operations

Numpy will most often return a `float64` as a result of a computation with mixed integer and floating-point operands (a typical example is the power operator `**`). Numba by contrast will select the highest precision amongst the floating-point operands, so for example `float32 ** int32` will return a `float32`, regardless of the input values. This makes performance characteristics easier to predict, but you should explicitly cast the input to `float64` if you need the extra precision.

2.9.2 Warnings and errors

When calling a *ufunc* created with `vectorize()`, Numpy will determine whether an error occurred by examining the FPU error word. It may then print out a warning or raise an exception (such as `RuntimeWarning: divide by zero encountered`), depending on the current error handling settings.

Depending on how LLVM optimized the ufunc's code, however, some spurious warnings or errors may appear. If you get caught by this issue, we recommend you call `numpy.seterr()` to change Numpy's error handling settings, or the `numpy.errstate` context manager to switch them temporarily:

```python
with np.errstate(all='ignore'):
    x = my_ufunc(y)
```

2.10 Python 2.7 End of Life Plan

As per PEP 373, Python 2.7 will cease to be supported in 2020. Like many projects, the Numba team has to consider how to time its own end of Python 2.7 support. Given how deeply Numba must interface with the Python interpreter, supporting both Python 2 and 3 creates quite a development and testing burden. In addition, Numba (specifically via llvmlite) has to deal with some tricky compiler issues on Windows, where LLVM requires Visual Studio 2015 or later, but Python 2.7 extensions must be built with Visual Studio 2008. Needless to say, the goal with this plan is to support our Python 2.7 user base (~13% of conda package downloads and 25-30% of PyPI downloads as of July 2019), but also clearly signal that *now is the time to switch to Python 3 if you have not already*.

Python 2.7 users of Numba should also be aware of NumPy's timeline for ending Python 2.7 support. Due to Numba's tight coupling with NumPy, the NumPy timeline has informed the Numba timeline below.

2.10.1 Timeline

This timeline was revised in August 2019 to reflect the reality that we wanted to get more improvements into Numba before ending Python 2.7 support. Note that the Numba versions mentioned below are more concrete than the dates.

The end of Python 2.7 support in Numba will be staged:

- **Numba 0.47**: (~Dec 2019) This is will be the last version of Numba to support both Python 2 and 3.

- **Numba 0.48 development cycle**: (Jan-Mar 2020) Python 2 support will be removed from `master` branch. If any critical bug fixes are discovered in the 0.47.0 release, they will be backported to create 0.47.x patch releases.

- **Numba 0.48**: (~Mar 2020) This will be the first release of Numba that is Python 3 only. We will set the metadata on the package to require Python 3.5 or later.

- After Feb 2020, the Numba core developers will stop doing patch releases to the 0.47.x series for Python 2.

If there are concerns about the above timeline, please raise an issue in our issue tracker.

2.11 Deprecation Notices

This section contains information about deprecation of behaviours, features and APIs that have become undesirable/obsolete. Any information about the schedule for their deprecation and reasoning behind the changes, along with examples, is provided. However, first is a small section on how to suppress deprecation warnings that may be raised from Numba so as to prevent warnings propagating into code that is consuming Numba.

2.11.1 Suppressing Deprecation warnings

All Numba deprecations are issued via `NumbaDeprecationWarning` or `NumbaPendingDeprecationWarning` s, to suppress the reporting of these the following code snippet can be used:

```
from numba.core.errors import NumbaDeprecationWarning, NumbaPendingDeprecationWarning
import warnings

warnings.simplefilter('ignore', category=NumbaDeprecationWarning)
warnings.simplefilter('ignore', category=NumbaPendingDeprecationWarning)
```

The `action` used above is `'ignore'`, other actions are available, see The Warnings Filter documentation for more information.

Note: It is **strongly recommended** that applications and libraries which choose to suppress these warnings should pin their Numba dependency to a suitable version because their users will no longer be aware of the coming incompatibility.

2.11.2 Deprecation of reflection for List and Set types

Reflection (*reflection*) is the jargon used in Numba to describe the process of ensuring that changes made by compiled code to arguments that are mutable Python container data types are visible in the Python interpreter when the compiled function returns. Numba has for some time supported reflection of `list` and `set` data types and it is support for this reflection that is scheduled for deprecation with view to replace with a better implementation.

Reason for deprecation

First recall that for Numba to be able to compile a function in `nopython` mode all the variables must have a concrete type ascertained through type inference. In simple cases, it is clear how to reflect changes to containers inside `nopython` mode back to the original Python containers. However, reflecting changes to complex data structures with nested container types (for example, lists of lists of integers) quickly becomes impossible to do efficiently and consistently. After a number of years of experience with this problem, it is clear that providing this behaviour is both fraught with difficulty and often leads to code which does not have good performance (all reflected data has to go through special APIs to convert the data to native formats at call time and and then back to CPython formats at return time). As a result of this, the sheer number of reported problems in the issue tracker, and how well a new approach that was taken with `typed.Dict` (typed dictionaries) has gone, the core developers have decided to deprecate the noted `reflection` behaviour.

Example(s) of the impact

At present only a warning of the upcoming change is issued. In future code such as:

```
from numba import njit

@njit
def foo(x):
    x.append(10)

a = [1, 2, 3]
foo(a)
```

will require adjustment to use a `typed.List` instance, this typed container is synonymous to the *Typed Dict*. An example of translating the above is:

```
from numba import njit
from numba.typed import List

@njit
def foo(x):
    x.append(10)

a = [1, 2, 3]
typed_a = List()
[typed_a.append(x) for x in a]
foo(typed_a)
```

For more information about `typed.List` see *Typed List*. Further usability enhancements for this feature were made in the 0.47.0 release cycle.

Schedule

This feature will be removed with respect to this schedule:

- Pending-deprecation warnings will be issued in version 0.44.0
- Deprecation warnings and replacements will be issued in version 0.52.0
- Support will be removed in version 0.53.0

Recommendations

Projects that need/rely on the deprecated behaviour should pin their dependency on Numba to a version prior to removal of this behaviour, or consider following replacement instructions that will be issued outlining how to adjust to the change.

Expected Replacement

As noted above `typed.List` will be used to permit similar functionality to reflection in the case of `list` s, a `typed.Set` will provide the equivalent for `set` (not implemented yet!). The advantages to this approach are:

- That the containers are typed means type inference has to work less hard.
- Nested containers (containers of containers of . . .) are more easily supported.
- Performance penalties currently incurred translating data to/from native formats are largely avoided.
- Numba's `typed.Dict` will be able to use these containers as values.

2.11.3 Deprecation of object mode *fall-back* behaviour when using `@jit`

The `numba.jit` decorator has for a long time followed the behaviour of first attempting to compile the decorated function in *nopython mode* and should this compilation fail it will *fall-back* and try again to compile but this time in *object mode*. It it this *fall-back* behaviour which is being deprecated, the result of which will be that `numba.jit` will by default compile in *nopython mode* and *object mode* compilation will become *opt-in* only.

Reason for deprecation

The *fall-back* has repeatedly caused confusion for users as seemingly innocuous changes in user code can lead to drastic performance changes as code which may have once compiled in *nopython mode* mode may silently switch to compiling in *object mode* e.g:

```python
from numba import jit

@jit
def foo():
    l = []
    for x in range(10):
        l.append(x)
    return l

foo()

assert foo.nopython_signatures # this was compiled in nopython mode

@jit
def bar():
    l = []
    for x in range(10):
        l.append(x)
    return reversed(l) # innocuous change, but no reversed support in nopython mode

bar()

assert not bar.nopython_signatures # this was not compiled in nopython mode
```

Another reason to remove the *fall-back* is that it is confusing for the compiler engineers developing Numba as it causes internal state problems that are really hard to debug and it makes manipulating the compiler pipelines incredibly challenging.

Further, it has long been considered best practice that the *nopython mode* keyword argument in the `numba.jit` decorator is set to `True` and that any user effort spent should go into making code work in this mode as there's very little gain if it does not. The result is that, as Numba has evolved, the amount of use *object mode* gets in practice and its general utility has decreased. It can be noted that there are some minor improvements available through the notion of *loop-lifting*, the cases of this being used in practice are, however, rare and often a legacy from use of less-recent Numba whereby such behaviour was better accommodated/the use of `@jit` with *fall-back* was recommended.

Example(s) of the impact

At present a warning of the upcoming change is issued if `@jit` decorated code uses the *fall-back* compilation path. In future code such as:

```python
@jit
def bar():
    l = []
    for x in range(10):
        l.append(x)
    return reversed(l)

bar()
```

will simply not compile, a `TypingError` would be raised.

Schedule

This feature will be removed with respect to this schedule:

- Deprecation warnings will be issued in version 0.44.0

- Support will be removed in version 0.54.0

Recommendations

Projects that need/rely on the deprecated behaviour should pin their dependency on Numba to a version prior to removal of this behaviour. Alternatively, to accommodate the scheduled deprecations, users with code compiled at present with `@jit` can supply the `nopython=True` keyword argument, if the code continues to compile then the code is already ready for this change. If the code does not compile, continue using the `@jit` decorator without `nopython=True` and profile the performance of the function. Then remove the decorator and again check the performance of the function. If there is no benefit to having the `@jit` decorator present consider removing it! If there is benefit to having the `@jit` decorator present, then to be future proof supply the keyword argument `forceobj=True` to ensure the function is always compiled in *object mode*.

2.11.4 Change of jitclass location

Between versions 0.48 and 0.49 Numba underwent a large amount of refactoring. One of the decisions made by the core developers as part of this refactoring was to move `numba.jitclass` to a new location `numba.experimental.jitclass`. This is to help reinforce expectations over the behaviour and support for certain features by deliberately placing them in an `experimental` submodule.

Example(s) of the impact

The `jitclass` decorator has historically been available via `from numba import jitclass`, any code using this import location will in future need to be updated to `from numba.experimental import jitclass`.

Recommendations

Simply update imports as follows:

- Change `from numba import jitclass` to `from numba.experimental import jitclass`

Schedule

This feature will be moved with respect to this schedule:

- Deprecation warnings will be issued in version 0.49.0

- Support for importing from `numba.jitclass` will be removed in version 0.52.0.

2.11.5 Deprecation of the target kwarg

There have been a number of users attempting to use the `target` keyword argument that's meant for internal use only. We are deprecating this argument, as alternative solutions are available to achieve the same behaviour.

Recommendations

Update the `jit` decorator as follows:

- Change `@numba.jit(..., target='cuda')` to `numba.cuda.jit(...)`.

Schedule

This feature will be moved with respect to this schedule:

- Deprecation warnings will be issued in 0.51.0.
- The target kwarg will be removed in version 0.53.0.

Numba for CUDA GPUs

3.1 Overview

Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. Kernels written in Numba appear to have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically.

3.1.1 Terminology

Several important terms in the topic of CUDA programming are listed here:

- *host*: the CPU
- *device*: the GPU
- *host memory*: the system main memory
- *device memory*: onboard memory on a GPU card
- *kernels*: a GPU function launched by the host and executed on the device
- *device function*: a GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

3.1.2 Programming model

Most CUDA programming facilities exposed by Numba map directly to the CUDA C language offered by NVidia. Therefore, it is recommended you read the official CUDA C programming guide.

3.1.3 Requirements

Supported GPUs

Numba supports CUDA-enabled GPU with compute capability 2.0 or above with an up-to-data Nvidia driver.

Software

You will need the CUDA toolkit version 8.0 or later installed. If you are using Conda, just type:

```
$ conda install cudatoolkit
```

If you are not using Conda or if you want to use a different version of CUDA toolkit, the following describe how Numba searches for a CUDA toolkit installation.

Setting CUDA Installation Path

Numba searches for a CUDA toolkit installation in the following order:

1. Conda installed *cudatoolkit* package.

2. Environment variable `CUDA_HOME`, which points to the directory of the installed CUDA toolkit (i.e. `/home/user/cuda-10`)

3. System-wide installation at exactly `/usr/local/cuda` on Linux platforms. Versioned installation paths (i.e. `/usr/local/cuda-10.0`) are intentionally ignored. Users can use `CUDA_HOME` to select specific versions.

In addition to the CUDA toolkit libraries, which can be installed by conda into an environment or installed system-wide by the CUDA SDK installer, the CUDA target in Numba also requires an up-to-date NVIDIA graphics driver. Updated graphics drivers are also installed by the CUDA SDK installer, so there is no need to do both. Note that on macOS, the CUDA SDK must be installed to get the required driver, and the driver is only supported on macOS prior to 10.14 (Mojave). If the `libcuda` library is in a non-standard location, users can set environment variable `NUMBA_CUDA_DRIVER` to the file path (not the directory path) of the shared library file.

3.1.4 Missing CUDA Features

Numba does not implement all features of CUDA, yet. Some missing features are listed below:

- dynamic parallelism
- texture memory

3.2 Writing CUDA Kernels

3.2.1 Introduction

CUDA has an execution model unlike the traditional sequential model used for programming CPUs. In CUDA, the code you write will be executed by multiple threads at once (often hundreds or thousands). Your solution will be modeled by defining a thread hierarchy of *grid*, *blocks* and *threads*.

Numba's CUDA support exposes facilities to declare and manage this hierarchy of threads. The facilities are largely similar to those exposed by NVidia's CUDA C language.

Numba also exposes three kinds of GPU memory: global *device memory* (the large, relatively slow off-chip memory that's connected to the GPU itself), on-chip *shared memory* and *local memory*. For all but the simplest algorithms, it

is important that you carefully consider how to use and access memory in order to minimize bandwidth requirements and contention.

3.2.2 Kernel declaration

A *kernel function* is a GPU function that is meant to be called from CPU code (*). It gives it two fundamental characteristics:

- kernels cannot explicitly return a value; all result data must be written to an array passed to the function (if computing a scalar, you will probably pass a one-element array);

- kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block (note that while a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes).

At first sight, writing a CUDA kernel with Numba looks very much like writing a *JIT function* for the CPU:

```python
@cuda.jit
def increment_by_one(an_array):
    """
    Increment all array elements by one.
    """
    # code elided here; read further for different implementations
```

(*) Note: newer CUDA devices support device-side kernel launching; this feature is called *dynamic parallelism* but Numba does not support it currently)

3.2.3 Kernel invocation

A kernel is typically launched in the following way:

```python
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

We notice two steps here:

- Instantiate the kernel proper, by specifying a number of blocks (or "blocks per grid"), and a number of threads per block. The product of the two will give the total number of threads launched. Kernel instantiation is done by taking the compiled kernel function (here `increment_by_one`) and indexing it with a tuple of integers.

- Running the kernel, by passing it the input array (and any separate output arrays if necessary). Kernels run asynchronously: launches queue their execution on the device and then return immediately. You can use *cuda. synchronize()* to wait for all previous kernel launches to finish executing.

Note: Passing an array that resides in host memory will implicitly cause a copy back to the host, which will be synchronous. In this case, the kernel launch will not return until the data is copied back, and therefore appears to execute synchronously.

Choosing the block size

It might seem curious to have a two-level hierarchy when declaring the number of threads needed by a kernel. The block size (i.e. number of threads per block) is often crucial:

- On the software side, the block size determines how many threads share a given area of *shared memory*.

- On the hardware side, the block size must be large enough for full occupation of execution units; recommendations can be found in the CUDA C Programming Guide.

Multi-dimensional blocks and grids

To help deal with multi-dimensional arrays, CUDA allows you to specify multi-dimensional blocks and grids. In the example above, you could make `blockspergrid` and `threadsperblock` tuples of one, two or three integers. Compared to 1D declarations of equivalent sizes, this doesn't change anything to the efficiency or behaviour of generated code, but can help you write your algorithms in a more natural way.

3.2.4 Thread positioning

When running a kernel, the kernel function's code is executed by every thread once. It therefore has to know which thread it is in, in order to know which array element(s) it is responsible for (complex algorithms may define more complex responsibilities, but the underlying principle is the same).

One way is for the thread to determine its position in the grid and block and manually compute the corresponding array position:

```
@cuda.jit
def increment_by_one(an_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] += 1
```

Note: Unless you are sure the block size and grid size is a divisor of your array size, you **must** check boundaries as shown above.

threadIdx, *blockIdx*, *blockDim* and *gridDim* are special objects provided by the CUDA backend for the sole purpose of knowing the geometry of the thread hierarchy and the position of the current thread within that geometry.

These objects can be 1D, 2D or 3D, depending on how the kernel was *invoked*. To access the value at each dimension, use the x, y and z attributes of these objects, respectively.

numba.cuda.**threadIdx**
> The thread indices in the current thread block. For 1D blocks, the index (given by the x attribute) is an integer spanning the range from 0 inclusive to *numba.cuda.blockDim* exclusive. A similar rule exists for each dimension when more than one dimension is used.

numba.cuda.**blockDim**
> The shape of the block of threads, as declared when instantiating the kernel. This value is the same for all threads in a given kernel, even if they belong to different blocks (i.e. each block is "full").

numba.cuda.**blockIdx**
> The block indices in the grid of threads launched a kernel. For a 1D grid, the index (given by the x attribute) is

an integer spanning the range from 0 inclusive to *numba.cuda.gridDim* exclusive. A similar rule exists for each dimension when more than one dimension is used.

numba.cuda.**gridDim**

> The shape of the grid of blocks, i.e. the total number of blocks launched by this kernel invocation, as declared when instantiating the kernel.

Absolute positions

Simple algorithms will tend to always use thread indices in the same way as shown in the example above. Numba provides additional facilities to automate such calculations:

numba.cuda.**grid**(*ndim*)

> Return the absolute position of the current thread in the entire grid of blocks. *ndim* should correspond to the number of dimensions declared when instantiating the kernel. If *ndim* is 1, a single integer is returned. If *ndim* is 2 or 3, a tuple of the given number of integers is returned.

numba.cuda.**gridsize**(*ndim*)

> Return the absolute size (or shape) in threads of the entire grid of blocks. *ndim* has the same meaning as in *grid()* above.

With these functions, the incrementation example can become:

```
@cuda.jit
def increment_by_one(an_array):
    pos = cuda.grid(1)
    if pos < an_array.size:
        an_array[pos] += 1
```

The same example for a 2D array and grid of threads would be:

```
@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
        an_array[x, y] += 1
```

Note the grid computation when instantiating the kernel must still be done manually, for example:

```
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(an_array.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(an_array.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
increment_a_2D_array[blockspergrid, threadsperblock](an_array)
```

Further Reading

Please refer to the the CUDA C Programming Guide for a detailed discussion of CUDA programming.

3.3 Memory management

3.3.1 Data transfer

Even though Numba can automatically transfer NumPy arrays to the device, it can only do so conservatively by always transferring device memory back to the host when a kernel finishes. To avoid the unnecessary transfer for read-only arrays, you can use the following APIs to manually control the transfer:

numba.cuda.**device_array**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*, *stream=0*)
> Allocate an empty device ndarray. Similar to [`numpy.empty()`](#).

numba.cuda.**device_array_like**(*ary*, *stream=0*)
> Call cuda.devicearray() with information from the array.

numba.cuda.**to_device**(*obj*, *stream=0*, *copy=True*, *to=None*)
> Allocate and transfer a numpy ndarray or structured scalar to the device.

> To copy host->device a numpy array:

```
ary = np.arange(10)
d_ary = cuda.to_device(ary)
```

> To enqueue the transfer to a stream:

```
stream = cuda.stream()
d_ary = cuda.to_device(ary, stream=stream)
```

> The resulting `d_ary` is a `DeviceNDArray`.

> To copy device->host:

```
hary = d_ary.copy_to_host()
```

> To copy device->host to an existing array:

```
ary = np.empty(shape=d_ary.shape, dtype=d_ary.dtype)
d_ary.copy_to_host(ary)
```

> To enqueue the transfer to a stream:

```
hary = d_ary.copy_to_host(stream=stream)
```

In addition to the device arrays, Numba can consume any object that implements *cuda array interface*. These objects also can be manually converted into a Numba device array by creating a view of the GPU buffer using the following APIs:

numba.cuda.**as_cuda_array**(*obj*)
> Create a DeviceNDArray from any object that implements the *cuda array interface*.

> A view of the underlying GPU buffer is created. No copying of the data is done. The resulting DeviceNDArray will acquire a reference from *obj*.

numba.cuda.**is_cuda_array**(*obj*)
> Test if the object has defined the *__cuda_array_interface__* attribute.

> Does not verify the validity of the interface.

Device arrays

Device array references have the following methods. These methods are to be called in host code, not within CUDA-jitted functions.

class `numba.cuda.cudadrv.devicearray.`**`DeviceNDArray`**(*shape,* *strides,* *dtype,* *stream=0,* *writeback=None,* *gpu_data=None*)

An on-GPU array type

`copy_to_host`(*ary=None*, *stream=0*)

Copy `self` to `ary` or create a new Numpy ndarray if `ary` is `None`.

If a CUDA `stream` is given, then the transfer will be made asynchronously as part as the given stream. Otherwise, the transfer is synchronous: the function returns after the copy is finished.

Always returns the host array.

Example:

```python
import numpy as np
from numba import cuda

arr = np.arange(1000)
d_arr = cuda.to_device(arr)

my_kernel[100, 100](d_arr)

result_array = d_arr.copy_to_host()
```

`is_c_contiguous`()

Return true if the array is C-contiguous.

`is_f_contiguous`()

Return true if the array is Fortran-contiguous.

`ravel`(*order='C'*, *stream=0*)

Flatten the array without changing its contents, similar to `numpy.ndarray.ravel()`.

`reshape`(**newshape*, ***kws*)

Reshape the array without changing its contents, similarly to `numpy.ndarray.reshape()`. Example:

```python
d_arr = d_arr.reshape(20, 50, order='F')
```

Note: DeviceNDArray defines the *cuda array interface*.

3.3.2 Pinned memory

`numba.cuda.`**`pinned`**(**arylist*)

A context manager for temporary pinning a sequence of host ndarrays.

`numba.cuda.`**`pinned_array`**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*)

Allocate a np.ndarray with a buffer that is pinned (pagelocked). Similar to np.empty().

3.3.3 Streams

Streams can be passed to functions that accept them (e.g. copies between the host and device) and into kernel launch configurations so that the operations are executed asynchronously.

numba.cuda.**stream**()
> Create a CUDA stream that represents a command queue for the device.

numba.cuda.**default_stream**()
> Get the default CUDA stream. CUDA semantics in general are that the default stream is either the legacy default stream or the per-thread default stream depending on which CUDA APIs are in use. In Numba, the APIs for the legacy default stream are always the ones in use, but an option to use APIs for the per-thread default stream may be provided in future.

numba.cuda.**legacy_default_stream**()
> Get the legacy default CUDA stream.

numba.cuda.**per_thread_default_stream**()
> Get the per-thread default CUDA stream.

numba.cuda.**external_stream**(*ptr*)
> Create a Numba stream object for a stream allocated outside Numba.
>
> > **Parameters ptr** (*int*) – Pointer to the external stream to wrap in a Numba Stream

CUDA streams have the following methods:

class numba.cuda.cudadrv.driver.**Stream**(*context*, *handle*, *finalizer*, *external=False*)

> **auto_synchronize**()
> > A context manager that waits for all commands in this stream to execute and commits any pending memory transfers upon exiting the context.
>
> **synchronize**()
> > Wait for all commands in this stream to execute. This will commit any pending memory transfers.

3.3.4 Shared memory and thread synchronization

A limited amount of shared memory can be allocated on the device to speed up access to data, when necessary. That memory will be shared (i.e. both readable and writable) amongst all threads belonging to a given block and has faster access times than regular device memory. It also allows threads to cooperate on a given solution. You can think of it as a manually-managed data cache.

The memory is allocated once for the duration of the kernel, unlike traditional dynamic memory management.

numba.cuda.shared.**array**(*shape*, *type*)
> Allocate a shared array of the given *shape* and *type* on the device. This function must be called on the device (i.e. from a kernel or device function). *shape* is either an integer or a tuple of integers representing the array's dimensions and must be a simple constant expression. *type* is a *Numba type* of the elements needing to be stored in the array.
>
> The returned array-like object can be read and written to like any normal device array (e.g. through indexing).
>
> A common pattern is to have each thread populate one element in the shared array and then wait for all threads to finish using *syncthreads()*.

numba.cuda.**syncthreads**()
> Synchronize all threads in the same thread block. This function implements the same pattern as barriers in traditional multi-threaded programming: this function waits until all threads in the block call it, at which point it returns control to all its callers.

See also:

Matrix multiplication example.

3.3.5 Local memory

Local memory is an area of memory private to each thread. Using local memory helps allocate some scratchpad area when scalar local variables are not enough. The memory is allocated once for the duration of the kernel, unlike traditional dynamic memory management.

`numba.cuda.local.`**`array`**(*shape*, *type*)
> Allocate a local array of the given *shape* and *type* on the device. *shape* is either an integer or a tuple of integers representing the array's dimensions and must be a simple constant expression. *type* is a *Numba type* of the elements needing to be stored in the array. The array is private to the current thread. An array-like object is returned which can be read and written to like any standard array (e.g. through indexing).

3.3.6 Constant memory

Constant memory is an area of memory that is read only, cached and off-chip, it is accessible by all threads and is host allocated. A method of creating an array in constant memory is through the use of:

`numba.cuda.const.`**`array_like`**(*arr*)
> Allocate and make accessible an array in constant memory based on array-like *arr*.

3.3.7 Deallocation Behavior

This section describes the deallocation behaviour of Numba's internal memory management. If an External Memory Management Plugin is in use (see *External Memory Management (EMM) Plugin interface*), then deallocation behaviour may differ; you may refer to the documentation for the EMM Plugin to understand its deallocation behaviour.

Deallocation of all CUDA resources are tracked on a per-context basis. When the last reference to a device memory is dropped, the underlying memory is scheduled to be deallocated. The deallocation does not occur immediately. It is added to a queue of pending deallocations. This design has two benefits:

1. Resource deallocation API may cause the device to synchronize; thus, breaking any asynchronous execution. Deferring the deallocation could avoid latency in performance critical code section.

2. Some deallocation errors may cause all the remaining deallocations to fail. Continued deallocation errors can cause critical errors at the CUDA driver level. In some cases, this could mean a segmentation fault in the CUDA driver. In the worst case, this could cause the system GUI to freeze and could only recover with a system reset. When an error occurs during a deallocation, the remaining pending deallocations are cancelled. Any deallocation error will be reported. When the process is terminated, the CUDA driver is able to release all allocated resources by the terminated process.

The deallocation queue is flushed automatically as soon as the following events occur:

- An allocation failed due to out-of-memory error. Allocation is retried after flushing all deallocations.

- The deallocation queue has reached its maximum size, which is default to 10. User can override by setting the environment variable *NUMBA_CUDA_MAX_PENDING_DEALLOCS_COUNT*. For example, *NUMBA_CUDA_MAX_PENDING_DEALLOCS_COUNT=20*, increases the limit to 20.

- The maximum accumulated byte size of resources that are pending deallocation is reached. This is default to 20% of the device memory capacity. User can override by setting the environment variable *NUMBA_CUDA_MAX_PENDING_DEALLOCS_RATIO*. For example, *NUMBA_CUDA_MAX_PENDING_DEALLOCS_RATIO=0.5* sets the limit to 50% of the capacity.

Sometimes, it is desired to defer resource deallocation until a code section ends. Most often, users want to avoid any implicit synchronization due to deallocation. This can be done by using the following context manager:

numba.cuda.**defer_cleanup**()

>   Temporarily disable memory deallocation. Use this to prevent resource deallocation breaking asynchronous execution.
>
>   For example:

```
with defer_cleanup():
    # all cleanup is deferred in here
    do_speed_critical_code()
# cleanup can occur here
```

>   Note: this context manager can be nested.

3.4 Writing Device Functions

CUDA device functions can only be invoked from within the device (by a kernel or another device function). To define a device function:

```
from numba import cuda

@cuda.jit(device=True)
def a_device_function(a, b):
    return a + b
```

Unlike a kernel function, a device function can return a value like normal functions.

3.5 Supported Python features in CUDA Python

This page lists the Python features supported in the CUDA Python. This includes all kernel and device functions compiled with @cuda.jit and other higher level Numba decorators that targets the CUDA GPU.

3.5.1 Language

Execution Model

CUDA Python maps directly to the *single-instruction multiple-thread* execution (SIMT) model of CUDA. Each instruction is implicitly executed by multiple threads in parallel. With this execution model, array expressions are less useful because we don't want multiple threads to perform the same task. Instead, we want threads to perform a task in a cooperative fashion.

For details please consult the CUDA Programming Guide.

Constructs

The following Python constructs are not supported:

- Exception handling (try .. except, try .. finally)
- Context management (the with statement)

- Comprehensions (either list, dict, set or generator comprehensions)
- Generator (any `yield` statements)

The `raise` statement is supported.

The `assert` statement is supported, but only has an effect when `debug=True` is passed to the `numba.cuda.jit()` decorator. This is similar to the behavior of the `assert` keyword in CUDA C/C++, which is ignored unless compiling with device debug turned on.

Printing of strings, integers, and floats is supported, but printing is an asynchronous operation - in order to ensure that all output is printed after a kernel launch, it is necessary to call `numba.cuda.synchronize()`. Eliding the call to `synchronize` is acceptable, but output from a kernel may appear during other later driver operations (e.g. subsequent kernel launches, memory transfers, etc.), or fail to appear before the program execution completes.

3.5.2 Built-in types

The following built-in types support are inherited from CPU nopython mode.

- int
- float
- complex
- bool
- None
- tuple

See *nopython built-in types*.

3.5.3 Built-in functions

The following built-in functions are supported:

- `abs()`
- `bool`
- `complex`
- `enumerate()`
- `float`
- `int`: only the one-argument form
- `len()`
- `min()`: only the multiple-argument form
- `max()`: only the multiple-argument form
- `range`
- `round()`
- `zip()`

3.5.4 Standard library modules

cmath

The following functions from the `cmath` module are supported:

- *cmath.acos()*
- *cmath.acosh()*
- *cmath.asin()*
- *cmath.asinh()*
- *cmath.atan()*
- *cmath.atanh()*
- *cmath.cos()*
- *cmath.cosh()*
- *cmath.exp()*
- *cmath.isfinite()*
- *cmath.isinf()*
- *cmath.isnan()*
- *cmath.log()*
- *cmath.log10()*
- *cmath.phase()*
- *cmath.polar()*
- *cmath.rect()*
- *cmath.sin()*
- *cmath.sinh()*
- *cmath.sqrt()*
- *cmath.tan()*
- *cmath.tanh()*

math

The following functions from the `math` module are supported:

- *math.acos()*
- *math.asin()*
- *math.atan()*
- *math.acosh()*
- *math.asinh()*
- *math.atanh()*
- *math.cos()*

- *math.sin()*
- *math.tan()*
- *math.hypot()*
- *math.cosh()*
- *math.sinh()*
- *math.tanh()*
- *math.atan2()*
- *math.erf()*
- *math.erfc()*
- *math.exp()*
- *math.expm1()*
- *math.fabs()*
- *math.gamma()*
- *math.lgamma()*
- *math.log()*
- *math.log10()*
- *math.log1p()*
- *math.sqrt()*
- *math.pow()*
- *math.ceil()*
- *math.floor()*
- *math.copysign()*
- math.fmod()
- math.modf()
- *math.isnan()*
- *math.isinf()*

operator

The following functions from the `operator` module are supported:

- `operator.add()`
- `operator.and_()`
- `operator.eq()`
- `operator.floordiv()`
- `operator.ge()`
- `operator.gt()`
- `operator.iadd()`

- `operator.iand()`
- `operator.ifloordiv()`
- `operator.ilshift()`
- `operator.imod()`
- `operator.imul()`
- `operator.invert()`
- `operator.ior()`
- `operator.ipow()`
- `operator.irshift()`
- `operator.isub()`
- `operator.itruediv()`
- `operator.ixor()`
- `operator.le()`
- `operator.lshift()`
- `operator.lt()`
- `operator.mod()`
- `operator.mul()`
- `operator.ne()`
- `operator.neg()`
- `operator.not_()`
- `operator.or_()`
- `operator.pos()`
- `operator.pow()`
- `operator.rshift()`
- `operator.sub()`
- `operator.truediv()`
- `operator.xor()`

3.5.5 Numpy support

Due to the CUDA programming model, dynamic memory allocation inside a kernel is inefficient and is often not needed. Numba disallows any memory allocating features. This disables a large number of NumPy APIs. For best performance, users should write code such that each thread is dealing with a single element at a time.

Supported numpy features:

- accessing *ndarray* attributes *.shape*, *.strides*, *.ndim*, *.size*, etc..
- scalar ufuncs that have equivalents in the *math* module; i.e. `np.sin(x[0])`, where x is a 1D array.
- indexing and slicing works.

Unsupported numpy features:

- array creation APIs.

- array methods.

- functions that returns a new array.

3.6 Supported Atomic Operations

Numba provides access to some of the atomic operations supported in CUDA. Those that are presently implemented
are as follows:

class numba.cuda.**atomic**
> Namespace for atomic operations

> **class add**(*ary*, *idx*, *val*)
> > Perform atomic ary[idx] += val. Supported on int32, float32, and float64 operands only.

> > Returns the old value at the index location as if it is loaded atomically.

> **class compare_and_swap**(*ary*, *old*, *val*)
> > Conditionally assign val to the first element of an 1D array ary if the current value matches old.

> > Returns the current value as if it is loaded atomically.

> **class max**(*ary*, *idx*, *val*)
> > Perform atomic ary[idx] = max(ary[idx], val).

> > Supported on int32, int64, uint32, uint64, float32, float64 operands only.

> > Returns the old value at the index location as if it is loaded atomically.

> **class min**(*ary*, *idx*, *val*)
> > Perform atomic ary[idx] = min(ary[idx], val).

> > Supported on int32, int64, uint32, uint64, float32, float64 operands only.

> > Returns the old value at the index location as if it is loaded atomically.

> **class nanmax**(*ary*, *idx*, *val*)
> > Perform atomic ary[idx] = max(ary[idx], val).

> > NOTE: NaN is treated as a missing value such that: nanmax(NaN, n) == n, nanmax(n, NaN) == n

> > Supported on int32, int64, uint32, uint64, float32, float64 operands only.

> > Returns the old value at the index location as if it is loaded atomically.

> **class nanmin**(*ary*, *idx*, *val*)
> > Perform atomic ary[idx] = min(ary[idx], val).

> > NOTE: NaN is treated as a missing value, such that: nanmin(NaN, n) == n, nanmin(n, NaN) == n

> > Supported on int32, int64, uint32, uint64, float32, float64 operands only.

> > Returns the old value at the index location as if it is loaded atomically.

3.6.1 Example

The following code demonstrates the use of *numba.cuda.atomic.max* to find the maximum value in an array.
Note that this is not the most efficient way of finding a maximum in this case, but that it serves as an example:

```
from numba import cuda
import numpy as np

@cuda.jit
def max_example(result, values):
    """Find the maximum value in values and store in result[0]"""
    tid = cuda.threadIdx.x
    bid = cuda.blockIdx.x
    bdim = cuda.blockDim.x
    i = (bid * bdim) + tid
    cuda.atomic.max(result, 0, values[i])


arr = np.random.rand(16384)
result = np.zeros(1, dtype=np.float64)

max_example[256,64](result, arr)
print(result[0]) # Found using cuda.atomic.max
print(max(arr))  # Print max(arr) for comparison (should be equal!)
```

Multiple dimension arrays are supported by using a tuple of ints for the index:

```
@cuda.jit
def max_example_3d(result, values):
    """
    Find the maximum value in values and store in result[0].
    Both result and values are 3d arrays.
    """
    i, j, k = cuda.grid(3)
    # Atomically store to result[0,1,2] from values[i, j, k]
    cuda.atomic.max(result, (0, 1, 2), values[i, j, k])

arr = np.random.rand(1000).reshape(10,10,10)
result = np.zeros((3, 3, 3), dtype=np.float64)
max_example_3d[(2, 2, 2), (5, 5, 5)](result, arr)
print(result[0, 1, 2], '==', np.max(arr))
```

3.7 Random Number Generation

Numba provides a random number generation algorithm that can be executed on the GPU. Due to technical issues with how NVIDIA implemented cuRAND, however, Numba's GPU random number generator is not based on cuRAND. Instead, Numba's GPU RNG is an implementation of the xoroshiro128+ algorithm. The xoroshiro128+ algorithm has a period of $2**128 - 1$, which is shorter than the period of the XORWOW algorithm used by default in cuRAND, but xoroshiro128+ still passes the BigCrush tests of random number generator quality.

When using any RNG on the GPU, it is important to make sure that each thread has its own RNG state, and they have been initialized to produce non-overlapping sequences. The numba.cuda.random module provides a host function to do this, as well as CUDA device functions to obtain uniformly or normally distributed random numbers.

Note: Numba (like cuRAND) uses the *Box-Muller transform <https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform>* to generate normally distributed random numbers from a uniform generator. However, Box-Muller generates pairs of random numbers, and the current implementation only returns one of them. As a result, generating normally distributed values is half the speed of uniformly distributed values.

---

numba.cuda.random.**create_xoroshiro128p_states**(*n*, *seed*, *subsequence_start=0*, *stream=0*)

    Returns a new device array initialized for n random number generators.

    This initializes the RNG states so that each state in the array corresponds subsequences in the separated by 2**64 steps from each other in the main sequence. Therefore, as long no CUDA thread requests more than 2**64 random numbers, all of the RNG states produced by this function are guaranteed to be independent.

    The subsequence_start parameter can be used to advance the first RNG state by a multiple of 2**64 steps.

> **Parameters**
>
> - **n** (`int`) – number of RNG states to create
> - **seed** (`uint64`) – starting seed for list of generators
> - **subsequence_start** (`uint64`) –
> - **stream** (`CUDA stream`) – stream to run initialization kernel on

numba.cuda.random.**init_xoroshiro128p_states**(*states*, *seed*, *subsequence_start=0*, *stream=0*)

    Initialize RNG states on the GPU for parallel generators.

    This initializes the RNG states so that each state in the array corresponds subsequences in the separated by 2**64 steps from each other in the main sequence. Therefore, as long no CUDA thread requests more than 2**64 random numbers, all of the RNG states produced by this function are guaranteed to be independent.

    The subsequence_start parameter can be used to advance the first RNG state by a multiple of 2**64 steps.

> **Parameters**
>
> - **states** (`1D DeviceNDArray, dtype=xoroshiro128p_dtype`) – array of RNG states
> - **seed** (`uint64`) – starting seed for list of generators

numba.cuda.random.**xoroshiro128p_uniform_float32**

    Return a float32 in range [0.0, 1.0) and advance `states[index]`.

> **Parameters**
>
> - **states** (`1D array, dtype=xoroshiro128p_dtype`) – array of RNG states
> - **index** (`int64`) – offset in states to update
>
> **Return type** float32

numba.cuda.random.**xoroshiro128p_uniform_float64**

    Return a float64 in range [0.0, 1.0) and advance `states[index]`.

> **Parameters**
>
> - **states** (`1D array, dtype=xoroshiro128p_dtype`) – array of RNG states
> - **index** (`int64`) – offset in states to update
>
> **Return type** float64

numba.cuda.random.**xoroshiro128p_normal_float32**

    Return a normally distributed float32 and advance `states[index]`.

    The return value is drawn from a Gaussian of mean=0 and sigma=1 using the Box-Muller transform. This advances the RNG sequence by two steps.

> **Parameters**
>
> - **states** (`1D array, dtype=xoroshiro128p_dtype`) – array of RNG states

- **index** (*int64*) – offset in states to update

> **Return type** float32

numba.cuda.random.**xoroshiro128p_normal_float64**

> Return a normally distributed float32 and advance states[index].

> The return value is drawn from a Gaussian of mean=0 and sigma=1 using the Box-Muller transform. This advances the RNG sequence by two steps.

> > **Parameters**
> >
> > - **states** (*1D array, dtype=xoroshiro128p_dtype*) – array of RNG states
> >
> > - **index** (*int64*) – offset in states to update
> >
> > **Return type** float64

3.7.1 Example

Here is a sample program that uses the random number generator:

```python
from __future__ import print_function, absolute_import

from numba import cuda
from numba.cuda.random import create_xoroshiro128p_states, xoroshiro128p_uniform_
→float32
import numpy as np

@cuda.jit
def compute_pi(rng_states, iterations, out):
    """Find the maximum value in values and store in result[0]"""
    thread_id = cuda.grid(1)

    # Compute pi by drawing random (x, y) points and finding what
    # fraction lie inside a unit circle
    inside = 0
    for i in range(iterations):
        x = xoroshiro128p_uniform_float32(rng_states, thread_id)
        y = xoroshiro128p_uniform_float32(rng_states, thread_id)
        if x**2 + y**2 <= 1.0:
            inside += 1

    out[thread_id] = 4.0 * inside / iterations

threads_per_block = 64
blocks = 24
rng_states = create_xoroshiro128p_states(threads_per_block * blocks, seed=1)
out = np.zeros(threads_per_block * blocks, dtype=np.float32)

compute_pi[blocks, threads_per_block](rng_states, 10000, out)
print('pi:', out.mean())
```

3.8 Device management

For multi-GPU machines, users may want to select which GPU to use. By default the CUDA driver selects the fastest GPU as the device 0, which is the default device used by Numba.

The features introduced on this page are generally not of interest unless working with systems hosting/offering more than one CUDA-capable GPU.

3.8.1 Device Selection

If at all required, device selection must be done before any CUDA feature is used.

```python
from numba import cuda
cuda.select_device(0)
```

The device can be closed by:

```python
cuda.close()
```

Users can then create a new context with another device.

```python
cuda.select_device(1)  # assuming we have 2 GPUs
```

numba.cuda.**select_device**(*device_id*)

> Create a new CUDA context for the selected *device_id*. *device_id* should be the number of the device (starting from 0; the device order is determined by the CUDA libraries). The context is associated with the current thread. Numba currently allows only one context per thread.
>
> If successful, this function returns a device instance.

numba.cuda.**close**()

> Explicitly close all contexts in the current thread.

Note: Compiled functions are associated with the CUDA context. This makes it not very useful to close and create new devices, though it is certainly useful for choosing which device to use when the machine has multiple GPUs.

3.9 The Device List

The Device List is a list of all the GPUs in the system, and can be indexed to obtain a context manager that ensures execution on the selected GPU.

numba.cuda.**gpus**

numba.cuda.cudadrv.devices.**gpus**

numba.cuda.gpus is an instance of the _DeviceList class, from which the current GPU context can also be retrieved:

class numba.cuda.cudadrv.devices.**_DeviceList**

> **current**
>> Returns the active device or None if there's no active device

3.10 Examples

3.10.1 Matrix multiplication

Here is a naive implementation of matrix multiplication using a CUDA kernel:

```python
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

This implementation is straightforward and intuitive but performs poorly, because the same matrix elements will be loaded multiple times from device memory, which is slow (some devices may have transparent data caches, but they may not be large enough to hold the entire inputs at once).

It will be faster if we use a blocked algorithm to reduce accesses to the device memory. CUDA provides a fast *shared memory* for threads in a block to cooperately compute on a task. The following implements a faster version of the square matrix multiplication using shared memory:

```python
from numba import cuda, float32

# Controls threads per block and shared memory usage.
# The computation will be done on blocks of TPBxTPB elements.
TPB = 16

@cuda.jit
def fast_matmul(A, B, C):
    # Define an array in the shared memory
    # The size and type of the arrays must be known at compile time
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bpg = cuda.gridDim.x    # blocks per grid

    if x >= C.shape[0] and y >= C.shape[1]:
        # Quit if (x, y) is outside of valid C boundary
        return

    # Each thread computes one element in the result matrix.
    # The dot product is chunked into dot products of TPB-long vectors.
    tmp = 0.
    for i in range(bpg):
        # Preload data into shared memory
        sA[tx, ty] = A[x, ty + i * TPB]
        sB[tx, ty] = B[tx + i * TPB, y]

        # Wait until all threads finish preloading
```

(continues on next page)

```
        cuda.syncthreads()

        # Computes partial product on the shared memory
        for j in range(TPB):
            tmp += sA[tx, j] * sB[j, ty]

        # Wait until all threads finish computing
        cuda.syncthreads()

    C[x, y] = tmp
```

Because the shared memory is a limited resources, the code preloads small block at a time from the input arrays. Then, it calls *syncthreads()* to wait until all threads have finished preloading and before doing the computation on the shared memory. It synchronizes again after the computation to ensure all threads have finished with the data in shared memory before overwriting it in the next loop iteration.

3.11 Debugging CUDA Python with the the CUDA Simulator

Numba includes a CUDA Simulator that implements most of the semantics in CUDA Python using the Python interpreter and some additional Python code. This can be used to debug CUDA Python code, either by adding print statements to your code, or by using the debugger to step through the execution of an individual thread.

Execution of kernels is performed by the simulator one block at a time. One thread is spawned for each thread in the block, and scheduling of the execution of these threads is left up to the operating system.

3.11.1 Using the simulator

The simulator is enabled by setting the environment variable *NUMBA_ENABLE_CUDASIM* to 1. CUDA Python code may then be executed as normal. The easiest way to use the debugger inside a kernel is to only stop a single thread, otherwise the interaction with the debugger is difficult to handle. For example, the kernel below will stop in the thread `<<<(3,0,0), (1, 0, 0)>>>`:

```
@cuda.jit
def vec_add(A, B, out):
    x = cuda.threadIdx.x
    bx = cuda.blockIdx.x
    bdx = cuda.blockDim.x
    if x == 1 and bx == 3:
        from pdb import set_trace; set_trace()
    i = bx * bdx + x
    out[i] = A[i] + B[i]
```

when invoked with a one-dimensional grid and one-dimensional blocks.

3.11.2 Supported features

The simulator aims to provide as complete a simulation of execution on a real GPU as possible - in particular, the following are supported:

- Atomic operations
- Constant memory

- Local memory

- Shared memory: declarations of shared memory arrays must be on separate source lines, since the simulator uses source line information to keep track of allocations of shared memory across threads.

- *syncthreads()* is supported - however, in the case where divergent threads enter different *syncthreads()* calls, the launch will not fail, but unexpected behaviour will occur. A future version of the simulator may detect this condition.

- The stream API is supported, but all operations occur sequentially and synchronously, unlike on a real device. Synchronising on a stream is therefore a no-op.

- The event API is also supported, but provides no meaningful timing information.

- Data transfer to and from the GPU - in particular, creating array objects with *device_array()* and *device_array_like()*. The APIs for pinned memory *pinned()* and *pinned_array()* are also supported, but no pinning takes place.

- The driver API implementation of the list of GPU contexts (`cuda.gpus` and `cuda.cudadrv.devices.gpus`) is supported, and reports a single GPU context. This context can be closed and reset as the real one would.

- The *detect()* function is supported, and reports one device called *SIMULATOR*.

Some limitations of the simulator include:

- It does not perform type checking/type inference. If any argument types to a jitted function are incorrect, or if the specification of the type of any local variables are incorrect, this will not be detected by the simulator.

- Only one GPU is simulated.

- Multithreaded accesses to a single GPU are not supported, and will result in unexpected behaviour.

- Most of the driver API is unimplemented.

- It is not possible to link PTX code with CUDA Python functions.

- Warps and warp-level operations are not yet implemented.

Obviously, the speed of the simulator is also much lower than that of a real device. It may be necessary to reduce the size of input data and the size of the CUDA grid in order to make debugging with the simulator tractable.

3.12 GPU Reduction

Writing a reduction algorithm for CUDA GPU can be tricky. Numba provides a `@reduce` decorator for converting a simple binary operation into a reduction kernel. An example follows:

```python
import numpy
from numba import cuda

@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(1234, dtype=numpy.float64)) + 1
expect = A.sum()      # numpy sum reduction
got = sum_reduce(A)   # cuda sum reduction
assert expect == got
```

Lambda functions can also be used here:

```
sum_reduce = cuda.reduce(lambda a, b: a + b)
```

3.12.1 The Reduce class

The `reduce` decorator creates an instance of the `Reduce` class. Currently, `reduce` is an alias to `Reduce`, but this behavior is not guaranteed.

class numba.cuda.**Reduce**(*functor*)

Create a reduction object that reduces values using a given binary function. The binary function is compiled once and cached inside this object. Keeping this object alive will prevent re-compilation.

> **__init__**(*functor*)
>
> > **Parameters functor** – A function implementing a binary operation for reduction. It will be compiled as a CUDA device function using `cuda.jit(device=True)`.
>
> **__call__**(*arr*, *size=None*, *res=None*, *init=0*, *stream=0*)
>
> Performs a full reduction.
>
> > **Parameters**
> >
> > - **arr** – A host or device array.
> >
> > - **size** – Optional integer specifying the number of elements in `arr` to reduce. If this parameter is not specified, the entire array is reduced.
> >
> > - **res** – Optional device array into which to write the reduction result to. The result is written into the first element of this array. If this parameter is specified, then no communication of the reduction output takes place from the device to the host.
> >
> > - **init** – Optional initial value for the reduction, the type of which must match `arr.dtype`.
> >
> > - **stream** – Optional CUDA stream in which to perform the reduction. If no stream is specified, the default stream of 0 is used.
> >
> > **Returns** If `res` is specified, `None` is returned. Otherwise, the result of the reduction is returned.

3.13 CUDA Ufuncs and Generalized Ufuncs

This page describes the CUDA ufunc-like object.

To support the programming pattern of CUDA programs, CUDA Vectorize and GUVectorize cannot produce a conventional ufunc. Instead, a ufunc-like object is returned. This object is a close analog but not fully compatible with a regular NumPy ufunc. The CUDA ufunc adds support for passing intra-device arrays (already on the GPU device) to reduce traffic over the PCI-express bus. It also accepts a *stream* keyword for launching in asynchronous mode.

3.13.1 Example: Basic Example

```
import math
from numba import vectorize, cuda
import numpy as np

@vectorize(['float32(float32, float32, float32)',
            'float64(float64, float64, float64)'],
```

```
            target='cuda')
def cu_discriminant(a, b, c):
    return math.sqrt(b ** 2 - 4 * a * c)


N = 10000
dtype = np.float32

# prepare the input
A = np.array(np.random.sample(N), dtype=dtype)
B = np.array(np.random.sample(N) + 10, dtype=dtype)
C = np.array(np.random.sample(N), dtype=dtype)

D = cu_discriminant(A, B, C)

print(D)  # print result
```

3.13.2 Example: Calling Device Functions

All CUDA ufunc kernels have the ability to call other CUDA device functions:

```
from numba import vectorize, cuda

# define a device function
@cuda.jit('float32(float32, float32, float32)', device=True, inline=True)
def cu_device_fn(x, y, z):
    return x ** y / z

# define a ufunc that calls our device function
@vectorize(['float32(float32, float32, float32)'], target='cuda')
def cu_ufunc(x, y, z):
    return cu_device_fn(x, y, z)
```

3.13.3 Generalized CUDA ufuncs

Generalized ufuncs may be executed on the GPU using CUDA, analogous to the CUDA ufunc functionality. This may be accomplished as follows:

```
from numba import guvectorize

@guvectorize(['void(float32[:,:], float32[:,:], float32[:,:])'],
             '(m,n),(n,p)->(m,p)', target='cuda')
def matmulcore(A, B, C):
    ...
```

There are times when the gufunc kernel uses too many of a GPU's resources, which can cause the kernel launch to fail. The user can explicitly control the maximum size of the thread block by setting the *max_blocksize* attribute on the compiled gufunc object.

```
from numba import guvectorize

@guvectorize(..., target='cuda')
def very_complex_kernel(A, B, C):
    ...
```

```
very_complex_kernel.max_blocksize = 32  # limits to 32 threads per block
```

3.14 Sharing CUDA Memory

3.14.1 Sharing between process

> **Warning:** This feature is limited to Linux only.

Export device array to another process

A device array can be shared with another process in the same machine using the CUDA IPC API. To do so, use the `.get_ipc_handle()` method on the device array to get a `IpcArrayHandle` object, which can be transferred to another process.

DeviceNDArray.**get_ipc_handle**()

> Returns a *IpcArrayHandle* object that is safe to serialize and transfer to another process to share the local allocation.
>
> Note: this feature is only available on Linux.

class numba.cuda.cudadrv.devicearray.**IpcArrayHandle**(*ipc_handle*, *array_desc*)

> An IPC array handle that can be serialized and transfer to another process in the same machine for share a GPU allocation.
>
> On the destination process, use the *.open()* method to creates a new *DeviceNDArray* object that shares the allocation from the original process. To release the resources, call the *.close()* method. After that, the destination can no longer use the shared array object. (Note: the underlying weakref to the resource is now dead.)
>
> This object implements the context-manager interface that calls the *.open()* and *.close()* method automatically:
>
> ```
> with the_ipc_array_handle as ipc_array:
>     # use ipc_array here as a normal gpu array object
>     some_code(ipc_array)
> # ipc_array is dead at this point
> ```
>
> **close**()
>
> > Closes the IPC handle to the array.
>
> **open**()
>
> > Returns a new *DeviceNDArray* that shares the allocation from the original process. Must not be used on the original process.

Import IPC memory from another process

The following function is used to open IPC handle from another process as a device array.

cuda.**open_ipc_array**(*shape*, *dtype*, *strides=None*, *offset=0*)

> A context manager that opens a IPC *handle* (*CUipcMemHandle*) that is represented as a sequence of bytes (e.g. *bytes*, tuple of int) and represent it as an array of the given *shape*, *strides* and *dtype*. The *strides* can be omitted. In that case, it is assumed to be a 1D C contiguous array.

Yields a device array.

The IPC handle is closed automatically when context manager exits.

3.15 CUDA Array Interface (Version 2)

The *cuda array interface* is created for interoperability between different implementation of GPU array-like objects in various projects. The idea is borrowed from the numpy array interface.

Note: Currently, we only define the Python-side interface. In the future, we may add a C-side interface for efficient exchange of the information in compiled code.

3.15.1 Python Interface Specification

Note: Experimental feature. Specification may change.

The `__cuda_array_interface__` attribute returns a dictionary (`dict`) that must contain the following entries:

- **shape**: `(integer, ...)`

  A tuple of `int` (or `long`) representing the size of each dimension.

- **typestr**: `str`

  The type string. This has the same definition as `typestr` in the numpy array interface.

- **data**: `(integer, boolean)`

  The **data** is a 2-tuple. The first element is the data pointer as a Python `int` (or `long`). The data must be device-accessible. For zero-size arrays, use `0` here. The second element is the read-only flag as a Python `bool`.

  Because the user of the interface may or may not be in the same context, the most common case is to use `cuPointerGetAttribute` with `CU_POINTER_ATTRIBUTE_DEVICE_POINTER` in the CUDA driver API (or the equivalent CUDA Runtime API) to retrieve a device pointer that is usable in the currently active context.

- **version**: `integer`

  An integer for the version of the interface being exported. The current version is *2*.

The following are optional entries:

- **strides**: `None` or `(integer, ...)`

  If **strides** is not given, or it is `None`, the array is in C-contiguous layout. Otherwise, a tuple of `int` (or `long`) is explicitly given for representing the number of bytes to skip to access the next element at each dimension.

- **descr**

  This is for describing more complicated types. This follows the same specification as in the numpy array interface.

- **mask**: `None` or object exposing the `__cuda_array_interface__`

  If `None` then all values in **data** are valid. All elements of the mask array should be interpreted only as true or not true indicating which elements of this array are valid. This has the same definition as `mask` in the numpy array interface.

Note: Numba does not currently support working with masked CUDA arrays and will raise a `NotImplementedError` exception if one is passed to a GPU function.

Lifetime management

Obtaining the value of the `__cuda_array_interface__` property of any object has no effect on the lifetime of the object from which it was created. In particular, note that the interface has no slot for the owner of the data.

It is therefore imperative for a consumer to retain a reference to the object owning the data for as long as they make use of the data.

Lifetime management in Numba

Numba provides two mechanisms for creating device arrays. Which to use depends on whether the created device array should maintain the life of the object from which it is created:

- `as_cuda_array`: This creates a device array that holds a reference to the owning object. As long as a reference to the device array is held, its underlying data will also be kept alive, even if all other references to the original owning object have been dropped.

- `from_cuda_array_interface`: This creates a device array with no reference to the owning object by default. The owning object, or some other object to be considered the owner can be passed in the `owner` parameter.

The interfaces of these functions are:

cuda.**as_cuda_array**()
> Create a DeviceNDArray from any object that implements the *cuda array interface*.

> A view of the underlying GPU buffer is created. No copying of the data is done. The resulting DeviceNDArray will acquire a reference from *obj*.

cuda.**from_cuda_array_interface**(*owner=None*)
> Create a DeviceNDArray from a cuda-array-interface description. The *owner* is the owner of the underlying memory. The resulting DeviceNDArray will acquire a reference from it.

Pointer Attributes

Additional information about the data pointer can be retrieved using `cuPointerGetAttribute` or `cudaPointerGetAttributes`. Such information include:

- the CUDA context that owns the pointer;
- is the pointer host-accessible?
- is the pointer a managed memory?

Differences with CUDA Array Interface (Version 0)

Version 0 of the CUDA Array Interface did not have the optional **mask** attribute to support masked arrays.

Differences with CUDA Array Interface (Version 1)

Versions 0 and 1 of the CUDA Array Interface neither clarified the **strides** attribute for C-contiguous arrays nor specified the treatment for zero-size arrays.

Interoperability

The following Python libraries have adopted the CUDA Array Interface:

- Numba
- CuPy
- PyTorch
- PyArrow
- mpi4py
- ArrayViews
- JAX
- The RAPIDS stack:
    - cuDF
    - cuML
    - cuSignal
    - RMM

If your project is not on this list, please feel free to report it on the Numba issue tracker.

3.16 External Memory Management (EMM) Plugin interface

The *CUDA Array Interface* enables sharing of data between different Python libraries that access CUDA devices. However, each library manages its own memory distinctly from the others. For example:

- By default, Numba allocates memory on CUDA devices by interacting with the CUDA driver API to call functions such as `cuMemAlloc` and `cuMemFree`, which is suitable for many use cases.
- The RAPIDS libraries (cuDF, cuML, etc.) use the RAPIDS Memory Manager (RMM) for allocating device memory.
- CuPy includes a memory pool implementation for both device and pinned memory.

When multiple CUDA-aware libraries are used together, it may be preferable for Numba to defer to another library for memory management. The EMM Plugin interface facilitates this, by enabling Numba to use another CUDA-aware library for all allocations and deallocations.

An EMM Plugin is used to facilitate the use of an external library for memory management. An EMM Plugin can be a part of an external library, or could be implemented as a separate library.

3.16.1 Overview of External Memory Management

When an EMM Plugin is in use (see *Setting the EMM Plugin*), Numba will make memory allocations and deallocations through the Plugin. It will never directly call functions such as `cuMemAlloc`, `cuMemFree`, etc.

EMM Plugins always take responsibility for the management of device memory. However, not all CUDA-aware libraries also support managing host memory, so a facility for Numba to continue the management of host memory whilst ceding control of device memory to the EMM is provided (see *The Host-Only CUDA Memory Manager*).

Effects on Deallocation Strategies

Numba's internal *Deallocation Behavior* is designed to increase efficiency by deferring deallocations until a significant quantity are pending. It also provides a mechanism for preventing deallocations entirely during critical sections, using the `defer_cleanup()` context manager.

When an EMM Plugin is in use, the deallocation strategy is implemented by the EMM, and Numba's internal deallocation mechanism is not used. The EMM Plugin could implement:

- A similar strategy to the Numba deallocation behaviour, or

- Something more appropriate to the plugin - for example, deallocated memory might immediately be returned to a memory pool.

The `defer_cleanup` context manager may behave differently with an EMM Plugin - an EMM Plugin should be accompanied by documentation of the behaviour of the `defer_cleanup` context manager when it is in use. For example, a pool allocator could always immediately return memory to a pool even when the context manager is in use, but could choose not to free empty pools until `defer_cleanup` is not in use.

Management of other objects

In addition to memory, Numba manages the allocation and deallocation of *events*, *streams*, and modules (a module is a compiled object, which is generated from `@cuda.jit`-ted functions). The management of events, streams, and modules is unchanged by the use of an EMM Plugin.

Asynchronous allocation and deallocation

The present EMM Plugin interface does not provide support for asynchronous allocation and deallocation. This may be added to a future version of the interface.

3.16.2 Implementing an EMM Plugin

An EMM Plugin is implemented by deriving from *BaseCUDAMemoryManager*. A summary of considerations for the implementation follows:

- Numba instantiates one instance of the EMM Plugin class per context. The context that owns an EMM Plugin object is accessible through `self.context`, if required.

- The EMM Plugin is transparent to any code that uses Numba - all its methods are invoked by Numba, and never need to be called by code that uses Numba.

- The allocation methods `memalloc`, `memhostalloc`, and `mempin`, should use the underlying library to allocate and/or pin device or host memory, and construct an instance of a *memory pointer* representing the memory to return back to Numba. These methods are always called when the current CUDA context is the context that owns the EMM Plugin instance.

- The `initialize` method is called by Numba prior to the first use of the EMM Plugin object for a context. This method should do anything required to prepare the underlying library for allocations in the current context. This method may be called multiple times, and must not invalidate previous state when it is called.

- The `reset` method is called when all allocations in the context are to be cleaned up. It may be called even prior to `initialize`, and an EMM Plugin implementation needs to guard against this.

- To support inter-GPU communication, the `get_ipc_handle` method should provide an *IpcHandle* for a given *MemoryPointer* instance. This method is part of the EMM interface (rather than being handled within Numba) because the base address of the allocation is only known by the underlying library. Closing an IPC handle is handled internally within Numba.

- It is optional to provide memory info from the `get_memory_info` method, which provides a count of the total and free memory on the device for the context. It is preferable to implement the method, but this may not be practical for all allocators. If memory info is not provided, this method should raise a `RuntimeError`.

- The `defer_cleanup` method should return a context manager that ensures that expensive cleanup operations are avoided whilst it is active. The nuances of this will vary between plugins, so the plugin documentation should include an explanation of how deferring cleanup affects deallocations, and performance in general.

- The `interface_version` property is used to ensure that the plugin version matches the interface provided by the version of Numba. At present, this should always be 1.

Full documentation for the base class follows:

class numba.cuda.**BaseCUDAMemoryManager**(**args*, ***kwargs*)
> Abstract base class for External Memory Management (EMM) Plugins.

> **memalloc**(*size*)
> > Allocate on-device memory in the current context.
> >
> > > **Parameters** **size** (*int*) – Size of allocation in bytes
> > >
> > > **Returns** A memory pointer instance that owns the allocated memory
> > >
> > > **Return type** *MemoryPointer*

> **memhostalloc**(*size*, *mapped*, *portable*, *wc*)
> > Allocate pinned host memory.
> >
> > > **Parameters**
> > >
> > > - **size** (*int*) – Size of the allocation in bytes
> > >
> > > - **mapped** (*bool*) – Whether the allocated memory should be mapped into the CUDA address space.
> > >
> > > - **portable** (*bool*) – Whether the memory will be considered pinned by all contexts, and not just the calling context.
> > >
> > > - **wc** (*bool*) – Whether to allocate the memory as write-combined.
> > >
> > > **Returns** A memory pointer instance that owns the allocated memory. The return type depends on whether the region was mapped into device memory.
> > >
> > > **Return type** *MappedMemory* or *PinnedMemory*

> **mempin**(*owner*, *pointer*, *size*, *mapped*)
> > Pin a region of host memory that is already allocated.
> >
> > > **Parameters**
> > >
> > > - **owner** – The object that owns the memory.
> > >
> > > - **pointer** (*int*) – The pointer to the beginning of the region to pin.
> > >
> > > - **size** (*int*) – The size of the region in bytes.
> > >
> > > - **mapped** (*bool*) – Whether the region should also be mapped into device memory.
> > >
> > > **Returns** A memory pointer instance that refers to the allocated memory.
> > >
> > > **Return type** *MappedMemory* or *PinnedMemory*

initialize()
> Perform any initialization required for the EMM plugin instance to be ready to use.
>
> > **Returns** None

get_ipc_handle(*memory*)
> Return an IPC handle from a GPU allocation.
>
> > **Parameters** **memory** (*MemoryPointer*) – Memory for which the IPC handle should be created.
> >
> > **Returns** IPC handle for the allocation
> >
> > **Return type** *IpcHandle*

get_memory_info()
> Returns (`free`, `total`) memory in bytes in the context. May raise `NotImplementedError`, if returning such information is not practical (e.g. for a pool allocator).
>
> > **Returns** Memory info
> >
> > **Return type** *MemoryInfo*

reset()
> Clears up all memory allocated in this context.
>
> > **Returns** None

defer_cleanup()
> Returns a context manager that ensures the implementation of deferred cleanup whilst it is active.
>
> > **Returns** Context manager

interface_version
> Returns an integer specifying the version of the EMM Plugin interface supported by the plugin implementation. Should always return 1 for implementations of this version of the specification.

The Host-Only CUDA Memory Manager

Some external memory managers will support management of on-device memory but not host memory. For implementing EMM Plugins using one of these memory managers, a partial implementation of a plugin that implements host-side allocation and pinning is provided. To use it, derive from *HostOnlyCUDAMemoryManager* instead of *BaseCUDAMemoryManager*. Guidelines for using this class are:

* The host-only memory manager implements `memhostalloc` and `mempin` - the EMM Plugin should still implement `memalloc`.

* If `reset` is overridden, it must also call `super().reset()` to allow the host allocations to be cleaned up.

* If `defer_cleanup` is overridden, it must hold an active context manager from `super().defer_cleanup()` to ensure that host-side cleanup is also deferred.

Documentation for the methods of *HostOnlyCUDAMemoryManager* follows:

class numba.cuda.**HostOnlyCUDAMemoryManager**(**args*, ***kwargs*)
> Base class for External Memory Management (EMM) Plugins that only implement on-device allocation. A subclass need not implement the `memhostalloc` and `mempin` methods.
>
> This class also implements `reset` and `defer_cleanup` (see *numba.cuda.* *BaseCUDAMemoryManager*) for its own internal state management. If an EMM Plugin based on this class also implements these methods, then its implementations of these must also call the method from `super()` to give HostOnlyCUDAMemoryManager an opportunity to do the necessary work for the host allocations it is managing.

> This class does not implement `interface_version`, as it will always be consistent with the version of Numba in which it is implemented. An EMM Plugin subclassing this class should implement `interface_version` instead.

> **memhostalloc**(*size*, *mapped=False*, *portable=False*, *wc=False*)
>> Implements the allocation of pinned host memory.
>>
>> It is recommended that this method is not overridden by EMM Plugin implementations - instead, use the *BaseCUDAMemoryManager*.

> **mempin**(*owner*, *pointer*, *size*, *mapped=False*)
>> Implements the pinning of host memory.
>>
>> It is recommended that this method is not overridden by EMM Plugin implementations - instead, use the *BaseCUDAMemoryManager*.

> **reset**()
>> Clears up all host memory (mapped and/or pinned) in the current context.
>>
>> EMM Plugins that override this method must call `super().reset()` to ensure that host allocations are also cleaned up.

> **defer_cleanup**()
>> Returns a context manager that disables cleanup of mapped or pinned host memory in the current context whilst it is active.
>>
>> EMM Plugins that override this method must obtain the context manager from this method before yielding to ensure that cleanup of host allocations is also deferred.

The IPC Handle Mixin

An implementation of the `get_ipc_handle()` function is is provided in the `GetIpcHandleMixin` class. This uses the driver API to determine the base address of an allocation for opening an IPC handle. If this implementation is appropriate for an EMM plugin, it can be added by mixing in the `GetIpcHandleMixin` class:

class numba.cuda.**GetIpcHandleMixin**
> A class that provides a default implementation of `get_ipc_handle()`.

> **get_ipc_handle**(*memory*)
>> Open an IPC memory handle by using `cuMemGetAddressRange` to determine the base pointer of the allocation. An IPC handle of type `cu_ipc_mem_handle` is constructed and initialized with `cuIpcGetMemHandle`. A *numba.cuda.IpcHandle* is returned, populated with the underlying `ipc_mem_handle`.

3.16.3 Classes and structures of returned objects

This section provides an overview of the classes and structures that need to be constructed by an EMM Plugin.

Memory Pointers

EMM Plugins should construct memory pointer instances that represent their allocations, for return to Numba. The appropriate memory pointer class to use in each method is:

- *MemoryPointer*: returned from `memalloc`
- *MappedMemory*: returned from `memhostalloc` or `mempin` when the host memory is mapped into the device memory space.

- *PinnedMemory*: return from `memhostalloc` or `mempin` when the host memory is not mapped into the device memory space.

Memory pointers can take a finalizer, which is a function that is called when the buffer is no longer needed. Usually the finalizer will make a call to the memory management library (either internal to Numba, or external if allocated by an EMM Plugin) to inform it that the memory is no longer required, and that it could potentially be freed and/or unpinned. The memory manager may choose to defer actually cleaning up the memory to any later time after the finalizer runs - it is not required to free the buffer immediately.

Documentation for the memory pointer classes follows.

class numba.cuda.**MemoryPointer**(*context*, *pointer*, *size*, *owner=None*, *finalizer=None*)
> A memory pointer that owns a buffer, with an optional finalizer. Memory pointers provide reference counting, and instances are initialized with a reference count of 1.
>
> The base `MemoryPointer` class does not use the reference count for managing the buffer lifetime. Instead, the buffer lifetime is tied to the memory pointer instance's lifetime:
>
> - When the instance is deleted, the finalizer will be called.
> - When the reference count drops to 0, no action is taken.
>
> Subclasses of `MemoryPointer` may modify these semantics, for example to tie the buffer lifetime to the reference count, so that the buffer is freed when there are no more references.
>
> > **Parameters**
> >
> > - **context** (`Context`) – The context in which the pointer was allocated.
> > - **pointer** (*ctypes.c_void_p*) – The address of the buffer.
> > - **size** (*int*) – The size of the allocation in bytes.
> > - **owner** (*NoneType*) – The owner is sometimes set by the internals of this class, or used for Numba's internal memory management. It should not be provided by an external user of the `MemoryPointer` class (e.g. from within an EMM Plugin); the default of *None* should always suffice.
> > - **finalizer** (*function*) – A function that is called when the buffer is to be freed.

The `AutoFreePointer` class need not be used directly, but is documented here as it is subclassed by *numba. cuda.MappedMemory*:

class numba.cuda.cudadrv.driver.**AutoFreePointer**(**args*, ***kwargs*)
> Modifies the ownership semantic of the MemoryPointer so that the instance lifetime is directly tied to the number of references.
>
> When the reference count reaches zero, the finalizer is invoked.
>
> Constructor arguments are the same as for `MemoryPointer`.

class numba.cuda.**MappedMemory**(*context*, *pointer*, *size*, *owner=None*, *finalizer=None*)
> A memory pointer that owns a buffer on the host that is mapped into device memory.
>
> > **Parameters**
> >
> > - **context** (`Context`) – The context in which the pointer was mapped.
> > - **pointer** (*ctypes.c_void_p*) – The address of the buffer.
> > - **size** (*int*) – The size of the buffer in bytes.
> > - **owner** (*NoneType*) – The owner is sometimes set by the internals of this class, or used for Numba's internal memory management. It should not be provided by an external user of

the `MappedMemory` class (e.g. from within an EMM Plugin); the default of *None* should always suffice.

- **finalizer** (*function*) – A function that is called when the buffer is to be freed.

class numba.cuda.**PinnedMemory**(*context*, *pointer*, *size*, *owner=None*, *finalizer=None*)
> A pointer to a pinned buffer on the host.

> **Parameters**

>> - **context** ([Context](#)) – The context in which the pointer was mapped.
>> - **owner** – The object owning the memory. For EMM plugin implementation, this ca
>> - **pointer** ([ctypes.c_void_p](#)) – The address of the buffer.
>> - **size** ([int](#)) – The size of the buffer in bytes.
>> - **owner** – An object owning the buffer that has been pinned. For EMM plugin implementation, the default of `None` suffices for memory allocated in `memhostalloc` - for `mempin`, it should be the owner passed in to the `mempin` method.
>> - **finalizer** (*function*) – A function that is called when the buffer is to be freed.

Memory Info

If an implementation of [get_memory_info()](#) is to provide a result, then it should return an instance of the `MemoryInfo` named tuple:

class numba.cuda.**MemoryInfo**(*free*, *total*)
> Free and total memory for a device.

> **free**
>> Free device memory in bytes.

> **total**
>> Total device memory in bytes.

IPC

An instance of `IpcHandle` is required to be returned from an implementation of [get_ipc_handle()](#):

class numba.cuda.**IpcHandle**(*base*, *handle*, *size*, *source_info=None*, *offset=0*)
> CUDA IPC handle. Serialization of the CUDA IPC handle object is implemented here.

> **Parameters**

>> - **base** ([MemoryPointer](#)) – A reference to the original allocation to keep it alive
>> - **handle** – The CUDA IPC handle, as a ctypes array of bytes.
>> - **size** ([int](#)) – Size of the original allocation
>> - **source_info** ([dict](#)) – The identity of the device on which the IPC handle was opened.
>> - **offset** ([int](#)) – The offset into the underlying allocation of the memory referred to by this IPC handle.

Guidance for constructing an IPC handle in the context of implementing an EMM Plugin:

- The `memory` parameter passed to the `get_ipc_handle` method of an EMM Plugin can be passed as the `base` parameter.

- A suitable type for the `handle` can be constructed as `ctypes.c_byte * 64`. The data for `handle` must be populated using a method for obtaining a CUDA IPC handle appropriate to the underlying library.

- `size` should match the size of the original allocation, which can be obtained with `memory.size` in `get_ipc_handle`.

- An appropriate value for `source_info` can be created by calling `self.context.device.get_device_identity()`.

- If the underlying memory does not point to the base of an allocation returned by the CUDA driver or runtime API (e.g. if a pool allocator is in use) then the `offset` from the base must be provided.

3.16.4 Setting the EMM Plugin

By default, Numba uses its internal memory management - if an EMM Plugin is to be used, it must be configured. There are two mechanisms for configuring the use of an EMM Plugin: an environment variable, and a function.

Environment variable

A module name can be provided in the environment variable, `NUMBA_CUDA_MEMORY_MANAGER`. If this environment variable is set, Numba will attempt to import the module, and and use its `_numba_memory_manager` global variable as the memory manager class. This is primarily useful for running the Numba test suite with an EMM Plugin, e.g.:

```
$ NUMBA_CUDA_MEMORY_MANAGER=rmm python -m numba.runtests numba.cuda.tests
```

Function

The *set_memory_manager()* function can be used to set the memory manager at runtime. This should be called prior to the initialization of any contexts, as EMM Plugin instances are instantiated along with contexts.

numba.cuda.**set_memory_manager**(*mm_plugin*)
> Configure Numba to use an External Memory Management (EMM) Plugin. If the EMM Plugin version does not match one supported by this version of Numba, a RuntimeError will be raised.

> > **Parameters** **mm_plugin** (`BaseCUDAMemoryManager`) – The class implementing the EMM Plugin.

> > **Returns** None

Resetting the memory manager

It is recommended that the memory manager is set once prior to using any CUDA functionality, and left unchanged for the remainder of execution. It is possible to set the memory manager multiple times, noting the following:

- At the time of their creation, contexts are bound to an instance of a memory manager for their lifetime.

- Changing the memory manager will have no effect on existing contexts - only contexts created after the memory manager was updated will use instances of the new memory manager.

- *numba.cuda.close()* can be used to destroy contexts after setting the memory manager so that they get re-created with the new memory manager.

> - This will invalidate any arrays, streams, events, and modules owned by the context.

- Attempting to use invalid arrays, streams, or events will likely fail with an exception being raised due to a `CUDA_ERROR_INVALID_CONTEXT` or `CUDA_ERROR_CONTEXT_IS_DESTROYED` return code from a Driver API function.

- Attempting to use an invalid module will result in similar, or in some cases a segmentation fault / access violation.

Note: The invalidation of modules means that all functions compiled with `@cuda.jit` prior to context destruction will need to be redefined, as the code underlying them will also have been unloaded from the GPU.

3.17 CUDA Frequently Asked Questions

3.17.1 nvprof reports "No kernels were profiled"

When using the `nvprof` tool to profile Numba jitted code for the CUDA target, the output contains `No kernels were profiled` but there are clearly running kernels present, what is going on?

This is quite likely due to the profiling data not being flushed on program exit, see the NVIDIA CUDA documentation for details. To fix this simply add a call to `numba.cuda.profile_stop()` prior to the exit point in your program (or wherever you want to stop profiling). For more on CUDA profiling support in Numba, see *Profiling*.

CUDA Python Reference

4.1 CUDA Host API

4.1.1 Device Management

Device detection and enquiry

The following functions are available for querying the available hardware:

numba.cuda.**is_available**()
> Returns a boolean to indicate the availability of a CUDA GPU.
>
> This will initialize the driver if it hasn't been initialized.

numba.cuda.**detect**()
> Detect supported CUDA hardware and print a summary of the detected hardware.
>
> Returns a boolean indicating whether any supported devices were detected.

Context management

CUDA Python functions execute within a CUDA context. Each CUDA device in a system has an associated CUDA context, and Numba presently allows only one context per thread. For further details on CUDA Contexts, refer to the CUDA Driver API Documentation on Context Management and the CUDA C Programming Guide Context Documentation. CUDA Contexts are instances of the *Context* class:

class numba.cuda.cudadrv.driver.**Context**(*device*, *handle*)
> This object wraps a CUDA Context resource.
>
> Contexts should not be constructed directly by user code.
>
> **get_memory_info**()
> > Returns (free, total) memory in bytes in the context.

pop()
> Pops this context off the current CPU thread. Note that this context must be at the top of the context stack, otherwise an error will occur.

push()
> Pushes this context on the current CPU Thread.

reset()
> Clean up all owned resources in this context.

The following functions can be used to get or select the context:

numba.cuda.**current_context**(*devnum=None*)
> Get the current device or use a device by device number, and return the CUDA context.

numba.cuda.**require_context**(*fn*)
> A decorator that ensures a CUDA context is available when *fn* is executed.

> Note: The function *fn* cannot switch CUDA-context.

The following functions affect the current context:

numba.cuda.**synchronize**()
> Synchronize the current context.

numba.cuda.**close**()
> Explicitly clears all contexts in the current thread, and destroys all contexts if the current thread is the main thread.

Device management

Numba maintains a list of supported CUDA-capable devices:

numba.cuda.**gpus**
> An indexable list of supported CUDA devices. This list is indexed by integer device ID.

Alternatively, the current device can be obtained:

numba.cuda.gpus.**current**()
> Return the currently-selected device.

Getting a device through *numba.cuda.gpus* always provides an instance of *numba.cuda.cudadrv. devices._DeviceContextManager*, which acts as a context manager for the selected device:

class numba.cuda.cudadrv.devices.**_DeviceContextManager**(*device*)
> Provides a context manager for executing in the context of the chosen device. The normal use of instances of this type is from numba.cuda.gpus. For example, to execute on device 2:

```python
with numba.cuda.gpus[2]:
    d_a = numba.cuda.to_device(a)
```

> to copy the array *a* onto device 2, referred to by *d_a*.

One may also select a context and device or get the current device using the following three functions:

numba.cuda.**select_device**(*device_id*)
> Make the context associated with device *device_id* the current context.

> Returns a Device instance.

> Raises exception on error.

numba.cuda.**get_current_device**()
> Get current device associated with the current thread

numba.cuda.**list_devices**()
> Return a list of all detected devices

The *numba.cuda.cudadrv.driver.Device* class can be used to enquire about the functionality of the selected device:

class numba.cuda.cudadrv.driver.**Device**
> The device associated with a particular context.

> **compute_capability**
> > A tuple, *(major, minor)* indicating the supported compute capability.

> **id**
> > The integer ID of the device.

> **name**
> > The name of the device (e.g. "GeForce GTX 970")

> **reset**()
> > Delete the context for the device. This will destroy all memory allocations, events, and streams created within the context.

4.1.2 Compilation

Numba provides an entry point for compiling a Python function to PTX without invoking any of the driver API. This can be useful for:

- Generating PTX that is to be inlined into other PTX code (e.g. from outside the Numba / Python ecosystem).

- Generating code when there is no device present.

- Generating code prior to a fork without initializing CUDA.

Note: It is the user's responsibility to manage any ABI issues arising from the use of compilation to PTX.

---

numba.cuda.**compile_ptx**(*pyfunc*, *args*, *debug=False*, *device=False*, *fastmath=False*, *cc=None*, *opt=True*)
> Compile a Python function to PTX for a given set of argument types.

> **Parameters**

> > - **pyfunc** – The Python function to compile.

> > - **args** – A tuple of argument types to compile for.

> > - **debug** (*bool*) – Whether to include debug info in the generated PTX.

> > - **device** (*bool*) – Whether to compile a device function. Defaults to `False`, to compile global kernel functions.

> > - **fastmath** (*bool*) – Whether to enable fast math flags (ftz=1, prec_sqrt=0, prec_div=, and fma=1)

> > - **cc** (*tuple*) – Compute capability to compile for, as a tuple `(MAJOR, MINOR)`. Defaults to `(5, 2)`.

> > - **opt** (*bool*) – Enable optimizations. Defaults to `True`.

> **Returns** (ptx, resty): The PTX code and inferred return type

---

> > **Return type** tuple

The environment variable `NUMBA_CUDA_DEFAULT_PTX_CC` can be set to control the default compute capability targeted by `compile_ptx` - see *GPU support*. If PTX for the compute capability of the current device is required, the `compile_ptx_for_current_device` function can be used:

numba.cuda.**compile_ptx_for_current_device**(*pyfunc*, *args*, *debug=False*, *device=False*, *fast-math=False*, *opt=True*)

> Compile a Python function to PTX for a given set of argument types for the current device's compute capabilility. This calls *compile_ptx()* with an appropriate `cc` value for the current device.

4.1.3 Measurement

Profiling

The NVidia Visual Profiler can be used directly on executing CUDA Python code - it is not a requirement to insert calls to these functions into user code. However, these functions can be used to allow profiling to be performed selectively on specific portions of the code. For further information on profiling, see the NVidia Profiler User's Guide.

numba.cuda.**profile_start**()

> Enable profile collection in the current context.

numba.cuda.**profile_stop**()

> Disable profile collection in the current context.

numba.cuda.**profiling**()

> Context manager that enables profiling on entry and disables profiling on exit.

Events

Events can be used to monitor the progress of execution and to record the timestamps of specific points being reached. Event creation returns immediately, and the created event can be queried to determine if it has been reached. For further information, see the CUDA C Programming Guide Events section.

The following functions are used for creating and measuring the time between events:

numba.cuda.**event**(*timing=True*)

> Create a CUDA event. Timing data is only recorded by the event if it is created with `timing=True`.

numba.cuda.**event_elapsed_time**(*evtstart*, *evtend*)

> Compute the elapsed time between two events in milliseconds.

Events are instances of the *numba.cuda.cudadrv.driver.Event* class:

class numba.cuda.cudadrv.driver.**Event**(*context*, *handle*, *finalizer=None*)

> **query**()
>
> > Returns True if all work before the most recent record has completed; otherwise, returns False.
>
> **record**(*stream=0*)
>
> > Set the record point of the event to the current point in the given stream.
> >
> > The event will be considered to have occurred when all work that was queued in the stream at the time of the call to `record()` has been completed.
>
> **synchronize**()
>
> > Synchronize the host thread for the completion of the event.

> **wait** (*stream=0*)
>> All future works submitted to stream will wait util the event completes.

4.1.4 Stream Management

Streams allow concurrency of execution on a single device within a given context. Queued work items in the same stream execute sequentially, but work items in different streams may execute concurrently. Most operations involving a CUDA device can be performed asynchronously using streams, including data transfers and kernel execution. For further details on streams, see the CUDA C Programming Guide Streams section.

Streams are instances of *numba.cuda.cudadrv.driver.Stream*:

class numba.cuda.cudadrv.driver.**Stream**(*context*, *handle*, *finalizer*, *external=False*)

> **add_callback**(*callback*, *arg*)
>> Add a callback to a compute stream. The user provided function is called from a driver thread once all preceding stream operations are complete.
>>
>> Callback functions are called from a CUDA driver thread, not from the thread that invoked *add_callback*. No CUDA API functions may be called from within the callback function.
>>
>> The duration of a callback function should be kept short, as the callback will block later work in the stream and may block other callbacks from being executed.
>>
>> Note: This function is marked as deprecated and may be replaced in a future CUDA release.
>>
>>> **Parameters**
>>>
>>> - **callback** – Callback function with arguments (stream, status, arg).
>>> - **arg** – User data to be passed to the callback function.
>
> **async_done**() → _asyncio.Future
>> Return an awaitable that resolves once all preceding stream operations are complete.
>
> **auto_synchronize**()
>> A context manager that waits for all commands in this stream to execute and commits any pending memory transfers upon exiting the context.
>
> **synchronize**()
>> Wait for all commands in this stream to execute. This will commit any pending memory transfers.

To create a new stream:

numba.cuda.**stream**()
> Create a CUDA stream that represents a command queue for the device.

To get the default stream:

numba.cuda.**default_stream**()
> Get the default CUDA stream. CUDA semantics in general are that the default stream is either the legacy default stream or the per-thread default stream depending on which CUDA APIs are in use. In Numba, the APIs for the legacy default stream are always the ones in use, but an option to use APIs for the per-thread default stream may be provided in future.

To get the default stream with an explicit choice of whether it is the legacy or per-thread default stream:

numba.cuda.**legacy_default_stream**()
> Get the legacy default CUDA stream.

numba.cuda.**per_thread_default_stream**()
> Get the per-thread default CUDA stream.

To construct a Numba `Stream` object using a stream allocated elsewhere, the `external_stream` function is provided. Note that the lifetime of external streams must be managed by the user - Numba will not deallocate an external stream, and the stream must remain valid whilst the Numba `Stream` object is in use.

numba.cuda.**external_stream**(*ptr*)
> Create a Numba stream object for a stream allocated outside Numba.
>
> > **Parameters ptr** (*int*) – Pointer to the external stream to wrap in a Numba Stream

4.1.5 Runtime

Numba generally uses the Driver API, but it provides a simple wrapper to the Runtime API so that the version of the runtime in use can be queried. This is accessed through `cuda.runtime`, which is an instance of the *numba.cuda.cudadrv.runtime.Runtime* class:

class numba.cuda.cudadrv.runtime.**Runtime**
> Runtime object that lazily binds runtime API functions.
>
> **get_version**()
> > Returns the CUDA Runtime version as a tuple (major, minor).

4.2 CUDA Kernel API

4.2.1 Kernel declaration

The `@cuda.jit` decorator is used to create a CUDA dispatcher object that can be configured and launched:

numba.cuda.**jit**(*func_or_sig=None*, *argtypes=None*, *device=False*, *inline=False*, *link=[]*, *debug=None*, *opt=True*, ***kws*)
> JIT compile a python function conforming to the CUDA Python specification. If a signature is supplied, then a function is returned that takes a function to compile.
>
> **Parameters**
>
> - **func_or_sig** – A function to JIT compile, or a signature of a function to compile. If a function is supplied, then a `numba.cuda.compiler.AutoJitCUDAKernel` is returned. If a signature is supplied, then a function is returned. The returned function accepts another function, which it will compile and then return a `numba.cuda.compiler.AutoJitCUDAKernel`.
>
> ---
> **Note:** A kernel cannot have any return value.
>
> ---
>
> - **device** (*bool*) – Indicates whether this is a device function.
> - **bind** (*bool*) – (Deprecated) Force binding to CUDA context immediately
> - **link** (*list*) – A list of files containing PTX source to link with the function
> - **debug** – If True, check for exceptions thrown when executing the kernel. Since this degrades performance, this should only be used for debugging purposes. Defaults to False. (The default value can be overridden by setting environment variable `NUMBA_CUDA_DEBUGINFO=1`.)
> - **fastmath** – If true, enables flush-to-zero and fused-multiply-add, disables precise division and square root. This parameter has no effect on device function, whose fastmath setting depends on the kernel function from which they are called.

- **max_registers** – Limit the kernel to using at most this number of registers per thread. Useful for increasing occupancy.

- **opt** (*bool*) – Whether to compile from LLVM IR to PTX with optimization enabled. When `True`, `-opt=3` is passed to NVVM. When `False`, `-opt=0` is passed to NVVM. Defaults to `True`.

4.2.2 Dispatcher objects

The usual syntax for configuring a Dispatcher with a launch configuration uses subscripting, with the arguments being as in the following:

```
# func is some function decorated with @cuda.jit
func[griddim, blockdim, stream, sharedmem]
```

The `griddim` and `blockdim` arguments specify the size of the grid and thread blocks, and may be either integers or tuples of length up to 3. The `stream` parameter is an optional stream on which the kernel will be launched, and the `sharedmem` parameter specifies the size of dynamic shared memory in bytes.

Subscripting the Dispatcher returns a configuration object that can be called with the kernel arguments:

```
configured = func[griddim, blockdim, stream, sharedmem]
configured(x, y, z)
```

However, it is more idiomatic to configure and call the kernel within a single statement:

```
func[griddim, blockdim, stream, sharedmem](x, y, z)
```

This is similar to launch configuration in CUDA C/C++:

```
func<<<griddim, blockdim, sharedmem, stream>>>(x, y, z)
```

Note: The order of `stream` and `sharedmem` are reversed in Numba compared to in CUDA C/C++.

Dispatcher objects also provide several utility methods for inspection and creating a specialized instance:

class `numba.cuda.compiler.`**Dispatcher**(*func*, *sigs*, *bind*, *targetoptions*)

CUDA Dispatcher object. When configured and called, the dispatcher will specialize itself for the given arguments (if no suitable specialized version already exists) & compute capability, and launch on the device associated with the current context.

Dispatcher objects are not to be constructed by the user, but instead are created using the *numba.cuda.jit()* decorator.

extensions

A list of objects that must have a *prepare_args* function. When a specialized kernel is called, each argument will be passed through to the *prepare_args* (from the last object in this list to the first). The arguments to *prepare_args* are:

- *ty* the numba type of the argument

- *val* the argument value itself

- *stream* the CUDA stream used for the current call to the kernel

- *retr* a list of zero-arg functions that you may want to append post-call cleanup work to.

The *prepare_args* function must return a tuple *(ty, val)*, which will be passed in turn to the next right-most *extension*. After all the extensions have been called, the resulting *(ty, val)* will be passed into Numba's default argument marshalling logic.

inspect_asm(*signature=None*, *compute_capability=None*)

Return the generated PTX assembly code for all signatures encountered thus far, or the PTX assembly code for a specific signature and compute_capability if given. If the dispatcher is specialized, the assembly code for the single specialization is returned.

inspect_llvm(*signature=None*, *compute_capability=None*)

Return the LLVM IR for all signatures encountered thus far, or the LLVM IR for a specific signature and compute_capability if given. If the dispatcher is specialized, the IR for the single specialization is returned.

inspect_sass(*signature=None*, *compute_capability=None*)

Return the generated SASS code for all signatures encountered thus far, or the SASS code for a specific signature and compute_capability if given.

Requires nvdisasm to be available on the PATH.

inspect_types(*file=None*)

Produce a dump of the Python source of this function annotated with the corresponding Numba IR and type information. The dump is written to *file*, or *sys.stdout* if *file* is *None*.

specialize(**args*)

Create a new instance of this dispatcher specialized for the given *args*.

specialized

True if the Dispatcher has been specialized.

4.2.3 Intrinsic Attributes and Functions

The remainder of the attributes and functions in this section may only be called from within a CUDA Kernel.

Thread Indexing

numba.cuda.**threadIdx**

The thread indices in the current thread block, accessed through the attributes x, y, and z. Each index is an integer spanning the range from 0 inclusive to the corresponding value of the attribute in *numba.cuda. blockDim* exclusive.

numba.cuda.**blockIdx**

The block indices in the grid of thread blocks, accessed through the attributes x, y, and z. Each index is an integer spanning the range from 0 inclusive to the corresponding value of the attribute in *numba.cuda. gridDim* exclusive.

numba.cuda.**blockDim**

The shape of a block of threads, as declared when instantiating the kernel. This value is the same for all threads in a given kernel, even if they belong to different blocks (i.e. each block is "full").

numba.cuda.**gridDim**

The shape of the grid of blocks, accessed through the attributes x, y, and z.

numba.cuda.**laneid**

The thread index in the current warp, as an integer spanning the range from 0 inclusive to the *numba.cuda. warpsize* exclusive.

numba.cuda.**warpsize**

The size in threads of a warp on the GPU. Currently this is always 32.

numba.cuda.**grid**(*ndim*)

> Return the absolute position of the current thread in the entire grid of blocks. *ndim* should correspond to the number of dimensions declared when instantiating the kernel. If *ndim* is 1, a single integer is returned. If *ndim* is 2 or 3, a tuple of the given number of integers is returned.
>
> Computation of the first integer is as follows:
>
> ```
> cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
> ```
>
> and is similar for the other two indices, but using the y and z attributes.

numba.cuda.**gridsize**(*ndim*)

> Return the absolute size (or shape) in threads of the entire grid of blocks. *ndim* should correspond to the number of dimensions declared when instantiating the kernel.
>
> Computation of the first integer is as follows:
>
> ```
> cuda.blockDim.x * cuda.gridDim.x
> ```
>
> and is similar for the other two indices, but using the y and z attributes.

Memory Management

numba.cuda.shared.**array**(*shape*, *dtype*)

> Creates an array in the local memory space of the CUDA kernel with the given shape and dtype.
>
> Returns an array with its content uninitialized.
>
> ---
>
> **Note:** All threads in the same thread block sees the same array.
>
> ---

numba.cuda.local.**array**(*shape*, *dtype*)

> Creates an array in the local memory space of the CUDA kernel with the given shape and dtype.
>
> Returns an array with its content uninitialized.
>
> ---
>
> **Note:** Each thread sees a unique array.
>
> ---

numba.cuda.const.**array_like**(*ary*)

> Copies the ary into constant memory space on the CUDA kernel at compile time.
>
> Returns an array like the ary argument.
>
> ---
>
> **Note:** All threads and blocks see the same array.
>
> ---

Synchronization and Atomic Operations

numba.cuda.atomic.**add**(*array*, *idx*, *value*)

> Perform array[idx] += value. Support int32, int64, float32 and float64 only. The idx argument can be an integer or a tuple of integer indices for indexing into multiple dimensional arrays. The number of element in idx must match the number of dimension of array.
>
> Returns the value of array[idx] before the storing the new value. Behaves like an atomic load.

numba.cuda.atomic.**max**(*array*, *idx*, *value*)

> Perform `array[idx] = max(array[idx], value)`. Support int32, int64, float32 and float64 only. The `idx` argument can be an integer or a tuple of integer indices for indexing into multiple dimensional arrays. The number of element in `idx` must match the number of dimension of `array`.

> Returns the value of `array[idx]` before the storing the new value. Behaves like an atomic load.

numba.cuda.**syncthreads**()

> Synchronize all threads in the same thread block. This function implements the same pattern as barriers in traditional multi-threaded programming: this function waits until all threads in the block call it, at which point it returns control to all its callers.

numba.cuda.**syncthreads_count**(*predicate*)

> An extension to *numba.cuda.syncthreads* where the return value is a count of the threads where `predicate` is true.

numba.cuda.**syncthreads_and**(*predicate*)

> An extension to *numba.cuda.syncthreads* where 1 is returned if `predicate` is true for all threads or 0 otherwise.

numba.cuda.**syncthreads_or**(*predicate*)

> An extension to *numba.cuda.syncthreads* where 1 is returned if `predicate` is true for any thread or 0 otherwise.

> **Warning:** All syncthreads functions must be called by every thread in the thread-block. Falling to do so may result in undefined behavior.

Memory Fences

The memory fences are used to guarantee the effect of memory operations are visible by other threads within the same thread-block, the same GPU device, and the same system (across GPUs on global memory). Memory loads and stores are guaranteed to not move across the memory fences by optimization passes.

> **Warning:** The memory fences are considered to be advanced API and most usercases should use the thread barrier (e.g. `syncthreads()`).

numba.cuda.**threadfence**()

> A memory fence at device level (within the GPU).

numba.cuda.**threadfence_block**()

> A memory fence at thread block level.

numba.cuda.**threadfence_system**()

> A memory fence at system level (across GPUs).

Warp Intrinsics

All warp level operations require at least CUDA 9. The argument `membermask` is a 32 bit integer mask with each bit corresponding to a thread in the warp, with 1 meaning the thread is in the subset of threads within the function call. The `membermask` must be all 1 if the GPU compute capability is below 7.x.

numba.cuda.**syncwarp**(*membermask*)

> Synchronize a masked subset of the threads in a warp.

numba.cuda.**all_sync**(*membermask*, *predicate*)

> If the predicate is true for all threads in the masked warp, then a non-zero value is returned, otherwise 0 is returned.

numba.cuda.**any_sync**(*membermask*, *predicate*)

> If the predicate is true for any thread in the masked warp, then a non-zero value is returned, otherwise 0 is returned.

numba.cuda.**eq_sync**(*membermask*, *predicate*)

> If the boolean predicate is the same for all threads in the masked warp, then a non-zero value is returned, otherwise 0 is returned.

numba.cuda.**ballot_sync**(*membermask*, *predicate*)

> Returns a mask of all threads in the warp whose predicate is true, and are within the given mask.

numba.cuda.**shfl_sync**(*membermask*, *value*, *src_lane*)

> Shuffles value across the masked warp and returns the value from src_lane. If this is outside the warp, then the given value is returned.

numba.cuda.**shfl_up_sync**(*membermask*, *value*, *delta*)

> Shuffles value across the masked warp and returns the value from laneid - delta. If this is outside the warp, then the given value is returned.

numba.cuda.**shfl_down_sync**(*membermask*, *value*, *delta*)

> Shuffles value across the masked warp and returns the value from laneid + delta. If this is outside the warp, then the given value is returned.

numba.cuda.**shfl_xor_sync**(*membermask*, *value*, *lane_mask*)

> Shuffles value across the masked warp and returns the value from laneid ^ lane_mask.

numba.cuda.**match_any_sync**(*membermask*, *value*, *lane_mask*)

> Returns a mask of threads that have same value as the given value from within the masked warp.

numba.cuda.**match_all_sync**(*membermask*, *value*, *lane_mask*)

> Returns a tuple of (mask, pred), where mask is a mask of threads that have same value as the given value from within the masked warp, if they all have the same value, otherwise it is 0. And pred is a boolean of whether or not all threads in the mask warp have the same warp.

Integer Intrinsics

A subset of the CUDA Math API's integer intrinsics are available. For further documentation, including semantics, please refer to the CUDA Toolkit documentation.

numba.cuda.**popc**()

> Returns the number of set bits in the given value.

numba.cuda.**brev**()

> Reverses the bit pattern of an integer value, for example 0b10110110 becomes 0b01101101.

numba.cuda.**clz**()

> Counts the number of leading zeros in a value.

numba.cuda.**ffs**()

> Find the position of the least significant bit set to 1 in an integer.

Floating Point Intrinsics

A subset of the CUDA Math API's floating point intrinsics are available. For further documentation, including semantics, please refer to the single and double precision parts of the CUDA Toolkit documentation.

numba.cuda.**fma**()
>   Perform the fused multiply-add operation. Named after the `fma` and `fmaf` in the C api, but maps to the `fma.rn.f32` and `fma.rn.f64` (round-to-nearest-even) PTX instructions.

Control Flow Instructions

A subset of the CUDA's control flow instructions are directly available as intrinsics. Avoiding branches is a key way to improve CUDA performance, and using these intrinsics mean you don't have to rely on the `nvcc` optimizer identifying and removing branches. For further documentation, including semantics, please refer to the relevant CUDA Toolkit documentation.

numba.cuda.**selp**()
>   Select between two expressions, depending on the value of the first argument. Similar to LLVM's `select` instruction.

4.3 Memory Management

numba.cuda.**to_device**(*obj*, *stream=0*, *copy=True*, *to=None*)
>   Allocate and transfer a numpy ndarray or structured scalar to the device.
>
>   To copy host->device a numpy array:

```
ary = np.arange(10)
d_ary = cuda.to_device(ary)
```

>   To enqueue the transfer to a stream:

```
stream = cuda.stream()
d_ary = cuda.to_device(ary, stream=stream)
```

>   The resulting `d_ary` is a `DeviceNDArray`.
>
>   To copy device->host:

```
hary = d_ary.copy_to_host()
```

>   To copy device->host to an existing array:

```
ary = np.empty(shape=d_ary.shape, dtype=d_ary.dtype)
d_ary.copy_to_host(ary)
```

>   To enqueue the transfer to a stream:

```
hary = d_ary.copy_to_host(stream=stream)
```

numba.cuda.**device_array**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*, *stream=0*)
>   Allocate an empty device ndarray. Similar to `numpy.empty()`.

numba.cuda.**device_array_like**(*ary*, *stream=0*)
>   Call cuda.devicearray() with information from the array.

numba.cuda.**pinned_array**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*)
>   Allocate a np.ndarray with a buffer that is pinned (pagelocked). Similar to np.empty().

numba.cuda.**mapped_array**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*, *stream=0*, *portable=False*, *wc=False*)
>   Allocate a mapped ndarray with a buffer that is pinned and mapped on to the device. Similar to np.empty()

Parameters

- **portable** – a boolean flag to allow the allocated device memory to be usable in multiple devices.

- **wc** – a boolean flag to enable writecombined allocation which is faster to write by the host and to read by the device, but slower to write by the host and slower to write by the device.

numba.cuda.**pinned**(**arylist*)

A context manager for temporary pinning a sequence of host ndarrays.

numba.cuda.**mapped**(**arylist*, ***kws*)

A context manager for temporarily mapping a sequence of host ndarrays.

4.3.1 Device Objects

class numba.cuda.cudadrv.devicearray.**DeviceNDArray**(*shape*, *strides*, *dtype*, *stream=0*, *writeback=None*, *gpu_data=None*)

An on-GPU array type

copy_to_device(*ary*, *stream=0*)

Copy *ary* to *self*.

If *ary* is a CUDA memory, perform a device-to-device transfer. Otherwise, perform a a host-to-device transfer.

copy_to_host(*ary=None*, *stream=0*)

Copy `self` to `ary` or create a new Numpy ndarray if `ary` is `None`.

If a CUDA `stream` is given, then the transfer will be made asynchronously as part as the given stream. Otherwise, the transfer is synchronous: the function returns after the copy is finished.

Always returns the host array.

Example:

```
import numpy as np
from numba import cuda

arr = np.arange(1000)
d_arr = cuda.to_device(arr)

my_kernel[100, 100](d_arr)

result_array = d_arr.copy_to_host()
```

is_c_contiguous()

Return true if the array is C-contiguous.

is_f_contiguous()

Return true if the array is Fortran-contiguous.

ravel(*order='C'*, *stream=0*)

Flatten the array without changing its contents, similar to `numpy.ndarray.ravel()`.

reshape(**newshape*, ***kws*)

Reshape the array without changing its contents, similarly to `numpy.ndarray.reshape()`. Example:

```
d_arr = d_arr.reshape(20, 50, order='F')
```

split (*section*, *stream=0*)
> Split the array into equal partition of the *section* size. If the array cannot be equally divided, the last section will be smaller.

class numba.cuda.cudadrv.devicearray.**DeviceRecord** (*dtype*, *stream=0*, *gpu_data=None*)
> An on-GPU record type

> **copy_to_device** (*ary*, *stream=0*)
>> Copy *ary* to *self*.

>> If *ary* is a CUDA memory, perform a device-to-device transfer. Otherwise, perform a a host-to-device transfer.

> **copy_to_host** (*ary=None*, *stream=0*)
>> Copy `self` to `ary` or create a new Numpy ndarray if `ary` is `None`.

>> If a CUDA `stream` is given, then the transfer will be made asynchronously as part as the given stream. Otherwise, the transfer is synchronous: the function returns after the copy is finished.

>> Always returns the host array.

>> Example:

```python
import numpy as np
from numba import cuda

arr = np.arange(1000)
d_arr = cuda.to_device(arr)

my_kernel[100, 100](d_arr)

result_array = d_arr.copy_to_host()
```

class numba.cuda.cudadrv.devicearray.**MappedNDArray** (*shape*, *strides*, *dtype*, *stream=0*, *writeback=None*, *gpu_data=None*)
> A host array that uses CUDA mapped memory.

> **copy_to_device** (*ary*, *stream=0*)
>> Copy *ary* to *self*.

>> If *ary* is a CUDA memory, perform a device-to-device transfer. Otherwise, perform a a host-to-device transfer.

> **copy_to_host** (*ary=None*, *stream=0*)
>> Copy `self` to `ary` or create a new Numpy ndarray if `ary` is `None`.

>> If a CUDA `stream` is given, then the transfer will be made asynchronously as part as the given stream. Otherwise, the transfer is synchronous: the function returns after the copy is finished.

>> Always returns the host array.

>> Example:

```python
import numpy as np
from numba import cuda

arr = np.arange(1000)
d_arr = cuda.to_device(arr)

my_kernel[100, 100](d_arr)
```

```
result_array = d_arr.copy_to_host()
```

split(*section*, *stream=0*)

 Split the array into equal partition of the *section* size. If the array cannot be equally divided, the last section will be smaller.

Numba for AMD ROC GPUs

5.1 Overview

Numba supports AMD ROC GPU programming by directly compiling a restricted subset of Python code into HSA kernels and device functions following the HSA execution model. Kernels written in Numba appear to have direct access to NumPy arrays.

5.1.1 Terminology

Several important terms in the topic of HSA programming are listed here:

- *kernels*: a GPU function launched by the host and executed on the device

- *device function*: a GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

5.1.2 Requirements

This document describes the requirements for using ROC. Essentially an AMD dGPU is needed (Fiji, Polaris and Vega families) and a CPU which supports PCIe Gen3 and PCIe Atomics (AMD Ryzen and EPYC, and Intel CPUs >= Haswell), full details are in the linked document. Further a linux operating system is needed, those supported and tested are also listed in the linked document.

5.1.3 Installation

Follow this document for installation instructions to enable ROC support for the system. Be sure to use the binary packages for the system's linux distribution to simplify the process. At this point the install should be tested by running:

```
$ /opt/rocm/bin/rocminfo
```

the output of which should list at least two HSA Agents, at least one of which should be a CPU and at least one of which should be a dGPU.

Assuming the installation is working correctly, the ROC support for Numba is provided by the `roctools` package which can be installed via `conda`, along with Numba, from the Numba channel as follows (creating an env called `numba_roc`):

```
$ conda create -n numba_roc -c numba numba roctools
```

Activating the env, and then running the Numba diagnostic tool should confirm that Numba is running with ROC support enabled, e.g.:

```
$ source activate numba_roc
$ numba -s
```

The output of `numba -s` should contain a section similar to:

```
__ROC Information__
ROC available                         : True
Available Toolchains                  : librocmlite library, ROC command line tools

Found 2 HSA Agents:
Agent id  : 0
    vendor: CPU
    name: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz
    type: CPU

Agent id  : 1
    vendor: AMD
    name: gfx803
    type: GPU

Found 1 discrete GPU(s)          : gfx803
```

confirming that ROC is available, listing the available toolchains and displaying the HSA Agents and dGPU count.

5.2 Writing HSA Kernels

5.2.1 Introduction

HSA provides an execution model similar to OpenCL. Instructions are executed in parallel by a group of hardware threads. In some way, this is similar to *single-instruction-multiple-data* (SIMD) model but with the convenience that the fine-grain scheduling is hidden from the programmer instead of programming with SIMD vectors as a data structure. In HSA, the code you write will be executed by multiple threads at once (often hundreds or thousands). Your solution will be modelled by defining a thread hierarchy of *grid*, *workgroup* and *workitem*.

Numba's HSA support exposes facilities to declare and manage this hierarchy of threads.

5.2.2 Introduction for CUDA Programmers

HSA execution model is similar to CUDA. The memory model employed by HSA on ROC GPUs is also similar to that of CUDA. ROC GPUs have dedicated on GPU memory, therefore the `to_device()` and `copy_to_host()` etc. are required as per CUDA.

Here's a quick mapping of the CUDA terms to HSA:

- A `workitem` is equivalent to a CUDA thread.

- A `workgroup` is equivalent to a CUDA thread block.

- A `grid` is equivalent to a CUDA grid.

- A `wavefront` is equivalent to a CUDA `warp`.

5.2.3 Kernel declaration

A *kernel function* is a GPU function that is meant to be called from CPU code. It gives it two fundamental characteristics:

- kernels cannot explicitly return a value; all result data must be written to an array passed to the function (if computing a scalar, you will probably pass a one-element array);

- kernels explicitly declare their thread hierarchy when called: i.e. the number of workgroups and the number of workitems per workgroup (note that while a kernel is compiled once, it can be called multiple times with different workgroup sizes or grid sizes).

At first sight, writing a HSA kernel with Numba looks very much like writing a *JIT function* for the CPU:

```python
@roc.jit
def increment_by_one(an_array):
    """
    Increment all array elements by one.
    """
    # code elided here; read further for different implementations
```

5.2.4 Kernel invocation

A kernel is typically launched in the following way:

```python
itempergroup = 32
groupperrange = (an_array.size + (itempergroup - 1)) // itempergroup
increment_by_one[groupperrange, itempergroup](an_array)
```

We notice two steps here:

- Instantiate the kernel proper, by specifying a number of workgroup (or "workgroup per grid"), and a number of workitems per workgroup. The product of the two will give the total number of workitem launched. Kernel instantiation is done by taking the compiled kernel function (here `increment_by_one`) and indexing it with a tuple of integers.

- Running the kernel, by passing it the input array (and any separate output arrays if necessary). By default, running a kernel is synchronous: the function returns when the kernel has finished executing and the data is synchronized back.

Choosing the workgroup size

It might seem curious to have a two-level hierarchy when declaring the number of workitem needed by a kernel. The workgroup size (i.e. number of workitem per workgroup) is often crucial:

- On the software side, the workgroup size determines how many threads share a given area of *shared memory*.

- **On the hardware side, the workgroup size must be large enough for full** occupation of execution units.

Multi-dimensional workgroup and grid

To help deal with multi-dimensional arrays, HSA allows you to specify multi-dimensional workgroups and grids. In the example above, you could make `itempergroup` and `groupperrange` tuples of one, two or three integers. Compared to 1D declarations of equivalent sizes, this doesn't change anything to the efficiency or behaviour of generated code, but can help you write your algorithms in a more natural way.

5.2.5 WorkItem positioning

When running a kernel, the kernel function's code is executed by every thread once. It therefore has to know which thread it is in, in order to know which array element(s) it is responsible for (complex algorithms may define more complex responsibilities, but the underlying principle is the same).

One way is for the thread to determines its position in the grid and workgroup and manually compute the corresponding array position:

```
@roc.jit
def increment_by_one(an_array):
    # workitem id in a 1D workgroup
    tx = roc.get_local_id(0)
    # workgroup id in a 1D grid
    ty = roc.get_group_id(0)
    # workgroup size, i.e. number of workitem per workgroup
    bw = roc.get_local_size(0)
    # Compute flattened index inside the array
    pos = tx + ty * bw
    # The above is equivalent to pos = roc.get_global_id(0)
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] += 1
```

Note: Unless you are sure the workgroup size and grid size is a divisor of your array size, you **must** check boundaries as shown above.

---

get_local_id(), *get_local_size()*, *get_group_id()* and *get_global_id()* are special functions provided by the HSA backend for the sole purpose of knowing the geometry of the thread hierarchy and the position of the current workitem within that geometry.

numba.roc.**get_local_id**(*dim*)
> Takes the index of the dimension being queried

> Returns local workitem ID in the the current workgroup for the given dimension. For 1D workgroup, the index is an integer spanning the range from 0 inclusive to *numba.roc.get_local_size()* exclusive.

numba.roc.**get_local_size**(*dim*)
> Takes the index of the dimension being queried

> Returns the size of the workgroup at the given dimension. The value is declared when instantiating the kernel. This value is the same for all workitems in a given kernel, even if they belong to different workgroups (i.e. each workgroups is "full").

numba.roc.**get_group_id**(*dim*)
> Takes the index of the dimension being queried

> Returns the workgroup ID in the grid of workgroup launched a kernel.

numba.roc.**get_global_id**(*dim*)
> Takes the index of the dimension being queried

Returns the global workitem ID for the given dimension. Unlike *numba.roc .get_local_id()*, this number is unique for all workitems in a grid.

5.3 Memory management

5.3.1 Data transfer

Even though Numba can automatically transfer NumPy arrays to the device, it can only do so conservatively by always transferring device memory back to the host when a kernel finishes. To avoid the unnecessary transfer for read-only arrays, you can use the following APIs to manually control the transfer:

numba.roc.**device_array**(*shape*, *dtype=np.float*, *strides=None*, *order='C'*)
> Allocate an empty device ndarray. Similar to `numpy.empty()`.

numba.roc.**device_array_like**(*ary*)
> Call roc.devicearray() with information from the array.

numba.roc.**to_device**(*obj*, *context*, *copy=True*, *to=None*)
> Allocate and transfer a numpy ndarray or structured scalar to the device.
>
> To copy host->device a numpy array:

```
ary = numpy.arange(10)
d_ary = roc.to_device(ary)
```

> The resulting `d_ary` is a `DeviceNDArray`.
>
> To copy device->host:

```
hary = d_ary.copy_to_host()
```

> To copy device->host to an existing array:

```
ary = numpy.empty(shape=d_ary.shape, dtype=d_ary.dtype)
d_ary.copy_to_host(ary)
```

Device arrays

Device array references have the following methods. These methods are to be called in host code, not within ROC-jitted functions.

class numba.roc.hsadrv.devicearray.**DeviceNDArray**(*shape*,   *strides*,   *dtype*, *dgpu_data=None*)
> An on-dGPU array type

> **copy_to_host**(*ary=None*, *stream=None*)
> > Copy `self` to `ary` or create a new Numpy ndarray if `ary` is `None`.
> >
> > The transfer is synchronous: the function returns after the copy is finished.
> >
> > Always returns the host array.
> >
> > Example:

```python
import numpy as np
from numba import hsa

arr = np.arange(1000)
d_arr = hsa.to_device(arr)

my_kernel[100, 100](d_arr)

result_array = d_arr.copy_to_host()
```

is_c_contiguous()
> Return true if the array is C-contiguous.

is_f_contiguous()
> Return true if the array is Fortran-contiguous.

ravel(*order='C'*)
> Flatten the array without changing its contents, similar to `numpy.ndarray.ravel()`.

reshape(**newshape*, ***kws*)
> Reshape the array without changing its contents, similarly to `numpy.ndarray.reshape()`. Example:

```python
d_arr = d_arr.reshape(20, 50, order='F')
```

Data Registration

The CPU and GPU do not share the same main memory, however, it is recommended to register a memory allocation to the HSA runtime for as a performance optimisation hint.

roc.**register**(**arrays*)
> Register every given array. The function can be used in a *with-context* for automically deregistration:

```python
array_a = numpy.arange(10)
array_b = numpy.arange(10)
with roc.register(array_a, array_b):
    some_hsa_code(array_a, array_b)
```

roc.**deregister**(**arrays*)
> Deregister every given array

5.3.2 Streams

numba.roc.**stream**()

ROC streams have the following methods:

class numba.roc.hsadrv.driver.**Stream**
> An asynchronous stream for async API

> **auto_synchronize**()
> > A context manager that waits for all commands in this stream to execute and commits any pending memory transfers upon exiting the context.

> **synchronize**()
> > Synchronize the stream.

5.3.3 Shared memory and thread synchronization

A limited amount of shared memory can be allocated on the device to speed up access to data, when necessary. That memory will be shared (i.e. both readable and writable) amongst all workitems belonging to a given group and has faster access times than regular device memory. It also allows workitems to cooperate on a given solution. You can think of it as a manually-managed data cache.

The memory is allocated once for the duration of the kernel, unlike traditional dynamic memory management.

numba.roc.shared.**array**(*shape*, *type*)
> Allocate a shared array of the given *shape* and *type* on the device. This function must be called on the device (i.e. from a kernel or device function). *shape* is either an integer or a tuple of integers representing the array's dimensions. *type* is a *Numba type* of the elements needing to be stored in the array.
>
> The returned array-like object can be read and written to like any normal device array (e.g. through indexing).
>
> A common pattern is to have each workitem populate one element in the shared array and then wait for all workitems to finish using :func:' .barrier'.

numba.roc.**barrier**(*scope*)
> The scope argument specifies the level of synchronization. Set scope to roc.CLK_GLOBAL_MEM_FENCE or roc.CLK_LOCAL_MEM_FENCE to synchronize all workitems across a workgroup when accessing global memory or local memory respectively.

numba.roc.**wavebarrier**()
> Creates an execution barrier across a wavefront to force a synchronization point.

See also:

Matrix multiplication example.

5.4 Writing Device Functions

HSA device functions are functions that will run on the device but can only be invoked from a kernel or another device function. Unlike a kernel function, a device function can return a value like normal functions. To define a device function the kwarg device must be set to True in the roc.jit decorator:

```python
from numba import roc

@roc.jit(device=True)
def a_device_function(a, b):
    return a + b
```

An example of using a device function:

```python
from numba import roc
import numpy as np

@roc.jit
def kernel(an_array):
    pos = roc.get_global_id(0)
    if pos < an_array.size:  # Check array boundaries
        an_array[pos] = a_device_function(1, pos) # call device function

@roc.jit(device = True)
def a_device_function(a, b):
    return a + b
```
(continues on next page)

```
n = 16
x = np.zeros(n)

kernel[1, n](x)

print(x)
```

5.5 Supported Atomic Operations

Numba provides access to some of the atomic operations supported in HSA, in the `numba.roc.atomic` class.

5.5.1 Example

The following code demonstrates the use of `numba.roc.atomic.add` to count every number in [0,32) occurred in the input array in parallel:

```python
from numba import roc
import numpy as np

@roc.jit
def hsa_atomic_histogram(ary):
    tid = roc.get_local_id(0)
    sm = roc.shared.array(32, numba.uint32)   # declare shared library
    sm[tid] = 0                               # init values to zero
    roc.barrier(1)                            # synchronize (wait for init)
    loc = ary[tid] % 32                       # ensure we are in range
    roc.atomic.add(sm, loc, 1)                # atomic add
    roc.barrier(1)                            # synchronize
    ary[tid] = sm[tid]                        # store result inplace

ary = np.random.randint(0, 32, size=32).astype(np.uint32)
orig = ary.copy()

# HSA version
hsa_atomic_histogram[1, 32](ary)

# Expected behavior
gold = np.zeros_like(ary)
for i in range(orig.size):
    gold[orig[i]] += 1

print(ary)  # HSA kernel result
print(gold) # for comparison
```

5.6 The Agents

The HSA agents found in the system can be listed and accessed via:

numba.roc.**agents**

5.7 ROC Ufuncs and Generalized Ufuncs

This page describes the ROC ufunc-like object.

To support the programming pattern of ROC programs, ROC Vectorize and GUVectorize cannot produce a conventional ufunc. Instead, a ufunc-like object is returned. This object is a close analog but not fully compatible with a regular NumPy ufunc. The ROC ufunc adds support for passing intra-device arrays (already on the GPU device) to reduce traffic over the PCI-express bus. It also accepts a *stream* keyword for launching in asynchronous mode.

5.7.1 Basic ROC UFunc Example

```python
import math
from numba import vectorize
import numpy as np

@vectorize(['float32(float32, float32, float32)',
            'float64(float64, float64, float64)'],
           target='roc')
def roc_discriminant(a, b, c):
    return math.sqrt(b ** 2 - 4 * a * c)

N = 10000
dtype = np.float32

# prepare the input
A = np.array(np.random.sample(N), dtype=dtype)
B = np.array(np.random.sample(N) + 10, dtype=dtype)
C = np.array(np.random.sample(N), dtype=dtype)

D = roc_discriminant(A, B, C)

print(D)  # print result
```

5.7.2 Calling Device Functions from ROC UFuncs

All ROC ufunc kernels have the ability to call other ROC device functions:

```python
from numba import vectorize, roc

# define a device function
@roc.jit('float32(float32, float32, float32)', device=True)
def roc_device_fn(x, y, z):
    return x ** y / z

# define a ufunc that calls our device function
@vectorize(['float32(float32, float32, float32)'], target='roc')
def roc_ufunc(x, y, z):
    return roc_device_fn(x, y, z)
```

5.7.3 Generalized ROC ufuncs

Generalized ufuncs may be executed on the GPU using ROC, analogous to the ROC ufunc functionality. This may be accomplished as follows:

```python
from numba import guvectorize

@guvectorize(['void(float32[:,:], float32[:,:], float32[:,:])'],
             '(m,n),(n,p)->(m,p)', target='roc')
def matmulcore(A, B, C):
    ...
```

See also:

Matrix multiplication example.

5.7.4 Async execution: A Chunk at a Time

Partitioning your data into chunks allows computation and memory transfer to be overlapped. This can increase the throughput of your ufunc and enables your ufunc to operate on data that is larger than the memory capacity of your GPU. For example:

```python
import math
from numba import vectorize, roc
import numpy as np

# the ufunc kernel
def discriminant(a, b, c):
    return math.sqrt(b ** 2 - 4 * a * c)

roc_discriminant = vectorize(['float32(float32, float32, float32)'],
                             target='roc')(discriminant)

N = int(1e+8)
dtype = np.float32

# prepare the input
A = np.array(np.random.sample(N), dtype=dtype)
B = np.array(np.random.sample(N) + 10, dtype=dtype)
C = np.array(np.random.sample(N), dtype=dtype)
D = np.zeros(A.shape, dtype=A.dtype)

# create a ROC stream
stream = roc.stream()

chunksize = 1e+6
chunkcount = N // chunksize

# partition numpy arrays into chunks
# no copying is performed
sA = np.split(A, chunkcount)
sB = np.split(B, chunkcount)
sC = np.split(C, chunkcount)
sD = np.split(D, chunkcount)

device_ptrs = []

# helper function, async requires operation on coarsegrain memory regions
def async_array(arr):
    coarse_arr = roc.coarsegrain_array(shape=arr.shape, dtype=arr.dtype)
    coarse_arr[:] = arr
```

(continues on next page)

```python
    return coarse_arr

with stream.auto_synchronize():
    # every operation in this context with be launched asynchronously
    # by using the ROC stream

    dchunks = [] # holds the result chunks

    # for each chunk
    for a, b, c, d in zip(sA, sB, sC, sD):
        # create coarse grain arrays
        asyncA = async_array(a)
        asyncB = async_array(b)
        asyncC = async_array(c)
        asyncD = async_array(d)

        # transfer to device
        dA = roc.to_device(asyncA, stream=stream)
        dB = roc.to_device(asyncB, stream=stream)
        dC = roc.to_device(asyncC, stream=stream)
        dD = roc.to_device(asyncD, stream=stream, copy=False) # no copying

        # launch kernel
        roc_discriminant(dA, dB, dC, out=dD, stream=stream)

        # retrieve result
        dD.copy_to_host(asyncD, stream=stream)

        # store device pointers to prevent them from freeing before
        # the kernel is scheduled
        device_ptrs.extend([dA, dB, dC, dD])

        # store result reference
        dchunks.append(asyncD)

# put result chunks into the output array 'D'
for i, result in enumerate(dchunks):
    sD[i][:] = result[:]

# data is ready at this point inside D
print(D)
```

5.8 Examples

5.8.1 Matrix multiplication

Here is a naive implementation of matrix multiplication using a HSA kernel:

```python
@roc.jit
def matmul(A, B, C):
    i = roc.get_global_id(0)
    j = roc.get_global_id(1)

    if i >= C.shape[0] or j >= C.shape[1]:
```

```
        return

    tmp = 0

    for k in range(A.shape[1]):
        tmp += A[i, k] * B[k, j]

    C[i, j] = tmp
```

This implementation is straightforward and intuitive but performs poorly, because the same matrix elements will be loaded multiple times from device memory, which is slow (some devices may have transparent data caches, but they may not be large enough to hold the entire inputs at once).

It will be faster if we use a blocked algorithm to reduce accesses to the device memory. HSA provides a fast *shared memory* for workitems in a group to cooperatively compute on a task. The following implements a faster version of the square matrix multiplication using shared memory:

```python
import numpy as np
from numba import roc
from numba import float32
from time import time as timer

blocksize = 16
gridsize = 16

@roc.jit('(float32[:,:], float32[:,:], float32[:,:])')
def matmulfast(A, B, C):
    x = roc.get_global_id(0)
    y = roc.get_global_id(1)

    tx = roc.get_local_id(0)
    ty = roc.get_local_id(1)

    sA = roc.shared.array(shape=(blocksize, blocksize), dtype=float32)
    sB = roc.shared.array(shape=(blocksize, blocksize), dtype=float32)

    if x >= C.shape[0] or y >= C.shape[1]:
        return

    tmp = 0

    for i in range(gridsize):
        # preload
        sA[tx, ty] = A[x, ty + i * blocksize]
        sB[tx, ty] = B[tx + i * blocksize, y]
        # wait for preload to end
        roc.barrier(1)
        # compute loop
        for j in range(blocksize):
            tmp += sA[tx, j] * sB[j, ty]
        # wait for compute to end
        roc.barrier(1)

    C[x, y] = tmp

N = gridsize * blocksize
A = np.random.random((N, N)).astype(np.float32)
```

```python
B = np.random.random((N, N)).astype(np.float32)
C = np.zeros_like(A)

griddim = gridsize, gridsize
blockdim = blocksize, blocksize

with roc.register(A, B, C):
    ts = timer()
    matmulfast[griddim, blockdim](A, B, C)
    te = timer()
    print("1st GPU time:", te - ts)

with roc.register(A, B, C):
    ts = timer()
    matmulfast[griddim, blockdim](A, B, C)
    te = timer()
    print("2nd GPU time:", te - ts)

ts = timer()
ans = np.dot(A, B)
te = timer()
print("CPU time:", te - ts)
np.testing.assert_allclose(ans, C, rtol=1e-5)
```

Because the shared memory is a limited resource, the code preloads a small block at a time from the input arrays. Then, it calls *barrier()* to wait until all threads have finished preloading before doing the computation on the shared memory. It synchronizes again after the computation to ensure all threads have finished with the data in shared memory before overwriting it in the next loop iteration.

Extending Numba

This chapter describes how to extend Numba to make it recognize and support additional operations, functions or types. Numba provides two categories of APIs to this end:

- The high-level APIs provide abstracted entry points which are sufficient for simple uses. They require little knowledge of Numba's internal compilation chain.

- The low-level APIs reflect Numba's internal compilation chain and allow flexible interaction with its various layers, but require more effort and experience with Numba internals.

It may be helpful for readers of this chapter to also read some of the documents in the *developer manual*, especially the *architecture document*.

6.1 High-level extension API

This extension API is exposed through the *numba.extending* module.

6.1.1 Implementing functions

The `@overload` decorator allows you to implement arbitrary functions for use in *nopython mode* functions. The function decorated with `@overload` is called at compile-time with the *types* of the function's runtime arguments. It should return a callable representing the *implementation* of the function for the given types. The returned implementation is compiled by Numba as if it were a normal function decorated with `@jit`. Additional options to `@jit` can be passed as dictionary using the `jit_options` argument.

For example, let's pretend Numba doesn't support the `len()` function on tuples yet. Here is how to implement it using `@overload`:

```
from numba import types
from numba.extending import overload

@overload(len)
```

(continues on next page)

```
def tuple_len(seq):
    if isinstance(seq, types.BaseTuple):
        n = len(seq)
        def len_impl(seq):
            return n
        return len_impl
```

You might wonder, what happens if `len()` is called with something else than a tuple? If a function decorated with `@overload` doesn't return anything (i.e. returns None), other definitions are tried until one succeeds. Therefore, multiple libraries may overload `len()` for different types without conflicting with each other.

6.1.2 Implementing methods

The `@overload_method` decorator similarly allows implementing a method on a type well-known to Numba. The following example implements the `take()` method on Numpy arrays:

```
@overload_method(types.Array, 'take')
def array_take(arr, indices):
    if isinstance(indices, types.Array):
        def take_impl(arr, indices):
            n = indices.shape[0]
            res = np.empty(n, arr.dtype)
            for i in range(n):
                res[i] = arr[indices[i]]
            return res
        return take_impl
```

6.1.3 Implementing attributes

The `@overload_attribute` decorator allows implementing a data attribute (or property) on a type. Only reading the attribute is possible; writable attributes are only supported through the *low-level API*.

The following example implements the `nbytes` attribute on Numpy arrays:

```
@overload_attribute(types.Array, 'nbytes')
def array_nbytes(arr):
    def get(arr):
        return arr.size * arr.itemsize
    return get
```

6.1.4 Importing Cython Functions

The function `get_cython_function_address` obtains the address of a C function in a Cython extension module. The address can be used to access the C function via a `ctypes.CFUNCTYPE()` callback, thus allowing use of the C function inside a Numba jitted function. For example, suppose that you have the file `foo.pyx`:

```
from libc.math cimport exp

cdef api double myexp(double x):
    return exp(x)
```

You can access `myexp` from Numba in the following way:

```
import ctypes
from numba.extending import get_cython_function_address

addr = get_cython_function_address("foo", "myexp")
functype = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
myexp = functype(addr)
```

The function `myexp` can now be used inside jitted functions, for example:

```
@njit
def double_myexp(x):
    return 2*myexp(x)
```

One caveat is that if your function uses Cython's fused types, then the function's name will be mangled. To find out the mangled name of your function you can check the extension module's `__pyx_capi__` attribute.

6.1.5 Implementing intrinsics

The `@intrinsic` decorator is used for marking a function *func* as typing and implementing the function in `nopython` mode using the llvmlite IRBuilder API. This is an escape hatch for expert users to build custom LLVM IR that will be inlined into the caller, there is no safety net!

The first argument to *func* is the typing context. The rest of the arguments corresponds to the type of arguments of the decorated function. These arguments are also used as the formal argument of the decorated function. If *func* has the signature `foo(typing_context, arg0, arg1)`, the decorated function will have the signature `foo(arg0, arg1)`.

The return values of *func* should be a 2-tuple of expected type signature, and a code-generation function that will passed to `lower_builtin()`. For an unsupported operation, return `None`.

Here is an example that cast any integer to a byte pointer:

```
from numba import types
from numba.extending import intrinsic

@intrinsic
def cast_int_to_byte_ptr(typingctx, src):
    # check for accepted types
    if isinstance(src, types.Integer):
        # create the expected type signature
        result_type = types.CPointer(types.uint8)
        sig = result_type(types.uintp)
        # defines the custom code generation
        def codegen(context, builder, signature, args):
            # llvm IRBuilder code here
            [src] = args
            rtype = signature.return_type
            llrtype = context.get_value_type(rtype)
            return builder.inttoptr(src, llrtype)
        return sig, codegen
```

it may be used as follows:

```
from numba import njit

@njit('void(int64)')
```

(continues on next page)

```
def foo(x):
    y = cast_int_to_byte_ptr(x)

foo.inspect_types()
```

and the output of `.inspect_types()` demonstrates the cast (note the `uint8*`):

```
def foo(x):

    #   x = arg(0, name=x)  :: int64
    #   $0.1 = global(cast_int_to_byte_ptr: <intrinsic cast_int_to_byte_ptr>)  ::
    →Function(<intrinsic cast_int_to_byte_ptr>)
    #   $0.3 = call $0.1(x, func=$0.1, args=[Var(x, check_intrin.py (24))], kws=(),
    →vararg=None)  :: (uint64,) -> uint8*
    #   del x
    #   del $0.1
    #   y = $0.3  :: uint8*
    #   del y
    #   del $0.3
    #   $const0.4 = const(NoneType, None)  :: none
    #   $0.5 = cast(value=$const0.4)  :: none
    #   del $const0.4
    #   return $0.5

    y = cast_int_to_byte_ptr(x)
```

6.1.6 Implementing mutable structures

> **Warning:** This is an experimental feature, the API may change without warning.

The `numba.experimental.structref` module provides utilities for defining mutable pass-by-reference structures, a `StructRef`. The following example demonstrates how to define a basic mutable structure:

Defining a StructRef

Listing 1: from `numba/tests/doc_examples/test_structref_usage.py`

```python
1  import numpy as np
2
3  from numba import njit
4  from numba.core import types
5  from numba.experimental import structref
6
7  from numba.tests.support import skip_unless_scipy
8
9
10 # Define a StructRef.
11 # `structref.register` associates the type with the default data model.
12 # This will also install getters and setters to the fields of
13 # the StructRef.
```

```python
14  @structref.register
15  class MyStructType(types.StructRef):
16      def preprocess_fields(self, fields):
17          # This method is called by the type constructor for additional
18          # preprocessing on the fields.
19          # Here, we don't want the struct to take Literal types.
20          return tuple((name, types.unliteral(typ)) for name, typ in fields)
21
22
23  # Define a Python type that can be use as a proxy to the StructRef
24  # allocated inside Numba. Users can construct the StructRef via
25  # the constructor for this type in python code and jit-code.
26  class MyStruct(structref.StructRefProxy):
27      def __new__(cls, name, vector):
28          # Overriding the __new__ method is optional, doing so
29          # allows Python code to use keyword arguments,
30          # or add other customized behavior.
31          # The default __new__ takes `*args`.
32          # IMPORTANT: Users should not override __init__.
33          return structref.StructRefProxy.__new__(cls, name, vector)
34
35      # By default, the proxy type does not reflect the attributes or
36      # methods to the Python side. It is up to users to define
37      # these. (This may be automated in the future.)
38
39      @property
40      def name(self):
41          # To access a field, we can define a function that simply
42          # return the field in jit-code.
43          # The definition of MyStruct_get_name is shown later.
44          return MyStruct_get_name(self)
45
46      @property
47      def vector(self):
48          # The definition of MyStruct_get_vector is shown later.
49          return MyStruct_get_vector(self)
50
51
52  @njit
53  def MyStruct_get_name(self):
54      # In jit-code, the StructRef's attribute is exposed via
55      # structref.register
56      return self.name
57
58
59  @njit
60  def MyStruct_get_vector(self):
61      return self.vector
62
63
64  # This associates the proxy with MyStructType for the given set of
65  # fields. Notice how we are not contraining the type of each field.
66  # Field types remain generic.
67  structref.define_proxy(MyStruct, MyStructType, ["name", "vector"])
```

The following demonstrates using the above mutable struct definition:

Listing 2: from `test_type_definition` of `numba/tests/`
`doc_examples/test_structref_usage.py`

```
1    # Let's test our new StructRef.
2
3    # Define one in Python
4    alice = MyStruct("Alice", vector=np.random.random(3))
5
6    # Define one in jit-code
7    @njit
8    def make_bob():
9        bob = MyStruct("unnamed", vector=np.zeros(3))
10       # Mutate the attributes
11       bob.name = "Bob"
12       bob.vector = np.random.random(3)
13       return bob
14
15   bob = make_bob()
16
17   # Out: Alice: [0.5488135  0.71518937 0.60276338]
18   print(f"{alice.name}: {alice.vector}")
19   # Out: Bob: [0.88325739 0.73527629 0.87746707]
20   print(f"{bob.name}: {bob.vector}")
21
22   # Define a jit function to operate on the structs.
23   @njit
24   def distance(a, b):
25       return np.linalg.norm(a.vector - b.vector)
26
27   # Out: 0.4332647200356598
28   print(distance(alice, bob))
```

Defining a method on StructRef

Methods and attributes can be attached using `@overload_*` as shown in the previous sections.

The following demonstrates the use of `@overload_method` to insert a method for instances of `MyStructType`:

Listing 3: from `test_overload_method` of `numba/tests/`
`doc_examples/test_structref_usage.py`

```
1    from numba.core.extending import overload_method
2    from numba.core.errors import TypingError
3
4    # Use @overload_method to add a method for
5    # MyStructType.distance(other)
6    # where *other* is an instance of MyStructType.
7    @overload_method(MyStructType, "distance")
8    def ol_distance(self, other):
9        # Guard that *other* is an instance of MyStructType
10       if not isinstance(other, MyStructType):
11           raise TypingError(
12               f"*other* must be a {MyStructType}; got {other}"
13           )
14
15       def impl(self, other):
```

(continues on next page)

```
16          return np.linalg.norm(self.vector - other.vector)
17
18      return impl
19
20  # Test
21  @njit
22  def test():
23      alice = MyStruct("Alice", vector=np.random.random(3))
24      bob = MyStruct("Bob", vector=np.random.random(3))
25      # Use the method
26      return alice.distance(bob)
```

`numba.experimental.structref` API Reference

Utilities for defining a mutable struct.

A mutable struct is passed by reference; hence, structref (a reference to a struct).

class `numba.experimental.structref.`**`StructRefProxy`**
    A PyObject proxy to the Numba allocated structref data structure.

Notes

- Subclasses should not define __init__.
- Subclasses can override __new__.

`numba.experimental.structref.`**`define_attributes`**(*struct_typeclass*)
    Define attributes on *struct_typeclass*.

    Defines both setters and getters in jit-code.

    This is called directly in *register()*.

`numba.experimental.structref.`**`define_boxing`**(*struct_type*, *obj_class*)
    Define the boxing & unboxing logic for *struct_type* to *obj_class*.

    Defines both boxing and unboxing.

    - boxing turns an instance of *struct_type* into a PyObject of *obj_class*
    - unboxing turns an instance of *obj_class* into an instance of *struct_type* in jit-code.

    Use this directly instead of *define_proxy()* when the user does not want any constructor to be defined.

`numba.experimental.structref.`**`define_constructor`**(*py_class*, *struct_typeclass*, *fields*)
    Define the jit-code constructor for *struct_typeclass* using the Python type *py_class* and the required *fields*.

    Use this instead of *define_proxy()* if the user does not want boxing logic defined.

`numba.experimental.structref.`**`define_proxy`**(*py_class*, *struct_typeclass*, *fields*)
    Defines a PyObject proxy for a structref.

    This makes *py_class* a valid constructor for creating a instance of *struct_typeclass* that contains the members as defined by *fields*.

        Parameters

            py_class [type] The Python class for constructing an instance of *struct_typeclass*.

> > > **struct_typeclass** [numba.core.types.Type] The structref type class to bind to.
> >
> > **fields** [Sequence[str]] A sequence of field names.
>
> **Returns**
>
> > **None**

numba.experimental.structref.**register**(*struct_type*)

> Register a *numba.core.types.StructRef* for use in jit-code.
>
> This defines the data-model for lowering an instance of *struct_type*. This defines attributes accessor and mutator for an instance of *struct_type*.
>
> > **Parameters**
> >
> > > **struct_type** [type] A subclass of *numba.core.types.StructRef*.
> >
> > **Returns**
> >
> > > **struct_type** [type] Returns the input argument so this can act like a decorator.

> **Examples**

6.2 Low-level extension API

This extension API is available through the `numba.extending` module. It allows you to hook directly into the Numba compilation chain. As such, it distinguished between several compilation phases:

- The *typing* phase deduces the types of variables in a compiled function by looking at the operations performed.

- The *lowering* phase converts high-level Python operations into low-level LLVM code. This phase exploits the typing information derived by the typing phase.

- *Boxing* and *unboxing* convert Python objects into native values, and vice-versa. They occur at the boundaries of calling a Numba function from the Python interpreter.

6.2.1 Typing

Type inference – or simply *typing* – is the process of assigning Numba types to all values involved in a function, so as to enable efficient code generation. Broadly speaking, typing comes in two flavours: typing plain Python *values* (e.g. function arguments or global variables) and typing *operations* (or *functions*) on known value types.

@typeof_impl.**register**(*cls*)

> Register the decorated function as typing Python values of class *cls*. The decorated function will be called with the signature (`val, c`) where *val* is the Python value being typed and *c* is a context object.

@**type_callable**(*func*)

> Register the decorated function as typing the callable *func*. *func* can be either an actual Python callable or a string denoting a operation internally known to Numba (for example `'getitem'`). The decorated function is called with a single *context* argument and must return a typer function. The typer function should have the same signature as the function being typed, and it is called with the Numba *types* of the function arguments; it should return either the Numba type of the function's return value, or `None` if inference failed.

6.2.2 Lowering

The following decorators all take a type specification of some kind. A type specification is usually a type class (such as `types.Float`) or a specific type instance (such as `types.float64`). Some values have a special meaning:

- `types.Any` matches any type; this allows doing your own dispatching inside the implementation
- `types.VarArg(<some type>)` matches any number of arguments of the given type; it can only appear as the last type specification when describing a function's arguments.

A *context* argument in the following APIs is a target context providing various utility methods for code generation (such as creating a constant, converting from a type to another, looking up the implementation of a specific function, etc.). A *builder* argument is a `llvmlite.ir.IRBuilder` instance for the LLVM code being generated.

A *signature* is an object specifying the concrete type of an operation. The `args` attribute of the signature is a tuple of the argument types. The `return_type` attribute of the signature is the type that the operation should return.

Note: Numba always reasons on Numba types, but the values being passed around during lowering are LLVM values: they don't hold the required type information, which is why Numba types are passed explicitly too.

LLVM has its own, very low-level type system: you can access the LLVM type of a value by looking up its `.type` attribute.

Native operations

@**lower_builtin**(*func*, *typespec*, *...*)
> Register the decorated function as implementing the callable *func* for the arguments described by the given Numba *typespecs*. As with `type_callable()`, *func* can be either an actual Python callable or a string denoting a operation internally known to Numba (for example `'getitem'`).
>
> The decorated function is called with four arguments (`context, builder, sig, args`). `sig` is the concrete signature the callable is being invoked with. `args` is a tuple of the values of the arguments the callable is being invoked with; each value in `args` corresponds to a type in `sig.args`. The function must return a value compatible with the type `sig.return_type`.

@**lower_getattr**(*typespec*, *name*)
> Register the decorated function as implementing the attribute *name* of the given *typespec*. The decorated function is called with four arguments (`context, builder, typ, value`). *typ* is the concrete type the attribute is being looked up on. *value* is the value the attribute is being looked up on.

@**lower_getattr_generic**(*typespec*)
> Register the decorated function as a fallback for attribute lookup on a given *typespec*. Any attribute that does not have a corresponding `lower_getattr()` declaration will go through `lower_getattr_generic()`. The decorated function is called with five arguments (`context, builder, typ, value, name`). *typ* and *value* are as in `lower_getattr()`. *name* is the name of the attribute being looked up.

@**lower_cast**(*fromspec*, *tospec*)
> Register the decorated function as converting from types described by *fromspec* to types described by *tospec*. The decorated function is called with five arguments (`context, builder, fromty, toty, value`). *fromty* and *toty* are the concrete types being converted from and to, respectively. *value* is the value being converted. The function must return a value compatible with the type `toty`.

Constants

@**lower_constant**(*typespec*)
> Register the decorated function as implementing the creation of constants for the Numba *typespec*. The decorated function is called with four arguments (`context, builder, ty, pyval`). *ty* is the concrete type to create a constant for. *pyval* is the Python value to convert into a LLVM constant. The function must return a value compatible with the type `ty`.

Boxing and unboxing

In these functions, *c* is a convenience object with several attributes:

- its `context` attribute is a target context as above
- its `builder` attribute is a `llvmlite.ir.IRBuilder` as above
- its `pyapi` attribute is an object giving access to a subset of the Python interpreter's C API

An object, as opposed to a native value, is a `PyObject *` pointer. Such pointers can be produced or processed by the methods in the `pyapi` object.

@**box**(*typespec*)
> Register the decorated function as boxing values matching the *typespec*. The decorated function is called with three arguments (`typ, val, c`). *typ* is the concrete type being boxed. *val* is the value being boxed. The function should return a Python object, or NULL to signal an error.

@**unbox**(*typespec*)
> Register the decorated function as unboxing values matching the *typespec*. The decorated function is called with three arguments (`typ, obj, c`). *typ* is the concrete type being unboxed. *obj* is the Python object (a `PyObject *` pointer, in C terms) being unboxed. The function should return a `NativeValue` object giving the unboxing result value and an optional error bit.

6.3 Example: an interval type

We will extend the Numba frontend to support a class that it does not currently support so as to allow:

- Passing an instance of the class to a Numba function
- Accessing attributes of the class in a Numba function
- Constructing and returning a new instance of the class from a Numba function

(all the above in *nopython mode*)

We will mix APIs from the *high-level extension API* and the *low-level extension API*, depending on what is available for a given task.

The starting point for our example is the following pure Python class:

```python
class Interval(object):
    """
    A half-open interval on the real number line.
    """
    def __init__(self, lo, hi):
        self.lo = lo
        self.hi = hi

    def __repr__(self):
        return 'Interval(%f, %f)' % (self.lo, self.hi)

    @property
    def width(self):
        return self.hi - self.lo
```

6.3.1 Extending the typing layer

Creating a new Numba type

As the `Interval` class is not known to Numba, we must create a new Numba type to represent instances of it. Numba does not deal with Python types directly: it has its own type system that allows a different level of granularity as well as various meta-information not available with regular Python types.

We first create a type class `IntervalType` and, since we don't need the type to be parametric, we instantiate a single type instance `interval_type`:

```python
from numba import types

class IntervalType(types.Type):
    def __init__(self):
        super(IntervalType, self).__init__(name='Interval')

interval_type = IntervalType()
```

Type inference for Python values

In itself, creating a Numba type doesn't do anything. We must teach Numba how to infer some Python values as instances of that type. In this example, it is trivial: any instance of the `Interval` class should be treated as belonging to the type `interval_type`:

```python
from numba.extending import typeof_impl

@typeof_impl.register(Interval)
def typeof_index(val, c):
    return interval_type
```

Function arguments and global values will thusly be recognized as belonging to `interval_type` whenever they are instances of `Interval`.

Type inference for operations

We want to be able to construct interval objects from Numba functions, so we must teach Numba to recognize the two-argument `Interval(lo, hi)` constructor. The arguments should be floating-point numbers:

```python
from numba.extending import type_callable

@type_callable(Interval)
def type_interval(context):
    def typer(lo, hi):
        if isinstance(lo, types.Float) and isinstance(hi, types.Float):
            return interval_type
    return typer
```

The `type_callable()` decorator specifies that the decorated function should be invoked when running type inference for the given callable object (here the `Interval` class itself). The decorated function must simply return a typer function that will be called with the argument types. The reason for this seemingly convoluted setup is for the typer function to have *exactly* the same signature as the typed callable. This allows handling keyword arguments correctly.

The *context* argument received by the decorated function is useful in more sophisticated cases where computing the callable's return type requires resolving other types.

6.3.2 Extending the lowering layer

We have finished teaching Numba about our type inference additions. We must now teach Numba how to actually generated code and data for the new operations.

Defining the data model for native intervals

As a general rule, *nopython mode* does not work on Python objects as they are generated by the CPython interpreter. The representations used by the interpreter are far too inefficient for fast native code. Each type supported in *nopython mode* therefore has to define a tailored native representation, also called a *data model*.

A common case of data model is an immutable struct-like data model, that is akin to a C `struct`. Our interval datatype conveniently falls in that category, and here is a possible data model for it:

```python
from numba.extending import models, register_model

@register_model(IntervalType)
class IntervalModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ('lo', types.float64),
            ('hi', types.float64),
            ]
        models.StructModel.__init__(self, dmm, fe_type, members)
```

This instructs Numba that values of type `IntervalType` (or any instance thereof) are represented as a structure of two fields `lo` and `hi`, each of them a double-precision floating-point number (`types.float64`).

Note: Mutable types need more sophisticated data models to be able to persist their values after modification. They typically cannot be stored and passed on the stack or in registers like immutable types do.

Exposing data model attributes

We want the data model attributes `lo` and `hi` to be exposed under the same names for use in Numba functions. Numba provides a convenience function to do exactly that:

```python
from numba.extending import make_attribute_wrapper

make_attribute_wrapper(IntervalType, 'lo', 'lo')
make_attribute_wrapper(IntervalType, 'hi', 'hi')
```

This will expose the attributes in read-only mode. As mentioned above, writable attributes don't fit in this model.

Exposing a property

As the `width` property is computed rather than stored in the structure, we cannot simply expose it like we did for `lo` and `hi`. We have to re-implement it explicitly:

```python
from numba.extending import overload_attribute

@overload_attribute(IntervalType, "width")
def get_width(interval):
```

```
    def getter(interval):
        return interval.hi - interval.lo
    return getter
```

You might ask why we didn't need to expose a type inference hook for this attribute? The answer is that `@overload_attribute` is part of the high-level API: it combines type inference and code generation in a single API.

Implementing the constructor

Now we want to implement the two-argument `Interval` constructor:

```python
from numba.extending import lower_builtin
from numba.core import cgutils

@lower_builtin(Interval, types.Float, types.Float)
def impl_interval(context, builder, sig, args):
    typ = sig.return_type
    lo, hi = args
    interval = cgutils.create_struct_proxy(typ)(context, builder)
    interval.lo = lo
    interval.hi = hi
    return interval._getvalue()
```

There is a bit more going on here. `@lower_builtin` decorates the implementation of the given callable or operation (here the `Interval` constructor) for some specific argument types. This allows defining type-specific implementations of a given operation, which is important for heavily overloaded functions such as `len()`.

`types.Float` is the class of all floating-point types (`types.float64` is an instance of `types.Float`). It is generally more future-proof to match argument types on their class rather than on specific instances (however, when *returning* a type – chiefly during the type inference phase –, you must usually return a type instance).

`cgutils.create_struct_proxy()` and `interval._getvalue()` are a bit of boilerplate due to how Numba passes values around. Values are passed as instances of `llvmlite.ir.Value`, which can be too limited: LLVM structure values especially are quite low-level. A struct proxy is a temporary wrapper around a LLVM structure value allowing to easily get or set members of the structure. The `_getvalue()` call simply gets the LLVM value out of the wrapper.

Boxing and unboxing

If you try to use an `Interval` instance at this point, you'll certainly get the error *"cannot convert Interval to native value"*. This is because Numba doesn't yet know how to make a native interval value from a Python `Interval` instance. Let's teach it how to do it:

```python
from numba.extending import unbox, NativeValue

@unbox(IntervalType)
def unbox_interval(typ, obj, c):
    """
    Convert a Interval object to a native interval structure.
    """
    lo_obj = c.pyapi.object_getattr_string(obj, "lo")
    hi_obj = c.pyapi.object_getattr_string(obj, "hi")
    interval = cgutils.create_struct_proxy(typ)(c.context, c.builder)
```

```python
    interval.lo = c.pyapi.float_as_double(lo_obj)
    interval.hi = c.pyapi.float_as_double(hi_obj)
    c.pyapi.decref(lo_obj)
    c.pyapi.decref(hi_obj)
    is_error = cgutils.is_not_null(c.builder, c.pyapi.err_occurred())
    return NativeValue(interval._getvalue(), is_error=is_error)
```

Unbox is the other name for "convert a Python object to a native value" (it fits the idea of a Python object as a sophisticated box containing a simple native value). The function returns a `NativeValue` object which gives its caller access to the computed native value, the error bit and possibly other information.

The snippet above makes abundant use of the `c.pyapi` object, which gives access to a subset of the Python interpreter's C API. Note the use of `c.pyapi.err_occurred()` to detect any errors that may have happened when unboxing the object (try passing `Interval('a', 'b')` for example).

We also want to do the reverse operation, called *boxing*, so as to return interval values from Numba functions:

```python
from numba.extending import box


@box(IntervalType)
def box_interval(typ, val, c):
    """
    Convert a native interval structure to an Interval object.
    """
    interval = cgutils.create_struct_proxy(typ)(c.context, c.builder, value=val)
    lo_obj = c.pyapi.float_from_double(interval.lo)
    hi_obj = c.pyapi.float_from_double(interval.hi)
    class_obj = c.pyapi.unserialize(c.pyapi.serialize_object(Interval))
    res = c.pyapi.call_function_objargs(class_obj, (lo_obj, hi_obj))
    c.pyapi.decref(lo_obj)
    c.pyapi.decref(hi_obj)
    c.pyapi.decref(class_obj)
    return res
```

6.3.3 Using it

nopython mode functions are now able to make use of Interval objects and the various operations you have defined on them. You can try for example the following functions:

```python
from numba import jit


@jit(nopython=True)
def inside_interval(interval, x):
    return interval.lo <= x < interval.hi


@jit(nopython=True)
def interval_width(interval):
    return interval.width


@jit(nopython=True)
def sum_intervals(i, j):
    return Interval(i.lo + j.lo, i.hi + j.hi)
```

6.3.4 Conclusion

We have shown how to do the following tasks:

- Define a new Numba type class by subclassing the `Type` class

- Define a singleton Numba type instance for a non-parametric type

- Teach Numba how to infer the Numba type of Python values of a certain class, using `typeof_impl.`
  `register`

- Define the data model for a Numba type using `StructModel` and `register_model`

- Implementing a boxing function for a Numba type using the `@box` decorator

- Implementing an unboxing function for a Numba type using the `@unbox` decorator and the `NativeValue`
  class

- Type and implement a callable using the `@type_callable` and `@lower_builtin` decorators

- Expose a read-only structure attribute using the `make_attribute_wrapper` convenience function

- Implement a read-only property using the `@overload_attribute` decorator

6.4 A guide to using `@overload`

As mentioned in the *high-level extension API*, you can use the `@overload` decorator to create a Numba implementation of a function that can be used in *nopython mode* functions. A common use case is to re-implement NumPy functions so that they can be called in `@jit` decorated code. This section discusses how and when to use the `@overload` decorator and what contributing such a function to the Numba code base might entail. This should help you get started when needing to use the `@overload` decorator or when attempting to contribute new functions to Numba itself.

The `@overload` decorator and it's variants are useful when you have a third-party library that you do not control and you wish to provide Numba compatible implementations for specific functions from that library.

6.4.1 Concrete Example

Let's assume that you are working on a minimization algorithm that makes use of `scipy.linalg.norm` to find different vector norms and the frobenius norm for matrices. You know that only integer and real numbers will be involved. (While this may sound like an artificial example, especially because a Numba implementation of `numpy.`
`linalg.norm` exists, it is largely pedagogical and serves to illustrate how and when to use `@overload`).

The skeleton might look something like this:

```python
def algorithm():
    # setup
    v = ...
    while True:
        # take a step
        d = scipy.linalg.norm(v)
        if d < tolerance:
            break
```

Now, let's further assume, that you have heard of Numba and you now wish to use it to accelerate your function. However, after adding the `jit(nopython=True)` decorator, Numba complains that `scipy.linalg.norm` isn't supported. From looking at the documentation, you realize that a norm is probably fairly easy to implement using NumPy. A good starting point is the following template.

```
# Declare that function `myfunc` is going to be overloaded (have a
# substitutable Numba implementation)
@overload(myfunc)
# Define the overload function with formal arguments
# these arguments must be matched in the inner function implementation
def jit_myfunc(arg0, arg1, arg2, ...):
    # This scope is for typing, access is available to the *type* of all
    # arguments. This information can be used to change the behaviour of the
    # implementing function and check that the types are actually supported
    # by the implementation.

    print(arg0) # this will show the Numba type of arg0

    # This is the definition of the function that implements the `myfunc` work.
    # It does whatever algorithm is needed to implement myfunc.
    def myfunc_impl(arg0, arg1, arg2, ...): # match arguments to jit_myfunc
        # < Implementation goes here >
        return # whatever needs to be returned by the algorithm

    # return the implementation
    return myfunc_impl
```

After some deliberation and tinkering, you end up with the following code:

```
import numpy as np
from numba import njit, types
from numba.extending import overload, register_jitable
from numba.core.errors import TypingError

import scipy.linalg


@register_jitable
def _oneD_norm_2(a):
    # re-usable implementation of the 2-norm
    val = np.abs(a)
    return np.sqrt(np.sum(val * val))


@overload(scipy.linalg.norm)
def jit_norm(a, ord=None):
    if isinstance(ord, types.Optional):
        ord = ord.type
    # Reject non integer, floating-point or None types for ord
    if not isinstance(ord, (types.Integer, types.Float, types.NoneType)):
        raise TypingError("'ord' must be either integer or floating-point")
    # Reject non-ndarray types
    if not isinstance(a, types.Array):
        raise TypingError("Only accepts NumPy ndarray")
    # Reject ndarrays with non integer or floating-point dtype
    if not isinstance(a.dtype, (types.Integer, types.Float)):
        raise TypingError("Only integer and floating point types accepted")
    # Reject ndarrays with unsupported dimensionality
    if not (0 <= a.ndim <= 2):
        raise TypingError('3D and beyond are not allowed')
    # Implementation for scalars/0d-arrays
    elif a.ndim == 0:
```

(continues on next page)

```python
            return a.item()
    # Implementation for vectors
    elif a.ndim == 1:
        def _oneD_norm_x(a, ord=None):
            if ord == 2 or ord is None:
                return _oneD_norm_2(a)
            elif ord == np.inf:
                return np.max(np.abs(a))
            elif ord == -np.inf:
                return np.min(np.abs(a))
            elif ord == 0:
                return np.sum(a != 0)
            elif ord == 1:
                return np.sum(np.abs(a))
            else:
                return np.sum(np.abs(a)**ord)**(1. / ord)
        return _oneD_norm_x
    # Implementation for matrices
    elif a.ndim == 2:
        def _two_D_norm_2(a, ord=None):
            return _oneD_norm_2(a.ravel())
        return _two_D_norm_2


if __name__ == "__main__":
    @njit
    def use(a, ord=None):
        # simple test function to check that the overload works
        return scipy.linalg.norm(a, ord)

    # spot check for vectors
    a = np.arange(10)
    print(use(a))
    print(scipy.linalg.norm(a))

    # spot check for matrices
    b = np.arange(9).reshape((3, 3))
    print(use(b))
    print(scipy.linalg.norm(b))
```

As you can see, the implementation only supports what you need right now:

- Only supports integer and floating-point types

- All vector norms

- Only the Frobenius norm for matrices

- Code sharing between vector and matrix implementations using `@register_jitable`.

- Norms are implemented using NumPy syntax. (This is possible because Numba is very aware of NumPy and many functions are supported.)

So what actually happens here? The `overload` decorator registers a suitable implementation for `scipy.linalg.norm` in case a call to this is encountered in code that is being JIT-compiled, for example when you decorate your `algorithm` function with `@jit(nopython=True)`. In that case, the function `jit_norm` will be called with the currently encountered types and will then return either `_oneD_norm_x` in the vector case and `_two_D_norm_2`.

You can download the example code here: `mynorm.py`

6.4.2 Implementing `@overload` for NumPy functions

Numba supports NumPy through the provision of `@jit` compatible re-implementations of NumPy functions. In such cases `@overload` is a very convenient option for writing such implementations, however there are a few additional things to watch out for.

- The Numba implementation should match the NumPy implementation as closely as feasible with respect to accepted types, arguments, raised exceptions and algorithmic complexity (Big-O / Landau order).

- When implementing supported argument types, bear in mind that, due to duck typing, NumPy does tend to accept a multitude of argument types beyond NumPy arrays such as scalar, list, tuple, set, iterator, generator etc. You will need to account for that during type inference and subsequently as part of the tests.

- A NumPy function may return a scalar, array or a data structure which matches one of its inputs, you need to be aware of type unification problems and dispatch to appropriate implementations. For example, `np.corrcoef` may return an array or a scalar depending on its inputs.

- If you are implementing a new function, you should always update the documentation. The sources can be found in `docs/source/reference/numpysupported.rst`. Be sure to mention any limitations that your implementation has, e.g. no support for the `axis` keyword.

- When writing tests for the functionality itself, it's useful to include handling of non-finite values, arrays with different shapes and layouts, complex inputs, scalar inputs, inputs with types for which support is not documented (e.g. a function which the NumPy docs say requires a float or int input might also 'work' if given a bool or complex input).

- When writing tests for exceptions, for example if adding tests to `numba/tests/test_np_functions.py`, you may encounter the following error message:

```
======================================================================
FAIL: test_foo (numba.tests.test_np_functions.TestNPFunctions)
----------------------------------------------------------------------
Traceback (most recent call last):
File "<path>/numba/numba/tests/support.py", line 645, in tearDown
    self.memory_leak_teardown()
File "<path>/numba/numba/tests/support.py", line 619, in memory_leak_teardown
    self.assert_no_memory_leak()
File "<path>/numba/numba/tests/support.py", line 628, in assert_no_memory_leak
    self.assertEqual(total_alloc, total_free)
AssertionError: 36 != 35
```

  This occurs because raising exceptions from jitted code leads to reference leaks. Ideally, you will place all exception testing in a separate test method and then add a call in each test to `self.disable_leak_check()` to disable the leak-check (inherit from `numba.tests.support.TestCase` to make that available).

- For many of the functions that are available in NumPy, there are corresponding methods defined on the NumPy `ndarray` type. For example, the function `repeat` is available as a NumPy module level function and a member function on the `ndarray` class.

```python
import numpy as np
a = np.arange(10)
# function
np.repeat(a, 10)
# method
a.repeat(10)
```

  Once you have written the function implementation, you can easily use `@overload_method` and reuse it. Just be sure to check that NumPy doesn't diverge in the implementations of its function/method.

  As an example, the `repeat` function/method:

```
@extending.overload_method(types.Array, 'repeat')
def array_repeat(a, repeats):
    def array_repeat_impl(a, repeat):
        # np.repeat has already been overloaded
        return np.repeat(a, repeat)

    return array_repeat_impl
```

- If you need to create ancillary functions, for example to re-use a small utility function or to split your implementation across functions for the sake of readability, you can make use of the `@register_jitable` decorator. This will make those functions available from within your `@jit` and `@overload` decorated functions.

- The Numba continuous integration (CI) set up tests a wide variety of NumPy versions, you'll sometimes be alerted to a change in behaviour from some previous NumPy version. If you can find supporting evidence in the NumPy change log / repository, then you'll need to decide whether to create branches and attempt to replicate the logic across versions, or use a version gate (with associated wording in the documentation) to advertise that Numba replicates NumPy from some particular version onwards.

- You can look at the Numba source code for inspiration, many of the overloaded NumPy functions and methods are in `numba/targets/arrayobj.py`. Below, you will find a list of implementations to look at that are well implemented in terms of accepted types and test coverage.

  - `np.repeat`

6.5 Registering Extensions with Entry Points

Often, third party packages will have a user-facing API as well as define extensions to the Numba compiler. In those situations, the new types and overloads can registered with Numba when the package is imported by the user. However, there are situations where a Numba extension would not normally be imported directly by the user, but must still be registered with the Numba compiler. An example of this is the numba-scipy package, which adds support for some SciPy functions to Numba. The end user does not need to `import numba_scipy` to enable compiler support for SciPy, the extension only needs to be installed in the Python environment.

Numba discovers extensions using the entry points feature of `setuptools`. This allows a Python package to register an initializer function that will be called before `numba` compiles for the first time. The delay ensures that the cost of importing extensions is deferred until it is necessary.

6.5.1 Adding Support for the "Init" Entry Point

A package can register an initialization function with Numba by adding the `entry_points` argument to the `setup()` function call in `setup.py`:

```
setup(
    ...,
    entry_points={
        "numba_extensions": [
            "init = numba_scipy:_init_extension",
        ],
    },
    ...
)
```

Numba currently only looks for the `init` entry point in the `numba_extensions` group. The entry point should be a function (any name, as long as it matches what is listed in `setup.py`) that takes no arguments, and the return

value is ignored. This function should register types, overloads, or call other Numba extension APIs. The order of initialization of extensions is undefined.

6.5.2 Testing your Entry Point

Numba loads all entry points when the first function is compiled. To test your entry point, it is not sufficient to just `import numba`; you have to define and run a small function, like this:

```python
import numba; numba.njit(lambda x: x + 1)(123)
```

It is not necessary to import your module: entry points are identified by the `entry_points.txt` file in your library's `*.egg-info` directory.

The `setup.py build` command does not create eggs, but `setup.py sdist` (for testing in a local directory) and `setup.py install` do. All entry points registered in eggs that are on the Python path are loaded. Be sure to check for stale `entry_points.txt` when debugging.

Developer Manual

7.1 Contributing to Numba

We welcome people who want to make contributions to Numba, big or small! Even simple documentation improvements are encouraged. If you have questions, don't hesitate to ask them (see below).

7.1.1 Communication

Real-time Chat

Numba uses Gitter for public real-time chat. To help improve the signal-to-noise ratio, we have two channels:

- numba/numba: General Numba discussion, questions, and debugging help.
- numba/numba-dev: Discussion of PRs, planning, release coordination, etc.

Both channels are public, but we may ask that discussions on numba-dev move to the numba channel. This is simply to ensure that numba-dev is easy for core developers to keep up with.

Note that the Github issue tracker is the best place to report bugs. Bug reports in chat are difficult to track and likely to be lost.

Forum

Numba uses Discourse as a forum for longer running threads such as design discussions and roadmap planning. There are various categories available and it can be reached at: numba.discourse.group.

Mailing-list

We have a public mailing-list that you can e-mail at numba-users@anaconda.com. You can subscribe and read the archives on Google Groups.

Weekly Meetings

The core Numba developers have a weekly video conference to discuss roadmap, feature planning, and outstanding issues. These meetings are invite only, but minutes will be taken and will be posted to the Numba wiki.

Bug tracker

We use the Github issue tracker to track both bug reports and feature requests. If you report an issue, please include specifics:

- what you are trying to do;
- which operating system you have and which version of Numba you are running;
- how Numba is misbehaving, e.g. the full error traceback, or the unexpected results you are getting;
- as far as possible, a code snippet that allows full reproduction of your problem.

7.1.2 Getting set up

If you want to contribute, we recommend you fork our Github repository, then create a branch representing your work. When your work is ready, you should submit it as a pull request from the Github interface.

If you want, you can submit a pull request even when you haven't finished working. This can be useful to gather feedback, or to stress your changes against the *continuous integration* platform. In this case, please prepend `[WIP]` to your pull request's title.

Build environment

Numba has a number of dependencies (mostly NumPy and llvmlite) with non-trivial build instructions. Unless you want to build those dependencies yourself, we recommend you use conda to create a dedicated development environment and install precompiled versions of those dependencies there.

First add the Anaconda Cloud `numba` channel so as to get development builds of the llvmlite library:

```
$ conda config --add channels numba
```

Then create an environment with the right dependencies:

```
$ conda create -n numbaenv python=3.6 llvmlite numpy scipy jinja2 cffi
```

Note: This installs an environment based on Python 3.6, but you can of course choose another version supported by Numba. To test additional features, you may also need to install `tbb` and/or `llvm-openmp` and `intel-openmp`.

To activate the environment for the current shell session:

```
$ conda activate numbaenv
```

Note: These instructions are for a standard Linux shell. You may need to adapt them for other platforms.

Once the environment is activated, you have a dedicated Python with the required dependencies:

```
$ python
Python 3.6.6 |Anaconda, Inc.| (default, Jun 28 2018, 11:07:29)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import llvmlite
>>> llvmlite.__version__
'0.24.0'
```

Building Numba

For a convenient development workflow, we recommend you build Numba inside its source checkout:

```
$ git clone git://github.com/numba/numba.git
$ cd numba
$ python setup.py build_ext --inplace
```

This assumes you have a working C compiler and runtime on your development system. You will have to run this command again whenever you modify C files inside the Numba source tree.

Running tests

Numba is validated using a test suite comprised of various kind of tests (unit tests, functional tests). The test suite is written using the standard `unittest` framework.

The tests can be executed via `python -m numba.runtests`. If you are running Numba from a source checkout, you can type `./runtests.py` as a shortcut. Various options are supported to influence test running and reporting. Pass `-h` or `--help` to get a glimpse at those options. Examples:

- to list all available tests:

  ```
  $ python -m numba.runtests -l
  ```

- to list tests from a specific (sub-)suite:

  ```
  $ python -m numba.runtests -l numba.tests.test_usecases
  ```

- to run those tests:

  ```
  $ python -m numba.runtests numba.tests.test_usecases
  ```

- to run all tests in parallel, using multiple sub-processes:

  ```
  $ python -m numba.runtests -m
  ```

- For a detailed list of all options:

  ```
  $ python -m numba.runtests -h
  ```

The numba test suite can take a long time to complete. When you want to avoid the long wait, it is useful to focus on the failing tests first with the following test runner options:

- The `--failed-first` option is added to capture the list of failed tests and to re-execute them first:

  ```
  $ python -m numba.runtests --failed-first -m -v -b
  ```

- The `--last-failed` option is used with `--failed-first` to execute the previously failed tests only:

```
$ python -m numba.runtests --last-failed -m -v -b
```

When debugging, it is useful to turn on logging. Numba logs using the standard `logging` module. One can use the standard ways (i.e. `logging.basicConfig`) to configure the logging behavior. To enable logging in the test runner, there is a `--log` flag for convenience:

```
$ python -m numba.runtests --log
```

7.1.3 Development rules

Code reviews

Any non-trivial change should go through a code review by one or several of the core developers. The recommended process is to submit a pull request on github.

A code review should try to assess the following criteria:

- general design and correctness
- code structure and maintainability
- coding conventions
- docstrings, comments
- test coverage

Coding conventions

All Python code should follow PEP 8. Our C code doesn't have a well-defined coding style (would it be nice to follow PEP 7?). Code and documentation should generally fit within 80 columns, for maximum readability with all existing tools (such as code review UIs).

Numba uses Flake8 to ensure a consistent Python code format throughout the project. `flake8` can be installed with `pip` or `conda` and then run from the root of the Numba repository:

```
flake8 numba
```

Optionally, you may wish to setup pre-commit hooks to automatically run `flake8` when you make a git commit. This can be done by installing `pre-commit`:

```
pip install pre-commit
```

and then running:

```
pre-commit install
```

from the root of the Numba repository. Now `flake8` will be run each time you commit changes. You can skip this check with `git commit --no-verify`.

Numba has started the process of using type hints in its code base. This will be a gradual process of extending the number of files that use type hints, as well as going from voluntary to mandatory type hints for new features. Mypy is used for automated static checking.

At the moment, only certain files are checked by mypy. The list can be found in `mypy.ini`. When making changes to those files, it is necessary to add the required type hints such that mypy tests will pass. Only in exceptional circumstances should `type:  ignore` comments be used.

If you are contributing a new feature, we encourage you to use type hints, even if the file is not currently in the checklist. If you want to contribute type hints to enable a new file to be in the checklist, please add the file to the `files` variable in `mypy.ini`, and decide what level of compliance you are targetting. Level 3 is basic static checks, while levels 2 and 1 represent stricter checking. The levels are described in details in `mypy.ini`.

There is potential for confusion between the Numba module `typing` and Python built-in module `typing` used for type hints, as well as between Numba types—such as `Dict` or `Literal`—and `typing` types of the same name. To mitigate the risk of confusion we use a naming convention by which objects of the built-in `typing` module are imported with an `pt` prefix. For example, `typing.Dict` is imported as `from typing import Dict as ptDict`.

Stability

The repository's `master` branch is expected to be stable at all times. This translates into the fact that the test suite passes without errors on all supported platforms (see below). This also means that a pull request also needs to pass the test suite before it is merged in.

Platform support

Every commit to the master branch is automatically tested on all of the platforms Numba supports. This includes ARMv7, ARMv8, POWER8, as well as both AMD and NVIDIA GPUs. The build system however is internal to Anaconda, so we also use Travis CI and Azure to provide public continuous integration information for as many combinations as can be supported by the service. Travis CI automatically tests all pull requests on OS X and Linux, as well as a sampling of different Python and NumPy versions, Azure does the same but also includes Windows. If you see problems on platforms you are unfamiliar with, feel free to ask for help in your pull request. The Numba core developers can help diagnose cross-platform compatibility issues.

7.1.4 Documentation

The Numba documentation is split over two repositories:

- This documentation is in the `docs` directory inside the Numba repository.

- The Numba homepage has its sources in a separate repository at https://github.com/numba/numba-webpage

Main documentation

This documentation is under the `docs` directory of the Numba repository. It is built with Sphinx and numpydoc, which are available using conda or pip; i.e. `conda install sphinx numpydoc`.

To build the documentation, you need the bootstrap theme:

```
$ pip install sphinx_bootstrap_theme
```

You can edit the source files under `docs/source/`, after which you can build and check the documentation:

```
$ make html
$ open _build/html/index.html
```

Core developers can upload this documentation to the Numba website at http://numba.pydata.org by using the `gh-pages.py` script under `docs`:

```
$ python gh-pages.py version  # version can be 'dev' or '0.16' etc
```

then verify the repository under the `gh-pages` directory and use `git push`.

Web site homepage

The Numba homepage on http://numba.pydata.org can be fetched from here: https://github.com/numba/numba-webpage

After pushing documentation to a new version, core developers will want to update the website. Some notable files:

- `index.rst` # Update main page
- `_templates/sidebar_versions.html` # Update sidebar links
- `doc.rst` # Update after adding a new version for numba docs
- `download.rst` # Updata after uploading new numba version to pypi

After updating run:

```
$ make html
```

and check out `_build/html/index.html`. To push updates to the Web site:

```
$ python _scripts/gh-pages.py
```

then verify the repository under the `gh-pages` directory. Make sure the `CNAME` file is present and contains a single line for `numba.pydata.org`. Finally, use `git push` to update the website.

7.2 A Map of the Numba Repository

The Numba repository is quite large, and due to age has functionality spread around many locations. To help orient developers, this document will try to summarize where different categories of functionality can be found.

7.2.1 Support Files

Build and Packaging

- setup.py - Standard Python distutils/setuptools script
- MANIFEST.in - Distutils packaging instructions
- requirements.txt - Pip package requirements, not used by conda
- versioneer.py - Handles automatic setting of version in installed package from git tags
- .flake8 - Preferences for code formatting. Files should be fixed and removed from the exception list as time allows.
- .pre-commit-config.yaml - Configuration file for pre-commit hooks.
- .readthedocs.yml - Configuration file for Read the Docs.
- buildscripts/condarecipe.local - Conda build recipe
- buildscripts/condarecipe_clone_icc_rt - Recipe to build a standalone icc_rt package.

Continuous Integration

- azure-pipelines.yml - Azure Pipelines CI config (active: Win/Mac/Linux)

- buildscripts/azure/ - Azure Pipeline configuration for specific platforms

- buildscripts/appveyor/ - Appveyor build scripts

- buildscripts/incremental/ - Generic scripts for building Numba on various CI systems

- codecov.yml - Codecov.io coverage reporting

Documentation

- LICENSE - License for Numba

- LICENSES.third-party - License for third party code vendored into Numba

- README.rst - README for repo, also uploaded to PyPI

- CONTRIBUTING.md - Documentation on how to contribute to project (out of date, should be updated to point to Sphinx docs)

- CHANGE_LOG - History of Numba releases, also directly embedded into Sphinx documentation

- docs/ - Documentation source

- docs/_templates/ - Directory for templates (to override defaults with Sphinx theme)

- docs/Makefile - Used to build Sphinx docs with `make`

- docs/source - ReST source for Numba documentation

- docs/_static/ - Static CSS and image assets for Numba docs

- docs/gh-pages.py - Utility script to update Numba docs (stored as gh-pages)

- docs/make.bat - Not used (remove?)

- docs/requirements.txt - Pip package requirements for building docs with Read the Docs.

- numba/scripts/generate_lower_listing.py - Dump all registered implementations decorated with `@lower*` for reference documentation. Currently misses implementations from the higher level extension API.

7.2.2 Numba Source Code

Numba ships with both the source code and tests in one package.

- numba/ - all of the source code and tests

Public API

These define aspects of the public Numba interface.

- numba/core/decorators.py - User-facing decorators for compiling regular functions on the CPU

- numba/core/extending.py - Public decorators for extending Numba (`overload`, `intrinsic`, etc) - numba/experimental/structref.py - Public API for defining a mutable struct

- numba/core/ccallback.py - `@cfunc` decorator for compiling functions to a fixed C signature. Used to make callbacks.

- numba/np/ufunc/decorators.py - ufunc/gufunc compilation decorators

- numba/core/config.py - Numba global config options and environment variable handling
- numba/core/annotations - Gathering and printing type annotations of Numba IR
- numba/core/annotations/pretty_annotate.py - Code highlighting of Numba functions and types (both ANSI terminal and HTML)

Dispatching

- numba/core/dispatcher.py - Dispatcher objects are compiled functions produced by `@jit`. A dispatcher has different implementations for different type signatures.
- numba/_dispatcher.h | numba/_dispatcher.c - C interface to C++ dispatcher implementation
- numba/_dispatcherimpl.cpp - C++ dispatcher implementation (for speed on common data types)

Compiler Pipeline

- numba/core/compiler.py - Compiler pipelines and flags
- numba/core/errors.py - Numba exception and warning classes
- numba/core/ir.py - Numba IR data structure objects
- numba/core/bytecode.py - Bytecode parsing and function identity (??)
- numba/core/interpreter.py - Translate Python interpreter bytecode to Numba IR
- numba/core/analysis.py - Utility functions to analyze Numba IR (variable lifetime, prune branches, etc)
- numba/core/dataflow.py - Dataflow analysis for Python bytecode (used in analysis.py)
- numba/core/controlflow.py - Control flow analysis of Numba IR and Python bytecode
- numba/core/typeinfer.py - Type inference algorithm
- numba/core/transforms.py - Numba IR transformations
- numba/core/rewrites - Rewrite passes used by compiler
- numba/core/rewrites/__init__.py - Loads all rewrite passes so they are put into the registry
- numba/core/rewrites/registry.py - Registry object for collecting rewrite passes
- numba/core/rewrites/ir_print.py - Write print() calls into special print nodes in the IR
- numba/core/rewrites/static_raise.py - Converts exceptions with static arguments into a special form that can be lowered
- numba/core/rewrites/macros.py - Generic support for macro expansion in the Numba IR
- numba/core/rewrites/static_getitem.py - Rewrites getitem and setitem with constant arguments to allow type inference
- numba/core/rewrites/static_binop.py - Rewrites binary operations (specifically `**`) with constant arguments so faster code can be generated
- numba/core/inline_closurecall.py - Inlines body of closure functions to call site. Support for array comprehensions, reduction inlining, and stencil inlining.
- numba/core/postproc.py - Postprocessor for Numba IR that computes variable lifetime, inserts del operations, and handles generators
- numba/core/lowering.py - General implementation of lowering Numba IR to LLVM numba/core/environment.py - Runtime environment object

- numba/core/withcontexts.py - General scaffolding for implementing context managers in nopython mode, and the objectmode context manager
- numba/core/pylowering.py - Lowering of Numba IR in object mode
- numba/core/pythonapi.py - LLVM IR code generation to interface with CPython API

Type Management

- numba/core/typeconv/ - Implementation of type casting and type signature matching in both C++ and Python
- numba/capsulethunk.h - Used by typeconv
- numba/core/types/ - definition of the Numba type hierarchy, used everywhere in compiler to select implementations
- numba/core/consts.py - Constant inference (used to make constant values available during codegen when possible)
- numba/core/datamodel - LLVM IR representations of data types in different contexts
- numba/core/datamodel/models.py - Models for most standard types
- numba/core/datamodel/registry.py - Decorator to register new data models
- numba/core/datamodel/packer.py - Pack typed values into a data structure
- numba/core/datamodel/testing.py - Data model tests (this should move??)
- numba/core/datamodel/manager.py - Map types to data models

Compiled Extensions

Numba uses a small amount of compiled C/C++ code for core functionality, like dispatching and type matching where performance matters, and it is more convenient to encapsulate direct interaction with CPython APIs.

- numba/_arraystruct.h - Struct for holding NumPy array attributes. Used in helperlib and the Numba Runtime.
- numba/_helperlib.c - C functions required by Numba compiled code at runtime. Linked into ahead-of-time compiled modules
- numba/_helpermod.c - Python extension module with pointers to functions from `_helperlib.c` and `_npymath_exports.c`
- numba/_npymath_exports.c - Export function pointer table to NumPy C math functions
- numba/_dynfuncmod.c - Python extension module exporting _dynfunc.c functionality
- numba/_dynfunc.c - C level Environment and Closure objects (keep in sync with numba/target/base.py)
- numba/mathnames.h - Macros for defining names of math functions
- numba/_pymodule.h - C macros for Python 2/3 portable naming of C API functions
- numba/mviewbuf.c - Handles Python memoryviews
- numba/_typeof.h | numba/_typeof.c - C implementation of type fingerprinting, used by dispatcher
- numba/_numba_common.h - Portable C macro for marking symbols that can be shared between object files, but not outside the library.

Misc Support

- numba/_version.py - Updated by versioneer

- numba/core/runtime - Language runtime. Currently manages reference-counted memory allocated on the heap by Numba-compiled functions

- numba/core/ir_utils.py - Utility functions for working with Numba IR data structures

- numba/core/cgutils.py - Utility functions for generating common code patterns in LLVM IR

- numba/core/utils.py - Python 2 backports of Python 3 functionality (also imports local copy of `six`)

- numba/misc/appdirs.py - Vendored package for determining application config directories on every platform

- numba/core/compiler_lock.py - Global compiler lock because Numba's usage of LLVM is not thread-safe

- numba/misc/special.py - Python stub implementations of special Numba functions (prange, gdb*)

- numba/core/itanium_mangler.py - Python implementation of Itanium C++ name mangling

- numba/misc/findlib.py - Helper function for locating shared libraries on all platforms

- numba/core/debuginfo.py - Helper functions to construct LLVM IR debug info

- numba/core/unsafe/refcount.py - Read reference count of object

- numba/core/unsafe/eh.py - Exception handling helpers

- numba/core/unsafe/nrt.py - Numba runtime (NRT) helpers

- numba/cpython/unsafe/tuple.py - Replace a value in a tuple slot

- numba/np/unsafe/ndarray.py - NumPy array helpers

- numba/core/unsafe/bytes.py - Copying and dereferencing data from void pointers

- numba/misc/dummyarray.py - Used by GPU backends to hold array information on the host, but not the data.

- numba/core/callwrapper.py - Handles argument unboxing and releasing the GIL when moving from Python to nopython mode

- numba/np/numpy_support.py - Helper functions for working with NumPy and translating Numba types to and from NumPy dtypes.

- numba/core/tracing.py - Decorator for tracing Python calls and emitting log messages

- numba/core/funcdesc.py - Classes for describing function metadata (used in the compiler)

- numba/core/sigutils.py - Helper functions for parsing and normalizing Numba type signatures

- numba/core/serialize.py - Support for pickling compiled functions

- numba/core/caching.py - Disk cache for compiled functions

- numba/np/npdatetime.py - Helper functions for implementing NumPy datetime64 support

Core Python Data Types

- numba/_hashtable.h | numba/_hashtable.c - Adaptation of the Python 3.7 hash table implementation

- numba/cext/dictobject.h | numba/cext/dictobject.c - C level implementation of typed dictionary

- numba/typed/dictobject.py - Nopython mode wrapper for typed dictionary

- numba/cext/listobject.h | numba/cext/listobject.c - C level implementation of typed list

- numba/typed/listobject.py - Nopython mode wrapper for typed list

- numba/typed/typedobjectutils.py - Common utilities for typed dictionary and list
- numba/cpython/unicode.py - Unicode strings (Python 3.5 and later)
- numba/typed - Python interfaces to statically typed containers
- numba/typed/typeddict.py - Python interface to typed dictionary
- numba/typed/typedlist.py - Python interface to typed list
- numba/experimental/jitclass - Implementation of experimental JIT compilation of Python classes
- numba/core/generators.py - Support for lowering Python generators

Math

- numba/_random.c - Reimplementation of NumPy / CPython random number generator
- numba/_lapack.c - Wrappers for calling BLAS and LAPACK functions (requires SciPy)

ParallelAccelerator

Code transformation passes that extract parallelizable code from a function and convert it into multithreaded gufunc calls.

- numba/parfors/parfor.py - General ParallelAccelerator
- numba/parfors/parfor_lowering.py - gufunc lowering for ParallelAccelerator
- numba/parfors/array_analysis.py - Array analysis passes used in ParallelAccelerator

Stencil

Implementation of `@stencil`:

- numba/stencils/stencil.py - Stencil function decorator (implemented without ParallelAccelerator)
- numba/stencils/stencilparfor.py - ParallelAccelerator implementation of stencil

Debugging Support

- numba/misc/gdb_hook.py - Hooks to jump into GDB from nopython mode
- numba/misc/cmdlang.gdb - Commands to setup GDB for setting explicit breakpoints from Python

Type Signatures (CPU)

Some (usually older) Numba supported functionality separates the declaration of allowed type signatures from the definition of implementations. This package contains registries of type signatures that must be matched during type inference.

- numba/core/typing - Type signature module
- numba/core/typing/templates.py - Base classes for type signature templates
- numba/core/typing/cmathdecl.py - Python complex math (`cmath`) module
- numba/core/typing/bufproto.py - Interpreting objects supporting the buffer protocol

- numba/core/typing/mathdecl.py - Python `math` module
- numba/core/typing/listdecl.py - Python lists
- numba/core/typing/builtins.py - Python builtin global functions and operators
- numba/core/typing/randomdecl.py - Python and NumPy `random` modules
- numba/core/typing/setdecl.py - Python sets
- numba/core/typing/npydecl.py - NumPy ndarray (and operators), NumPy functions
- numba/core/typing/arraydecl.py - Python `array` module
- numba/core/typing/context.py - Implementation of typing context (class that collects methods used in type inference)
- numba/core/typing/collections.py - Generic container operations and namedtuples
- numba/core/typing/ctypes_utils.py - Typing ctypes-wrapped function pointers
- numba/core/typing/enumdecl.py - Enum types
- numba/core/typing/cffi_utils.py - Typing of CFFI objects
- numba/core/typing/typeof.py - Implementation of typeof operations (maps Python object to Numba type)
- numba/core/typing/npdatetime.py - Datetime dtype support for NumPy arrays

Target Implementations (CPU)

Implementations of Python / NumPy functions and some data models. These modules are responsible for generating LLVM IR during lowering. Note that some of these modules do not have counterparts in the typing package because newer Numba extension APIs (like overload) allow typing and implementation to be specified together.

- numba/core/cpu.py - Context for code gen on CPU
- numba/core/base.py - Base class for all target contexts
- numba/core/codegen.py - Driver for code generation
- numba/core/boxing.py - Boxing and unboxing for most data types
- numba/core/intrinsics.py - Utilities for converting LLVM intrinsics to other math calls
- numba/core/callconv.py - Implements different calling conventions for Numba-compiled functions
- numba/core/options.py - Container for options that control lowering
- numba/core/optional.py - Special type representing value or `None`
- numba/core/registry.py - Registry object for collecting implementations for a specific target
- numba/core/imputils.py - Helper functions for lowering
- numba/core/externals.py - Registers external C functions needed to link generated code
- numba/core/fastmathpass.py - Rewrite pass to add fastmath attributes to function call sites and binary operations
- numba/core/removerefctpass.py - Rewrite pass to remove unnecessary incref/decref pairs
- numba/core/descriptors.py - empty base class for all target descriptors (is this needed?)
- numba/cpython/builtins.py - Python builtin functions and operators
- numba/cpython/cmathimpl.py - Python complex math module
- numba/cpython/enumimpl.py - Enum objects

- numba/cpython/hashing.py - Hashing algorithms
- numba/cpython/heapq.py - Python `heapq` module
- numba/cpython/iterators.py - Iterable data types and iterators
- numba/cpython/listobj.py - Python lists
- numba/cpython/mathimpl.py - Python `math` module
- numba/cpython/numbers.py - Numeric values (int, float, etc)
- numba/cpython/printimpl.py - Print function
- numba/cpython/randomimpl.py - Python and NumPy `random` modules
- numba/cpython/rangeobj.py - Python *range* objects
- numba/cpython/slicing.py - Slice objects, and index calculations used in slicing
- numba/cpython/setobj.py - Python set type
- numba/cpython/tupleobj.py - Tuples (statically typed as immutable struct)
- numba/misc/cffiimpl.py - CFFI functions
- numba/misc/quicksort.py - Quicksort implementation used with list and array objects
- numba/misc/mergesort.py - Mergesort implementation used with array objects
- numba/np/arraymath.py - Math operations on arrays (both Python and NumPy)
- numba/np/arrayobj.py - Array operations (both NumPy and buffer protocol)
- numba/np/linalg.py - NumPy linear algebra operations
- numba/np/npdatetime.py - NumPy datetime operations
- numba/np/npyfuncs.py - Kernels used in generating some NumPy ufuncs
- numba/np/npyimpl.py - Implementations of most NumPy ufuncs
- numba/np/polynomial.py - `numpy.roots` function
- numba/np/ufunc_db.py - Big table mapping types to ufunc implementations

Ufunc Compiler and Runtime

- numba/np/ufunc - ufunc compiler implementation
- numba/np/ufunc/_internal.h | numba/np/ufunc/_internal.c - Python extension module with helper functions that use CPython & NumPy C API
- numba/np/ufunc/_ufunc.c - Used by *_internal.c*
- numba/np/ufunc/deviceufunc.py - Custom ufunc dispatch for non-CPU targets
- numba/np/ufunc/gufunc_scheduler.h | numba/np/ufunc/gufunc_scheduler.cpp - Schedule work chunks to threads
- numba/np/ufunc/dufunc.py - Special ufunc that can compile new implementations at call time
- numba/np/ufunc/ufuncbuilder.py - Top-level orchestration of ufunc/gufunc compiler pipeline
- numba/np/ufunc/sigparse.py - Parser for generalized ufunc indexing signatures
- numba/np/ufunc/parallel.py - Codegen for `parallel` target
- numba/np/ufunc/array_exprs.py - Rewrite pass for turning array expressions in regular functions into ufuncs

---

- numba/np/ufunc/wrappers.py - Wrap scalar function kernel with loops

- numba/np/ufunc/workqueue.h | numba/np/ufunc/workqueue.c - Threading backend based on pthreads/Windows threads and queues

- numba/np/ufunc/omppool.cpp - Threading backend based on OpenMP

- numba/np/ufunc/tbbpool.cpp - Threading backend based on TBB

Unit Tests (CPU)

CPU unit tests (GPU target unit tests listed in later sections

- runtests.py - Convenience script that launches test runner and turns on full compiler tracebacks

- run_coverage.py - Runs test suite with coverage tracking enabled

- .coveragerc - Coverage.py configuration

- numba/runtests.py - Entry point to unittest runner

- numba/testing/_runtests.py - Implementation of custom test runner command line interface

- numba/tests/test_* - Test cases

- numba/tests/*_usecases.py - Python functions compiled by some unit tests

- numba/tests/support.py - Helper functions for testing and special TestCase implementation

- numba/tests/dummy_module.py - Module used in `test_dispatcher.py`

- numba/tests/npyufunc - ufunc / gufunc compiler tests

- numba/testing - Support code for testing

- numba/testing/loader.py - Find tests on disk

- numba/testing/notebook.py - Support for testing notebooks

- numba/testing/main.py - Numba test runner

Command Line Utilities

- bin/numba - Command line stub, delegates to main in `numba_entry.py`

- numba/misc/numba_entry.py - Main function for `numba` command line tool

- numba/pycc - Ahead of time compilation of functions to shared library extension

- numba/pycc/__init__.py - Main function for `pycc` command line tool

- numba/pycc/cc.py - User-facing API for tagging functions to compile ahead of time

- numba/pycc/compiler.py - Compiler pipeline for creating standalone Python extension modules

- numba/pycc/llvm_types.py - Aliases to LLVM data types used by `compiler.py`

- numba/pycc/pycc - Stub to call main function. Is this still used?

- numba/pycc/modulemixin.c - C file compiled into every compiled extension. Pulls in C source from Numba core that is needed to make extension standalone

- numba/pycc/platform.py - Portable interface to platform-specific compiler toolchains

- numba/pycc/decorators.py - Deprecated decorators for tagging functions to compile. Use `cc.py` instead.

CUDA GPU Target

Note that the CUDA target does reuse some parts of the CPU target.

- numba/cuda/ - The implementation of the CUDA (NVIDIA GPU) target and associated unit tests
- numba/cuda/decorators.py - Compiler decorators for CUDA kernels and device functions
- numba/cuda/dispatcher.py - Dispatcher for CUDA JIT functions
- numba/cuda/printimpl.py - Special implementation of device printing
- numba/cuda/libdevice.py - Registers libdevice functions
- numba/cuda/kernels/ - Custom kernels for reduction and transpose
- numba/cuda/device_init.py - Initializes the CUDA target when imported
- numba/cuda/compiler.py - Compiler pipeline for CUDA target
- numba/cuda/intrinsic_wrapper.py - CUDA device intrinsics (shuffle, ballot, etc)
- numba/cuda/initialize.py - Defered initialization of the CUDA device and subsystem. Called only when user imports `numba.cuda`
- numba/cuda/simulator_init.py - Initalizes the CUDA simulator subsystem (only when user requests it with env var)
- numba/cuda/random.py - Implementation of random number generator
- numba/cuda/api.py - User facing APIs imported into `numba.cuda.*`
- numba/cuda/stubs.py - Python placeholders for functions that only can be used in GPU device code
- numba/cuda/simulator/ - Simulate execution of CUDA kernels in Python interpreter
- numba/cuda/vectorizers.py - Subclasses of ufunc/gufunc compilers for CUDA
- numba/cuda/args.py - Management of kernel arguments, including host<->device transfers
- numba/cuda/target.py - Typing and target contexts for GPU
- numba/cuda/cudamath.py - Type signatures for math functions in CUDA Python
- numba/cuda/errors.py - Validation of kernel launch configuration
- numba/cuda/nvvmutils.py - Helper functions for generating NVVM-specific IR
- numba/cuda/testing.py - Support code for creating CUDA unit tests and capturing standard out
- numba/cuda/cudadecl.py - Type signatures of CUDA API (threadIdx, blockIdx, atomics) in Python on GPU
- numba/cuda/cudaimpl.py - Implementations of CUDA API functions on GPU
- numba/cuda/codegen.py - Code generator object for CUDA target
- numba/cuda/cudadrv/ - Wrapper around CUDA driver API
- numba/cuda/tests/ - CUDA unit tests, skipped when CUDA is not detected
- numba/cuda/tests/cudasim/ - Tests of CUDA simulator
- numba/cuda/tests/nocuda/ - Tests for NVVM functionality when CUDA not present
- numba/cuda/tests/cudapy/ - Tests of compiling Python functions for GPU
- numba/cuda/tests/cudadrv/ - Tests of Python wrapper around CUDA API

ROCm GPU Target

Note that the ROCm target does reuse some parts of the CPU target, and duplicates some code from CUDA target. A future refactoring could pull out the common subset of CUDA and ROCm. An older version of this target was based on the HSA API, so "hsa" appears in many places.

- numba/roc - ROCm GPU target for AMD GPUs
- numba/roc/descriptor.py - TargetDescriptor subclass for ROCm target
- numba/roc/enums.py - Internal constants
- numba/roc/mathdecl.py - Declarations of math functions that can be used on device
- numba/roc/mathimpl.py - Implementations of math functions for device
- numba/roc/compiler.py - Compiler pipeline for ROCm target
- numba/roc/hlc - Wrapper around LLVM interface for AMD GPU
- numba/roc/initialize.py - Register ROCm target for ufunc/gufunc compiler
- numba/roc/hsadecl.py - Type signatures for ROCm device API in Python
- numba/roc/hsaimpl.py - Implementations of ROCm device API
- numba/roc/dispatch.py - ufunc/gufunc dispatcher
- numba/roc/README.md - Notes on testing target (should be deleted)
- numba/roc/api.py - Host API for ROCm actions
- numba/roc/gcn_occupancy.py - Heuristic to compute occupancy of kernels
- numba/roc/stubs.py - Host stubs for device functions
- numba/roc/vectorizers.py - Builds ufuncs
- numba/roc/target.py - Target and typing contexts
- numba/roc/hsadrv - Python wrapper around ROCm (based on HSA) driver API calls
- numba/roc/codegen.py - Codegen subclass for ROCm target
- numba/roc/decorators.py - `@jit` decorator for kernels and device functions
- numba/roc/servicelib/threadlocal.py - Thread-local stack used by ROC targets
- numba/roc/servicelib/service.py - Should be removed?
- numba/roc/tests - Unit tests for ROCm target
- numba/roc/tests/hsapy - Tests of compiling ROCm kernels written in Python syntax
- numba/roc/tests/hsadrv - Tests of Python wrapper on platform API.

7.3 Numba architecture

7.3.1 Introduction

Numba is a compiler for Python bytecode with optional type-specialization.

Suppose you enter a function like this into the standard Python interpreter (henceforward referred to as "CPython"):

```
def add(a, b):
    return a + b
```

The interpreter will immediately parse the function and convert it into a bytecode representation that describes how the CPython interpreter should execute the function at a low level. For the example above, it looks something like this:

```
>>> import dis
>>> dis.dis(add)
2           0 LOAD_FAST                0 (a)
            3 LOAD_FAST                1 (b)
            6 BINARY_ADD
            7 RETURN_VALUE
```

CPython uses a stack-based interpreter (much like an HP calculator), so the code first pushes two local variables onto the stack. The BINARY_ADD opcode pops the top two arguments off the stack and makes a Python C API function call that is equivalent to calling a.__add__(b). The result is then pushed onto the top of the interpreter stack. Finally, the RETURN_VALUE opcode returns value on the top of the stack as the result of the function call.

Numba can take this bytecode and compile it to machine code that performs the same operations as the CPython interpreter, treating a and b as generic Python objects. The full semantics of Python are preserved, and the compiled function can be used with any kind of objects that have the add operator defined. When a Numba function is compiled this way, we say that it has been compiled in *object mode*, because the code still manipulates Python objects.

Numba code compiled in object mode is not much faster than executing the original Python function in the CPython interpreter. However, if we specialize the function to only run with certain data types, Numba can generate much shorter and more efficient code that manipulates the data natively without any calls into the Python C API. When code has been compiled for specific data types so that the function body no longer relies on the Python runtime, we say the function has been compiled in *nopython mode*. Numeric code compiled in nopython mode can be hundreds of times faster than the original Python.

7.3.2 Compiler architecture

Like many compilers, Numba can be conceptually divided into a *frontend* and a *backend*.

The Numba *frontend* comprises the stages which analyze the Python bytecode, translate it to *Numba IR* and perform various transformations and analysis steps on the IR. One of the key steps is *type inference*. The frontend must succeed in typing all variables unambiguously in order for the backend to generate code in *nopython mode*, because the backend uses type information to match appropriate code generators with the values they operate on.

The Numba *backend* walks the Numba IR resulting from the frontend analyses and exploits the type information deduced by the type inference phase to produce the right LLVM code for each encountered operation. After LLVM code is produced, the LLVM library is asked to optimize it and generate native processor code for the final, native function.

There are other pieces besides the compiler frontend and backend, such as the caching machinery for JIT functions. Those pieces are not considered in this document.

7.3.3 Contexts

Numba is quite flexible, allowing it to generate code for different hardware architectures like CPUs and GPUs. In order to support these different applications, Numba uses a *typing context* and a *target context*.

A *typing context* is used in the compiler frontend to perform type inference on operations and values in the function. Similar typing contexts could be used for many architectures because for nearly all cases, typing inference is hardware-independent. However, Numba currently has a different typing context for each target.

A *target context* is used to generate the specific instruction sequence required to operate on the Numba types identified during type inference. Target contexts are architecture-specific and are flexible in defining the execution model and available Python APIs. For example, Numba has a "cpu" and a "cuda" context for those two kinds of architecture, and a "parallel" context which produces multithreaded CPU code.

7.3.4 Compiler stages

The *jit()* decorator in Numba ultimately calls `numba.compiler.compile_extra()` which compiles the Python function in a multi-stage process, described below.

Stage 1: Analyze bytecode

At the start of compilation, the function bytecode is passed to an instance of the Numba interpreter (`numba.interpreter`). The interpreter object analyzes the bytecode to find the control flow graph (`numba.controlflow`). The control flow graph (CFG) describes the ways that execution can move from one block to the next inside the function as a result of loops and branches.

The data flow analysis (`numba.dataflow`) takes the control flow graph and traces how values get pushed and popped off the Python interpreter stack for different code paths. This is important to understand the lifetimes of variables on the stack, which are needed in Stage 2.

If you set the environment variable `NUMBA_DUMP_CFG` to 1, Numba will dump the results of the control flow graph analysis to the screen. Our `add()` example is pretty boring, since there is only one statement block:

```
CFG adjacency lists:
{0: []}
CFG dominators:
{0: set([0])}
CFG post-dominators:
{0: set([0])}
CFG back edges: []
CFG loops:
{}
CFG node-to-loops:
{0: []}
```

A function with more complex flow control will have a more interesting control flow graph. This function:

```python
def doloops(n):
    acc = 0
    for i in range(n):
        acc += 1
        if n == 10:
            break
    return acc
```

compiles to this bytecode:

```
 9           0 LOAD_CONST               1 (0)
             3 STORE_FAST               1 (acc)

10           6 SETUP_LOOP              46 (to 55)
             9 LOAD_GLOBAL              0 (range)
            12 LOAD_FAST                0 (n)
            15 CALL_FUNCTION            1
```

```
             18 GET_ITER
       >>    19 FOR_ITER                 32 (to 54)
             22 STORE_FAST                2 (i)

11           25 LOAD_FAST                 1 (acc)
             28 LOAD_CONST                2 (1)
             31 INPLACE_ADD
             32 STORE_FAST                1 (acc)

12           35 LOAD_FAST                 0 (n)
             38 LOAD_CONST                3 (10)
             41 COMPARE_OP                2 (==)
             44 POP_JUMP_IF_FALSE        19

13           47 BREAK_LOOP
             48 JUMP_ABSOLUTE            19
             51 JUMP_ABSOLUTE            19
       >>    54 POP_BLOCK

14     >>    55 LOAD_FAST                 1 (acc)
             58 RETURN_VALUE
```

The corresponding CFG for this bytecode is:

```
CFG adjacency lists:
{0: [6], 6: [19], 19: [54, 22], 22: [19, 47], 47: [55], 54: [55], 55: []}
CFG dominators:
{0: set([0]),
 6: set([0, 6]),
 19: set([0, 6, 19]),
 22: set([0, 6, 19, 22]),
 47: set([0, 6, 19, 22, 47]),
 54: set([0, 6, 19, 54]),
 55: set([0, 6, 19, 55])}
CFG post-dominators:
{0: set([0, 6, 19, 55]),
 6: set([6, 19, 55]),
 19: set([19, 55]),
 22: set([22, 55]),
 47: set([47, 55]),
 54: set([54, 55]),
 55: set([55])}
CFG back edges: [(22, 19)]
CFG loops:
{19: Loop(entries=set([6]), exits=set([54, 47]), header=19, body=set([19, 22]))}
CFG node-to-loops:
{0: [], 6: [], 19: [19], 22: [19], 47: [], 54: [], 55: []}
```

The numbers in the CFG refer to the bytecode offsets shown just to the left of the opcode names above.

Stage 2: Generate the Numba IR

Once the control flow and data analyses are complete, the Numba interpreter can step through the bytecode and translate it into an Numba-internal intermediate representation. This translation process changes the function from a stack machine representation (used by the Python interpreter) to a register machine representation (used by LLVM).

Although the IR is stored in memory as a tree of objects, it can be serialized to a string for debugging. If you set the environment variable NUMBA_DUMP_IR equal to 1, the Numba IR will be dumped to the screen. For the add() function described above, the Numba IR looks like:

```
label 0:
    a = arg(0, name=a)                          ['a']
    b = arg(1, name=b)                          ['b']
    $0.3 = a + b                                ['$0.3', 'a', 'b']
    del b                                       []
    del a                                       []
    $0.4 = cast(value=$0.3)                     ['$0.3', '$0.4']
    del $0.3                                    []
    return $0.4                                 ['$0.4']
```

The `del` instructions are produced by *Live Variable Analysis*. Those instructions ensure references are not leaked. In *nopython mode*, some objects are tracked by the Numba runtime and some are not. For tracked objects, a dereference operation is emitted; otherwise, the instruction is an no-op. In *object mode* each variable contains an owned reference to a PyObject.

Stage 3: Macro expansion

Now that the function has been translated into the Numba IR, macro expansion can be performed. Macro expansion converts specific attributes that are known to Numba into IR nodes representing function calls. This is initiated in the `numba.compiler.translate_stage` function, and is implemented in `numba.macro`.

Examples of attributes that are macro-expanded include the CUDA intrinsics for grid, block and thread dimensions and indices. For example, the assignment to `tx` in the following function:

```
@cuda.jit(argtypes=[f4[:]])
def f(a):
    tx = cuda.threadIdx.x
```

has the following representation after translation to Numba IR:

```
$0.1 = global(cuda: <module 'numba.cuda' from '...'>) ['$0.1']
$0.2 = getattr(value=$0.1, attr=threadIdx) ['$0.1', '$0.2']
del $0.1                                   []
$0.3 = getattr(value=$0.2, attr=x)        ['$0.2', '$0.3']
del $0.2                                   []
tx = $0.3                                  ['$0.3', 'tx']
```

After macro expansion, the `$0.3 = getattr(value=$0.2, attr=x)` IR node is translated into:

```
$0.3 = call tid.x(, )                      ['$0.3']
```

which represents an instance of the `Intrinsic` IR node for calling the `tid.x` intrinsic function.

Stage 4: Rewrite untyped IR

Before running type inference, it may be desired to run certain transformations on the Numba IR. One such example is to detect `raise` statements which have an implicitly constant argument, so as to support them in *nopython mode*. Let's say you compile the following function with Numba:

```
def f(x):
    if x == 0:
        raise ValueError("x cannot be zero")
```

If you set the *NUMBA_DUMP_IR* environment variable to `1`, you'll see the IR being rewritten before the type inference phase:

```
REWRITING:
    del $0.3                                []
    $12.1 = global(ValueError: <class 'ValueError'>) ['$12.1']
    $const12.2 = const(str, x cannot be zero) ['$const12.2']
    $12.3 = call $12.1($const12.2)          ['$12.1', '$12.3', '$const12.2']
    del $const12.2                          []
    del $12.1                               []
    raise $12.3                             ['$12.3']
    ─────────────────────────────────────────────────────
    del $0.3                                []
    $12.1 = global(ValueError: <class 'ValueError'>) ['$12.1']
    $const12.2 = const(str, x cannot be zero) ['$const12.2']
    $12.3 = call $12.1($const12.2)          ['$12.1', '$12.3', '$const12.2']
    del $const12.2                          []
    del $12.1                               []
    raise <class 'ValueError'>('x cannot be zero') []
```

Stage 5: Infer types

Now that the Numba IR has been generated and macro-expanded, type analysis can be performed. The types of the function arguments can be taken either from the explicit function signature given in the `@jit` decorator (such as `@jit('float64(float64, float64)')`), or they can be taken from the types of the actual function arguments if compilation is happening when the function is first called.

The type inference engine is found in `numba.typeinfer`. Its job is to assign a type to every intermediate variable in the Numba IR. The result of this pass can be seen by setting the *NUMBA_DUMP_ANNOTATION* environment variable to 1:

```
-------------------------------ANNOTATION--------------------------------
# File: archex.py
# --- LINE 4 ---

@jit(nopython=True)

# --- LINE 5 ---

def add(a, b):

    # --- LINE 6 ---
    # label 0
    #   a = arg(0, name=a)  :: int64
    #   b = arg(1, name=b)  :: int64
    #   $0.3 = a + b  :: int64
    #   del b
    #   del a
    #   $0.4 = cast(value=$0.3)  :: int64
    #   del $0.3
    #   return $0.4

    return a + b
```

If type inference fails to find a consistent type assignment for all the intermediate variables, it will label every variable as type `pyobject` and fall back to object mode. Type inference can fail when unsupported Python types, language features, or functions are used in the function body.

Stage 6a: Rewrite typed IR

This pass's purpose is to perform any high-level optimizations that still require, or could at least benefit from, Numba IR type information.

One example of a problem domain that isn't as easily optimized once lowered is the domain of multidimensional array operations. When Numba lowers an array operation, Numba treats the operation like a full ufunc kernel. During lowering a single array operation, Numba generates an inline broadcasting loop that creates a new result array. Then Numba generates an application loop that applies the operator over the array inputs. Recognizing and rewriting these loops once they are lowered into LLVM is hard, if not impossible.

An example pair of optimizations in the domain of array operators is loop fusion and shortcut deforestation. When the optimizer recognizes that the output of one array operator is being fed into another array operator, and only to that array operator, it can fuse the two loops into a single loop. The optimizer can further eliminate the temporary array allocated for the initial operation by directly feeding the result of the first operation into the second, skipping the store and load to the intermediate array. This elimination is known as shortcut deforestation. Numba currently uses the rewrite pass to implement these array optimizations. For more information, please consult the "*Case study: Array Expressions*" subsection, later in this document.

One can see the result of rewriting by setting the `NUMBA_DUMP_IR` environment variable to a non-zero value (such as 1). The following example shows the output of the rewrite pass as it recognizes an array expression consisting of a multiply and add, and outputs a fused kernel as a special operator, `arrayexpr()`:

```
REWRITING:
a0 = arg(0, name=a0)                        ['a0']
a1 = arg(1, name=a1)                        ['a1']
a2 = arg(2, name=a2)                        ['a2']
$0.3 = a0 * a1                              ['$0.3', 'a0', 'a1']
del a1                                      []
del a0                                      []
$0.5 = $0.3 + a2                            ['$0.3', '$0.5', 'a2']
del a2                                      []
del $0.3                                    []
$0.6 = cast(value=$0.5)                     ['$0.5', '$0.6']
del $0.5                                    []
return $0.6                                 ['$0.6']

a0 = arg(0, name=a0)                        ['a0']
a1 = arg(1, name=a1)                        ['a1']
a2 = arg(2, name=a2)                        ['a2']
$0.5 = arrayexpr(ty=array(float64, 1d, C), expr=('+', [('*', [Var(a0, test.py (14)),
→Var(a1, test.py (14))]), Var(a2, test.py (14))])) ['$0.5', 'a0', 'a1', 'a2']
del a0                                      []
del a1                                      []
del a2                                      []
$0.6 = cast(value=$0.5)                     ['$0.5', '$0.6']
del $0.5                                    []
return $0.6                                 ['$0.6']
```

Following this rewrite, Numba lowers the array expression into a new ufunc-like function that is inlined into a single loop that only allocates a single result array.

Stage 6b: Perform Automatic Parallelization

This pass is only performed if the `parallel` option in the `jit()` decorator is set to `True`. This pass finds parallelism implicit in the semantics of operations in the Numba IR and replaces those operations with explicitly parallel representations of those operations using a special *parfor* operator. Then, optimizations are performed to maximize the number of parfors that are adjacent to each other such that they can then be fused together into one parfor that takes only one pass over the data and will thus typically have better cache performance. Finally, during lowering, these parfor operators are converted to a form similar to guvectorize to implement the actual parallelism.

The automatic parallelization pass has a number of sub-passes, many of which are controllable using a dictionary of options passed via the `parallel` keyword argument to `jit()`:

```
{ 'comprehension': True/False,   # parallel comprehension
  'prange':        True/False,   # parallel for-loop
  'numpy':         True/False,   # parallel numpy calls
  'reduction':     True/False,   # parallel reduce calls
  'setitem':       True/False,   # parallel setitem
  'stencil':       True/False,   # parallel stencils
  'fusion':        True/False,   # enable fusion or not
}
```

The default is set to *True* for all of them. The sub-passes are described in more detail in the following paragraphs.

1. **CFG Simplification** Sometimes Numba IR will contain chains of blocks containing no loops which are merged in this sub-pass into single blocks. This sub-pass simplifies subsequent analysis of the IR.

2. **Numpy canonicalization** Some Numpy operations can be written as operations on Numpy objects (e.g. `arr.sum()`), or as calls to Numpy taking those objects (e.g. `numpy.sum(arr)`). This sub-pass converts all such operations to the latter form for cleaner subsequent analysis.

3. **Array analysis** A critical requirement for later parfor fusion is that parfors have identical iteration spaces and these iteration spaces typically correspond to the sizes of the dimensions of Numpy arrays. In this sub-pass, the IR is analyzed to determine equivalence classes for the dimensions of Numpy arrays. Consider the example, `a = b + 1`, where `a` and `b` are both Numpy arrays. Here, we know that each dimension of `a` must have the same equivalence class as the corresponding dimension of `b`. Typically, routines rich in Numpy operations will enable equivalence classes to be fully known for all arrays created within a function.

   Array analysis will also reason about size equivalence for slice selection, and boolean array masking (one dimensional only). For example, it is able to infer that `a[1 : n-1]` is of the same size as `b[0 : n-2]`.

   Array analysis may also insert safety assumptions to ensure pre-conditions related to array sizes are met before an operation can be parallelized. For example, `np.dot(X, w)` between a 2-D matrix `X` and a 1-D vector `w` requires that the second dimension of `X` is of the same size as `w`. Usually this kind of runtime check is automatically inserted, but if array analysis can infer such equivalence, it will skip them.

   Users can even help array analysis by turning implicit knowledge about array sizes into explicit assertions. For example, in the code below:

   ```
   @numba.njit(parallel=True)
   def logistic_regression(Y, X, w, iterations):
       assert(X.shape == (Y.shape[0], w.shape[0]))
       for i in range(iterations):
           w -= np.dot(((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y), X)
       return w
   ```

   Making the explicit assertion helps eliminate all bounds checks in the rest of the function.

4. **`prange()` to parfor** The use of prange (*Explicit Parallel Loops*) in a for loop is an explicit indication from the programmer that all iterations of the for loop can execute in parallel. In this sub-pass, we analyze the CFG to locate loops and to convert those loops controlled by a prange object to the explicit *parfor* operator. Each explicit parfor operator consists of:

   a. A list of loop nest information that describes the iteration space of the parfor. Each entry in the loop nest list contains an indexing variable, the start of the range, the end of the range, and the step value for each iteration.

   b. An initialization (init) block which contains instructions to be executed one time before the parfor begins executing.

   c. A loop body comprising a set of basic blocks that correspond to the body of the loop and compute one point in the iteration space.

   d. The index variables used for each dimension of the iteration space.

   For parfor *pranges*, the loop nest is a single entry where the start, stop, and step fields come from the specified *prange*. The init block is empty for *prange* parfors and the loop body is the set of blocks in the loop minus the loop header.

   With parallelization on, array comprehensions (*List comprehension*) will also be translated to prange so as to run in parallel. This behavior be disabled by setting `parallel={'comprehension':  False}`.

   Likewise, the overall *prange* to *parfor* translation can be disabled by setting `parallel={'prange': False}`, in which case *prange* is treated the same as *range*.

5. **Numpy to parfor** In this sub-pass, Numpy functions such as `ones`, `zeros`, `dot`, most of the random number generating functions, arrayexprs (from Section *Stage 6a: Rewrite typed IR*), and Numpy reductions are converted to parfors. Generally, this conversion creates the loop nest list, whose length is equal to the number of dimensions of the left-hand side of the assignment instruction in the IR. The number and size of the dimensions of the left-hand-side array is taken from the array analysis information generated in sub-pass 3 above. An instruction to create the result Numpy array is generated and stored in the new parfor's init block. A basic block is created for the loop body and an instruction is generated and added to the end of that block to store the result of the computation into the array at the current point in the iteration space. The result stored into the array depends on the operation that is being converted. For example, for `ones`, the value stored is a constant 1. For calls to generate a random array, the value comes from a call to the same random number function but with the size parameter dropped and therefore returning a scalar. For arrayexpr operators, the arrayexpr tree is converted to Numba IR and the value at the root of that expression tree is used to write into the output array. The translation from Numpy functions and arrayexpr operators to *parfor* can be disabled by setting `parallel={'numpy':  False}`.

   For reductions, the loop nest list is similarly created using the array analysis information for the array being reduced. In the init block, the initial value is assigned to the reduction variable. The loop body consists of a single block in which the next value in the iteration space is fetched and the reduction operation is applied to that value and the current reduction value and the result stored back into the reduction value. The translation of reduction functions to *parfor* can be disabled by setting `parallel={'reduction': False}`.

   Setting the *NUMBA_DEBUG_ARRAY_OPT_STATS* environment variable to 1 will show some statistics about parfor conversions in general.

6. **Setitem to parfor** Setting a range of array elements using a slice or boolean array selection can also run in parallel. Statement such as `A[P]  =  B[Q]` (or a simpler case `A[P]  =  c`, where `c` is a scalar) is translated to *parfor* if one of the following conditions is met:

   a. `P` and `Q` are slices or multi-dimensional selector involving scalar and slices, and `A[P]` and `B[Q]` are considered size equivalent by array analysis. Only 2-value slice/range is supported, 3-value with a step will not be translated to *parfor*.

b. `P` and `Q` are the same boolean array.

This translation can be disabled by setting `parallel={'setitem':  False}`.

7. **Simplification** Performs a copy propagation and dead code elimination pass.

8. **Fusion** This sub-pass first processes each basic block and does a reordering of the instructions within the block with the goal of pushing parfors lower in the block and lifting non-parfors towards the start of the block. In practice, this approach does a good job of getting parfors adjacent to each other in the IR, which enables more parfors to then be fused. During parfor fusion, each basic block is repeatedly scanned until no further fusion is possible. During this scan, each set of adjacent instructions are considered. Adjacent instructions are fused together if:

   a. they are both parfors

   b. the parfors' loop nests are the same size and the array equivalence classes for each dimension of the loop nests are the same, and

   c. the first parfor does not create a reduction variable used by the second parfor.

   The two parfors are fused together by adding the second parfor's init block to the first's, merging the two parfors' loop bodies together and replacing the instances of the second parfor's loop index variables in the second parfor's body with the loop index variables for the first parfor. Fusion can be disabled by setting `parallel={'fusion':  False}`.

   Setting the *NUMBA_DEBUG_ARRAY_OPT_STATS* environment variable to 1 will show some statistics about parfor fusions.

9. **Push call objects and compute parfor parameters** In the lowering phase described in Section *Stage 7a: Generate nopython LLVM IR*, each parfor becomes a separate function executed in parallel in `guvectorize` (*The @guvectorize decorator*) style. Since parfors may use variables defined previously in a function, when those parfors become separate functions, those variables must be passed to the parfor function as parameters. In this sub-pass, a use-def scan is made over each parfor body and liveness information is used to determine which variables are used but not defined by the parfor. That list of variables is stored here in the parfor for use during lowering. Function variables are a special case in this process since function variables cannot be passed to functions compiled in nopython mode. Instead, for function variables, this sub-pass pushes the assignment instruction to the function variable into the parfor body so that those do not need to be passed as parameters.

   To see the intermediate IR between the above sub-passes and other debugging information, set the *NUMBA_DEBUG_ARRAY_OPT* environment variable to 1. For the example in Section *Stage 6a: Rewrite typed IR*, the following IR with a parfor is generated during this stage:

```
label 0:
    a0 = arg(0, name=a0)                        ['a0']
    a0_sh_attr0.0 = getattr(attr=shape, value=a0) ['a0', 'a0_sh_attr0.0']
    $consta00.1 = const(int, 0)                 ['$consta00.1']
    a0size0.2 = static_getitem(value=a0_sh_attr0.0, index_var=$consta00.1,␣
↪index=0) ['$consta00.1', 'a0_sh_attr0.0', 'a0size0.2']
    a1 = arg(1, name=a1)                        ['a1']
    a1_sh_attr0.3 = getattr(attr=shape, value=a1) ['a1', 'a1_sh_attr0.3']
    $consta10.4 = const(int, 0)                 ['$consta10.4']
    a1size0.5 = static_getitem(value=a1_sh_attr0.3, index_var=$consta10.4,␣
↪index=0) ['$consta10.4', 'a1_sh_attr0.3', 'a1size0.5']
    a2 = arg(2, name=a2)                        ['a2']
    a2_sh_attr0.6 = getattr(attr=shape, value=a2) ['a2', 'a2_sh_attr0.6']
    $consta20.7 = const(int, 0)                 ['$consta20.7']
    a2size0.8 = static_getitem(value=a2_sh_attr0.6, index_var=$consta20.7,␣
↪index=0) ['$consta20.7', 'a2_sh_attr0.6', 'a2size0.8']
```

(continues on next page)

```
---begin parfor 0---
index_var =  parfor_index.9
LoopNest(index_variable=parfor_index.9, range=0,a0size0.2,1 correlation=5)
init block:
    $np_g_var.10 = global(np: <module 'numpy' from '/usr/local/lib/python3.5/
→dist-packages/numpy/__init__.py'>) ['$np_g_var.10']
    $empty_attr_attr.11 = getattr(attr=empty, value=$np_g_var.10) ['$empty_
→attr_attr.11', '$np_g_var.10']
    $np_typ_var.12 = getattr(attr=float64, value=$np_g_var.10) ['$np_g_var.10
→', '$np_typ_var.12']
    $0.5 = call $empty_attr_attr.11(a0size0.2, $np_typ_var.12, kws=(), func=
→$empty_attr_attr.11, vararg=None, args=[Var(a0size0.2, test2.py (7)), Var(
→$np_typ_var.12, test2.py (7))]) ['$0.5', '$empty_attr_attr.11', '$np_typ_
→var.12', 'a0size0.2']
label 1:
    $arg_out_var.15 = getitem(value=a0, index=parfor_index.9) ['$arg_out_var.
→15', 'a0', 'parfor_index.9']
    $arg_out_var.16 = getitem(value=a1, index=parfor_index.9) ['$arg_out_var.
→16', 'a1', 'parfor_index.9']
    $arg_out_var.14 = $arg_out_var.15 * $arg_out_var.16 ['$arg_out_var.14', '
→$arg_out_var.15', '$arg_out_var.16']
    $arg_out_var.17 = getitem(value=a2, index=parfor_index.9) ['$arg_out_var.
→17', 'a2', 'parfor_index.9']
    $expr_out_var.13 = $arg_out_var.14 + $arg_out_var.17 ['$arg_out_var.14', '
→$arg_out_var.17', '$expr_out_var.13']
    $0.5[parfor_index.9] = $expr_out_var.13  ['$0.5', '$expr_out_var.13',
→'parfor_index.9']
----end parfor 0----
    $0.6 = cast(value=$0.5)                    ['$0.5', '$0.6']
    return $0.6                                ['$0.6']
```

Stage 7a: Generate nopython LLVM IR

If type inference succeeds in finding a Numba type for every intermediate variable, then Numba can (potentially) generate specialized native code. This process is called *lowering*. The Numba IR tree is translated into LLVM IR by using helper classes from llvmlite. The machine-generated LLVM IR can seem unnecessarily verbose, but the LLVM toolchain is able to optimize it quite easily into compact, efficient code.

The basic lowering algorithm is generic, but the specifics of how particular Numba IR nodes are translated to LLVM instructions is handled by the target context selected for compilation. The default target context is the "cpu" context, defined in `numba.targets.cpu`.

The LLVM IR can be displayed by setting the *NUMBA_DUMP_LLVM* environment variable to 1. For the "cpu" context, our `add()` example would look like:

```
define i32 @"__main__.add$1.int64.int64"(i64* %"retptr",
                                          {i8*, i32}** %"excinfo",
                                          i8* %"env",
                                          i64 %"arg.a", i64 %"arg.b")
{
   entry:
     %"a" = alloca i64
     %"b" = alloca i64
```

```
    %"$0.3" = alloca i64
    %"$0.4" = alloca i64
    br label %"B0"
  B0:
    store i64 %"arg.a", i64* %"a"
    store i64 %"arg.b", i64* %"b"
    %".8" = load i64* %"a"
    %".9" = load i64* %"b"
    %".10" = add i64 %".8", %".9"
    store i64 %".10", i64* %"$0.3"
    %".12" = load i64* %"$0.3"
    store i64 %".12", i64* %"$0.4"
    %".14" = load i64* %"$0.4"
    store i64 %".14", i64* %"retptr"
    ret i32 0
}
```

The post-optimization LLVM IR can be output by setting `NUMBA_DUMP_OPTIMIZED` to 1. The optimizer shortens the code generated above quite significantly:

```
define i32 @"__main__.add$1.int64.int64"(i64* nocapture %retptr,
                                          { i8*, i32 }** nocapture readnone %excinfo,
                                          i8* nocapture readnone %env,
                                          i64 %arg.a, i64 %arg.b)
{
  entry:
    %.10 = add i64 %arg.b, %arg.a
    store i64 %.10, i64* %retptr, align 8
    ret i32 0
}
```

If created during *Stage 6b: Perform Automatic Parallelization*, parfor operations are lowered in the following manner. First, instructions in the parfor's init block are lowered into the existing function using the normal lowering code. Second, the loop body of the parfor is turned into a separate GUFunc. Third, code is emitted for the current function to call the parallel GUFunc.

To create a GUFunc from the parfor body, the signature of the GUFunc is created by taking the parfor parameters as identified in step 9 of Stage *Stage 6b: Perform Automatic Parallelization* and adding to that a special *schedule* parameter, across which the GUFunc will be parallelized. The schedule parameter is in effect a static schedule mapping portions of the parfor iteration space to Numba threads and so the length of the schedule array is the same as the number of configured Numba threads. To make this process easier and somewhat less dependent on changes to Numba IR, this stage creates a Python function as text that contains the parameters to the GUFunc and iteration code that takes the current schedule entry and loops through the specified portion of the iteration space. In the body of that loop, a special sentinel is inserted for subsequent easy location. This code that handles the processing of the iteration space is then `eval`'ed into existence and the Numba compiler's run_frontend function is called to generate IR. That IR is scanned to locate the sentinel and the sentinel is replaced with the loop body of the parfor. Then, the process of creating the parallel GUFunc is completed by compiling this merged IR with the Numba compiler's `compile_ir` function.

To call the parallel GUFunc, the static schedule must be created. Code is inserted to call a function named `do_scheduling`. This function is called with the size of each of the parfor's dimensions and the number *N* of configured Numba threads (`NUMBA_NUM_THREADS`). The `do_scheduling` function will divide the iteration space into N approximately equal sized regions (linear for 1D, rectangular for 2D, or hyperrectangles for 3+D) and the resulting schedule is passed to the parallel GUFunc. The number of threads dedicated to a given dimension of the full iteration space is roughly proportional to the ratio of the size of the given dimension to the sum of the sizes of all the dimensions of the iteration space.

Parallel reductions are not natively provided by GUFuncs but the parfor lowering strategy allows us to use GUFuncs

in a way that reductions can be performed in parallel. To accomplish this, for each reduction variable computed by a parfor, the parallel GUFunc and the code that calls it are modified to make the scalar reduction variable into an array of reduction variables whose length is equal to the number of Numba threads. In addition, the GUFunc still contains a scalar version of the reduction variable that is updated by the parfor body during each iteration. One time at the end of the GUFunc this local reduction variable is copied into the reduction array. In this way, false sharing of the reduction array is prevented. Code is also inserted into the main function after the parallel GUFunc has returned that does a reduction across this smaller reduction array and this final reduction value is then stored into the original scalar reduction variable.

The GUFunc corresponding to the example from Section *Stage 6b: Perform Automatic Parallelization* can be seen below:

```
label 0:
    sched.29 = arg(0, name=sched)           ['sched.29']
    a0 = arg(1, name=a0)                     ['a0']
    a1 = arg(2, name=a1)                     ['a1']
    a2 = arg(3, name=a2)                     ['a2']
    _0_5 = arg(4, name=_0_5)                 ['_0_5']
    $3.1.24 = global(range: <class 'range'>) ['$3.1.24']
    $const3.3.21 = const(int, 0)             ['$const3.3.21']
    $3.4.23 = getitem(value=sched.29, index=$const3.3.21) ['$3.4.23', '$const3.3.21',
→'sched.29']
    $const3.6.28 = const(int, 1)             ['$const3.6.28']
    $3.7.27 = getitem(value=sched.29, index=$const3.6.28) ['$3.7.27', '$const3.6.28',
→'sched.29']
    $const3.8.32 = const(int, 1)             ['$const3.8.32']
    $3.9.31 = $3.7.27 + $const3.8.32         ['$3.7.27', '$3.9.31', '$const3.8.32']
    $3.10.36 = call $3.1.24($3.4.23, $3.9.31, kws=[], func=$3.1.24, vararg=None,␣
→args=[Var($3.4.23, <string> (2)), Var($3.9.31, <string> (2))]) ['$3.1.24', '$3.10.36
→', '$3.4.23', '$3.9.31']
    $3.11.30 = getiter(value=$3.10.36)       ['$3.10.36', '$3.11.30']
    jump 1                                   []
label 1:
    $28.2.35 = iternext(value=$3.11.30)      ['$28.2.35', '$3.11.30']
    $28.3.25 = pair_first(value=$28.2.35)    ['$28.2.35', '$28.3.25']
    $28.4.40 = pair_second(value=$28.2.35)   ['$28.2.35', '$28.4.40']
    branch $28.4.40, 2, 3                    ['$28.4.40']
label 2:
    $arg_out_var.15 = getitem(value=a0, index=$28.3.25) ['$28.3.25', '$arg_out_var.15
→', 'a0']
    $arg_out_var.16 = getitem(value=a1, index=$28.3.25) ['$28.3.25', '$arg_out_var.16
→', 'a1']
    $arg_out_var.14 = $arg_out_var.15 * $arg_out_var.16 ['$arg_out_var.14', '$arg_out_
→var.15', '$arg_out_var.16']
    $arg_out_var.17 = getitem(value=a2, index=$28.3.25) ['$28.3.25', '$arg_out_var.17
→', 'a2']
    $expr_out_var.13 = $arg_out_var.14 + $arg_out_var.17 ['$arg_out_var.14', '$arg_
→out_var.17', '$expr_out_var.13']
    _0_5[$28.3.25] = $expr_out_var.13        ['$28.3.25', '$expr_out_var.13', '_0_5']
    jump 1                                   []
label 3:
    $const44.1.33 = const(NoneType, None)    ['$const44.1.33']
    $44.2.39 = cast(value=$const44.1.33)     ['$44.2.39', '$const44.1.33']
    return $44.2.39                          ['$44.2.39']
```

Stage 7b: Generate object mode LLVM IR

If type inference fails to find Numba types for all values inside a function, the function will be compiled in object mode. The generated LLVM will be significantly longer, as the compiled code will need to make calls to the Python C API to perform basically all operations. The optimized LLVM for our example `add()` function is:

```llvm
@PyExc_SystemError = external global i8
@".const.Numba_internal_error:_object_mode_function_called_without_an_environment" =
→internal constant [73 x i8] c"Numba internal error: object mode function called
→without an environment\00"
@".const.name_'a'_is_not_defined" = internal constant [24 x i8] c"name 'a' is not
→defined\00"
@PyExc_NameError = external global i8
@".const.name_'b'_is_not_defined" = internal constant [24 x i8] c"name 'b' is not
→defined\00"

define i32 @"__main__.add$1.pyobject.pyobject"(i8** nocapture %retptr, { i8*, i32 }**
→nocapture readnone %excinfo, i8* readnone %env, i8* %arg.a, i8* %arg.b) {
entry:
  %.6 = icmp eq i8* %env, null
  br i1 %.6, label %entry.if, label %entry.endif, !prof !0

entry.if:                                         ; preds = %entry
  tail call void @PyErr_SetString(i8* @PyExc_SystemError, i8* getelementptr inbounds
→([73 x i8]* @".const.Numba_internal_error:_object_mode_function_called_without_an_
→environment", i64 0, i64 0))
  ret i32 -1

entry.endif:                                      ; preds = %entry
  tail call void @Py_IncRef(i8* %arg.a)
  tail call void @Py_IncRef(i8* %arg.b)
  %.21 = icmp eq i8* %arg.a, null
  br i1 %.21, label %B0.if, label %B0.endif, !prof !0

B0.if:                                            ; preds = %entry.endif
  tail call void @PyErr_SetString(i8* @PyExc_NameError, i8* getelementptr inbounds
→([24 x i8]* @".const.name_'a'_is_not_defined", i64 0, i64 0))
  tail call void @Py_DecRef(i8* null)
  tail call void @Py_DecRef(i8* %arg.b)
  ret i32 -1

B0.endif:                                         ; preds = %entry.endif
  %.30 = icmp eq i8* %arg.b, null
  br i1 %.30, label %B0.endif1, label %B0.endif1.1, !prof !0

B0.endif1:                                        ; preds = %B0.endif
  tail call void @PyErr_SetString(i8* @PyExc_NameError, i8* getelementptr inbounds
→([24 x i8]* @".const.name_'b'_is_not_defined", i64 0, i64 0))
  tail call void @Py_DecRef(i8* %arg.a)
  tail call void @Py_DecRef(i8* null)
  ret i32 -1

B0.endif1.1:                                      ; preds = %B0.endif
  %.38 = tail call i8* @PyNumber_Add(i8* %arg.a, i8* %arg.b)
  %.39 = icmp eq i8* %.38, null
  br i1 %.39, label %B0.endif1.1.if, label %B0.endif1.1.endif, !prof !0

B0.endif1.1.if:                                   ; preds = %B0.endif1.1
```

(continues on next page)

```
  tail call void @Py_DecRef(i8* %arg.a)
  tail call void @Py_DecRef(i8* %arg.b)
  ret i32 -1

B0.endif1.1.endif:                                ; preds = %B0.endif1.1
  tail call void @Py_DecRef(i8* %arg.b)
  tail call void @Py_DecRef(i8* %arg.a)
  tail call void @Py_IncRef(i8* %.38)
  tail call void @Py_DecRef(i8* %.38)
  store i8* %.38, i8** %retptr, align 8
  ret i32 0
}

declare void @PyErr_SetString(i8*, i8*)

declare void @Py_IncRef(i8*)

declare void @Py_DecRef(i8*)

declare i8* @PyNumber_Add(i8*, i8*)
```

The careful reader might notice several unnecessary calls to `Py_IncRef` and `Py_DecRef` in the generated code. Currently Numba isn't able to optimize those away.

Object mode compilation will also attempt to identify loops which can be extracted and statically-typed for "nopython" compilation. This process is called *loop-lifting*, and results in the creation of a hidden nopython mode function just containing the loop which is then called from the original function. Loop-lifting helps improve the performance of functions that need to access uncompilable code (such as I/O or plotting code) but still contain a time-intensive section of compilable code.

Stage 8: Compile LLVM IR to machine code

In both *object mode* and *nopython mode*, the generated LLVM IR is compiled by the LLVM JIT compiler and the machine code is loaded into memory. A Python wrapper is also created (defined in `numba.dispatcher.Dispatcher`) which can do the dynamic dispatch to the correct version of the compiled function if multiple type specializations were generated (for example, for both `float32` and `float64` versions of the same function).

The machine assembly code generated by LLVM can be dumped to the screen by setting the *NUMBA_DUMP_ASSEMBLY* environment variable to 1:

```
        .globl  __main__.add$1.int64.int64
        .align  16, 0x90
        .type   __main__.add$1.int64.int64,@function
__main__.add$1.int64.int64:
        addq    %r8, %rcx
        movq    %rcx, (%rdi)
        xorl    %eax, %eax
        retq
```

The assembly output will also include the generated wrapper function that translates the Python arguments to native data types.

7.4 Polymorphic dispatching

Functions compiled using *jit()* or *vectorize()* are open-ended: they can be called with many different input types and have to select (possibly compile on-the-fly) the right low-level specialization. We hereby explain how this mechanism is implemented.

7.4.1 Requirements

JIT-compiled functions can take several arguments and each of them is taken into account when selecting a specialization. Thus it is a form of multiple dispatch, more complex than single dispatch.

Each argument weighs in the selection based on its *Numba type*. Numba types are often more granular than Python types: for example, Numba types Numpy arrays differently depending on their dimensionality and their layout (C-contiguous, etc.).

Once a Numba type is inferred for each argument, a specialization must be chosen amongst the available ones; or, if not suitable specialization is found, a new one must be compiled. This is not a trivial decision: there can be multiple specializations compatible with a given concrete signature (for example, say a two-argument function has compiled specializations for (`float64, float64`) and (`complex64, complex64`), and it is called with (`float32, float32`)).

Therefore, there are two crucial steps in the dispatch mechanism:

1. infer the Numba types of the concrete arguments
2. select the best available specialization (or choose to compile a new one) for the inferred Numba types

Compile-time vs. run-time

This document discusses dispatching when it is done at runtime, i.e. when a JIT-compiled function is called from pure Python. In that context, performance is important. To stay in the realm of normal function call overhead in Python, the overhead of dispatching should stay under a microsecond. Of course, *the faster the better...*

When a JIT-compiled function is called from another JIT-compiled function (in *nopython mode*), the polymorphism is resolved at compile-time, using a non-performance critical mechanism, bearing zero runtime performance overhead.

Note: In practice, the performance-critical parts described here are coded in C.

7.4.2 Type resolution

The first step is therefore to infer, at call-time, a Numba type for each of the function's concrete arguments. Given the finer granularity of Numba types compared to Python types, one cannot simply lookup an object's class and key a dictionary with it to obtain the corresponding Numba type.

Instead, there is a machinery to inspect the object and, based on its Python type, query various properties to infer the appropriate Numba type. This can be more or less complex: for example, a Python `int` argument will always infer to a Numba `intp` (a pointer-sized integer), but a Python `tuple` argument can infer to multiple Numba types (depending on the tuple's size and the concrete type of each of its elements).

The Numba type system is high-level and written in pure Python; there is a pure Python machinery, based on a generic function, to do said inference (in `numba.typing.typeof`). That machinery is used for compile-time inference, e.g. on constants. Unfortunately, it is too slow for run-time value-based dispatching. It is only used as a fallback for rarely used (or difficult to infer) types, and exhibits multiple-microsecond overhead.

Typecodes

The Numba type system is really too high-level to be manipulated efficiently from C code. Therefore, the C dispatching layer uses another representation based on integer typecodes. Each Numba type gets a unique integer typecode when constructed; also, an interning system ensure no two instances of same type are created. The dispatching layer is therefore able to *eschew* the overhead of the Numba type system by working with simple integer typecodes, amenable to well-known optimizations (fast hash tables, etc.).

The goal of the type resolution step becomes: infer a Numba *typecode* for each of the function's concrete arguments. Ideally, it doesn't deal with Numba types anymore...

Hard-coded fast paths

While eschewing the abstraction and object-orientation overhead of the type system, the integer typecodes still have the same conceptual complexity. Therefore, an important technique to speed up inference is to first go through checks for the most important types, and hard-code a fast resolution for each of them.

Several types benefit from such an optimization, notably:

- basic Python scalars (`bool`, `int`, `float`, `complex`);
- basic Numpy scalars (the various kinds of integer, floating-point, complex numbers);
- Numpy arrays of certain dimensionalities and basic element types.

Each of those fast paths ideally uses a hard-coded result value or a direct table lookup after a few simple checks.

However, we can't apply that technique to all argument types; there would be an explosion of ad-hoc internal caches, and it would become difficult to maintain. Besides, the recursive application of hard-coded fast paths would not necessarily combine into a low overhead (in the nested tuple case, for example).

Fingerprint-based typecode cache

For non-so-trivial types (imagine a tuple, or a Numpy `datetime64` array, for example), the hard-coded fast paths don't match. Another mechanism then kicks in, more generic.

The principle here is to examine each argument value, as the pure Python machinery would do, and to describe its Numba type unambiguously. The difference is that *we don't actually compute a Numba type*. Instead, we compute a simple bytestring, a low-level possible denotation of that Numba type: a *fingerprint*. The fingerprint format is designed to be short and extremely simple to compute from C code (in practice, it has a bytecode-like format).

Once the fingerprint is computed, it is looked up in a cache mapping fingerprints to typecodes. The cache is a hash table, and the lookup is fast thanks to the fingerprints being generally very short (rarely more than 20 bytes).

If the cache lookup fails, the typecode must first be computed using the slow pure Python machinery. Luckily, this would only happen once: on subsequent calls, the cached typecode would be returned for the given fingerprint.

In rare cases, a fingerprint cannot be computed efficiently. This is the case for some types which cannot be easily inspected from C: for example `cffi` function pointers. Then, the slow Pure Python machinery is invoked at each function call with such an argument.

Note: Two fingerprints may denote a single Numba type. This does not make the mechanism incorrect; it only creates more cache entries.

Summary

Type resolution of a function argument involves the following mechanisms in order:

- Try a few hard-coded fast paths, for common simple types.

- If the above failed, compute a fingerprint for the argument and lookup its typecode in a cache.

- If all the above failed, invoke the pure Python machinery which will determine a Numba type for the argument (and look up its typecode).

7.4.3 Specialization selection

At the previous step, an integer typecode has been determined for each concrete argument to the JIT-compiled function. Now it remains to match that concrete signature against each of the available specializations for the function. There can be three outcomes:

- There is a satisfying best match: the corresponding specialization is then invoked (it will handle argument unboxing and other details).

- There is a tie between two or more "best matches": an exception is raised, refusing to solve the ambiguity.

- There is no satisfying match: a new specialization is compiled tailored for the concrete argument types that were inferred.

The selection works by looping over all available specializations, and computing the compatibility of each concrete argument type with the corresponding type in the specialization's intended signature. Specifically, we are interested in:

1. Whether the concrete argument type is allowed to convert implicitly to the specialization's argument type;

2. If so, at what semantic (user-visible) cost the conversion comes.

Implicit conversion rules

There are five possible kinds of implicit conversion from a source type to a destination type (note this is an asymmetric relationship):

1. *exact match*: the two types are identical; this is the ideal case, since the specialization would behave exactly as intended;

2. *same-kind promotion*: the two types belong to the same "kind" (for example `int32` and `int64` are two integer types), and the source type can be converted losslessly to the destination type (e.g. from `int32` to `int64`, but not the reverse);

3. *safe conversion*: the two types belong to different kinds, but the source type can be reasonably converted to the destination type (e.g. from `int32` to `float64`, but not the reverse);

4. *unsafe conversion*: a conversion is available from the source type to the destination type, but it may lose precision, magnitude, or another desirable quality.

5. *no conversion*: there is no correct or reasonably efficient way to convert between the two types (for example between an `int64` and a `datetime64`, or a C-contiguous array and a Fortran-contiguous array).

When a specialization is examined, the latter two cases eliminate it from the final choice: i.e. when at least one argument has *no conversion* or only an *unsafe conversion* to the signature's argument type.

Note: However, if the function is compiled with explicit signatures in the `jit()` call (and therefore it is not allowed to compile new specializations), *unsafe conversion* is allowed.

Candidates and best match

If a specialization is not eliminated by the rule above, it enters the list of *candidates* for the final choice. Those candidates are ranked by an ordered 4-uple of integers: `(number of unsafe conversions, number of safe conversions, number of same-kind promotions, number of exact matches)` (note the sum of the tuple's elements is equal to the number of arguments). The best match is then the #1 result in sorted ascending order, thereby preferring exact matches over promotions, promotions over safe conversions, safe conversions over unsafe conversions.

Implementation

The above-described mechanism works on integer typecodes, not on Numba types. It uses an internal hash table storing the possible conversion kind for each pair of compatible types. The internal hash table is in part built at startup (for built-in trivial types such as `int32`, `int64` etc.), in part filled dynamically (for arbitrarily complex types such as array types: for example to allow using a C-contiguous 2D array where a function expects a non-contiguous 2D array).

Summary

Selecting the right specialization involves the following steps:

- Examine each available specialization and match it against the concrete argument types.
- Eliminate any specialization where at least one argument doesn't offer sufficient compatibility.
- If there are remaining candidates, choose the best one in terms of preserving the types' semantics.

7.4.4 Miscellaneous

Some benchmarks of dispatch performance exist in the Numba benchmarks repository.

Some unit tests of specific aspects of the machinery are available in `numba.tests.test_typeinfer` and `numba.tests.test_typeof`. Higher-level dispatching tests are in `numba.tests.test_dispatcher`.

7.5 Notes on generators

Numba recently gained support for compiling generator functions. This document explains some of the implementation choices.

7.5.1 Terminology

For clarity, we distinguish between *generator functions* and *generators*. A generator function is a function containing one or several `yield` statements. A generator (sometimes also called "generator iterator") is the return value of a generator function; it resumes execution inside its frame each time `next()` is called.

A *yield point* is the place where a `yield` statement is called. A *resumption point* is the place just after a *yield point* where execution is resumed when `next()` is called again.

7.5.2 Function analysis

Suppose we have the following simple generator function:

```
def gen(x, y):
    yield x + y
    yield x - y
```

Here is its CPython bytecode, as printed out using `dis.dis()`:

```
7           0 LOAD_FAST                0 (x)
            3 LOAD_FAST                1 (y)
            6 BINARY_ADD
            7 YIELD_VALUE
            8 POP_TOP

8           9 LOAD_FAST                0 (x)
           12 LOAD_FAST                1 (y)
           15 BINARY_SUBTRACT
           16 YIELD_VALUE
           17 POP_TOP
           18 LOAD_CONST               0 (None)
           21 RETURN_VALUE
```

When compiling this function with *NUMBA_DUMP_IR* set to 1, the following information is printed out:

```
--------------------------------IR DUMP: gen----------------------------------
label 0:
    x = arg(0, name=x)                      ['x']
    y = arg(1, name=y)                      ['y']
    $0.3 = x + y                            ['$0.3', 'x', 'y']
    $0.4 = yield $0.3                       ['$0.3', '$0.4']
    del $0.4                                []
    del $0.3                                []
    $0.7 = x - y                            ['$0.7', 'x', 'y']
    del y                                   []
    del x                                   []
    $0.8 = yield $0.7                       ['$0.7', '$0.8']
    del $0.8                                []
    del $0.7                                []
    $const0.9 = const(NoneType, None)       ['$const0.9']
    $0.10 = cast(value=$const0.9)           ['$0.10', '$const0.9']
    del $const0.9                           []
    return $0.10                            ['$0.10']
----------------------------GENERATOR INFO: gen-------------------------------
generator state variables: ['$0.3', '$0.7', 'x', 'y']
yield point #1: live variables = ['x', 'y'], weak live variables = ['$0.3']
yield point #2: live variables = [], weak live variables = ['$0.7']
```

What does it mean? The first part is the Numba IR, as already seen in *Stage 2: Generate the Numba IR*. We can see the two yield points (`yield $0.3` and `yield $0.7`).

The second part shows generator-specific information. To understand it we have to understand what suspending and resuming a generator means.

When suspending a generator, we are not merely returning a value to the caller (the operand of the `yield` statement). We also have to save the generator's *current state* in order to resume execution. In trivial use cases, perhaps the CPU's register values or stack slots would be preserved until the next call to next(). However, any non-trivial case will hopelessly clobber those values, so we have to save them in a well-defined place.

What are the values we need to save? Well, in the context of the Numba Intermediate Representation, we must save all *live variables* at each yield point. These live variables are computed thanks to the control flow graph.

Once live variables are saved and the generator is suspended, resuming the generator simply involves the inverse operation: the live variables are restored from the saved generator state.

Note: It is the same analysis which helps insert Numba `del` instructions where appropriate.

Let's go over the generator info again:

```
generator state variables: ['$0.3', '$0.7', 'x', 'y']
yield point #1: live variables = ['x', 'y'], weak live variables = ['$0.3']
yield point #2: live variables = [], weak live variables = ['$0.7']
```

Numba has computed the union of all live variables (denoted as "state variables"). This will help define the layout of the *generator structure*. Also, for each yield point, we have computed two sets of variables:

- the *live variables* are the variables which are used by code following the resumption point (i.e. after the `yield` statement)

- the *weak live variables* are variables which are del'ed immediately after the resumption point; they have to be saved in *object mode*, to ensure proper reference cleanup

7.5.3 The generator structure

Layout

Function analysis helps us gather enough information to define the layout of the generator structure, which will store the entire execution state of a generator. Here is a sketch of the generator structure's layout, in pseudo-code:

```
struct gen_struct_t {
   int32_t resume_index;
   struct gen_args_t {
      arg_0_t arg0;
      arg_1_t arg1;
      ...
      arg_N_t argN;
   }
   struct gen_state_t {
      state_0_t state_var0;
      state_1_t state_var1;
      ...
      state_N_t state_varN;
   }
}
```

Let's describe those fields in order.

- The first member, the *resume index*, is an integer telling the generator at which resumption point execution must resume. By convention, it can have two special values: 0 means execution must start at the beginning of the generator (i.e. the first time `next()` is called); -1 means the generator is exhausted and resumption must

immediately raise StopIteration. Other values indicate the yield point's index starting from 1 (corresponding to the indices shown in the generator info above).

- The second member, the *arguments structure* is read-only after it is first initialized. It stores the values of the arguments the generator function was called with. In our example, these are the values of `x` and `y`.

- The third member, the *state structure*, stores the live variables as computed above.

Concretely, our example's generator structure (assuming the generator function is called with floating-point numbers) is then:

```
struct gen_struct_t {
    int32_t resume_index;
    struct gen_args_t {
        double arg0;
        double arg1;
    }
    struct gen_state_t {
        double $0.3;
        double $0.7;
        double x;
        double y;
    }
}
```

Note that here, saving `x` and `y` is redundant: Numba isn't able to recognize that the state variables `x` and `y` have the same value as `arg0` and `arg1`.

Allocation

How does Numba ensure the generator structure is preserved long enough? There are two cases:

- When a Numba-compiled generator function is called from a Numba-compiled function, the structure is allocated on the stack by the callee. In this case, generator instantiation is practically costless.

- When a Numba-compiled generator function is called from regular Python code, a CPython-compatible wrapper is instantiated that has the right amount of allocated space to store the structure, and whose `tp_iternext` slot is a wrapper around the generator's native code.

7.5.4 Compiling to native code

When compiling a generator function, three native functions are actually generated by Numba:

- An initialization function. This is the function corresponding to the generator function itself: it receives the function arguments and stores them inside the generator structure (which is passed by pointer). It also initialized the *resume index* to 0, indicating that the generator hasn't started yet.

- A next() function. This is the function called to resume execution inside the generator. Its single argument is a pointer to the generator structure and it returns the next yielded value (or a special exit code is used if the generator is exhausted, for quick checking when called from Numba-compiled functions).

- An optional finalizer. In object mode, this function ensures that all live variables stored in the generator state are decref'ed, even if the generator is destroyed without having been exhausted.

The next() function

The next() function is the least straight-forward of the three native functions. It starts with a trampoline which dispatches execution to the right resume point depending on the *resume index* stored in the generator structure. Here is how the function start may look like in our example:

```
define i32 @"__main__.gen.next"(
   double* nocapture %retptr,
   { i8*, i32 }** nocapture readnone %excinfo,
   i8* nocapture readnone %env,
   { i32, { double, double }, { double, double, double, double } }* nocapture %arg.
→gen)
{
  entry:
     %gen.resume_index = getelementptr { i32, { double, double }, { double, double,␣
→double, double } }* %arg.gen, i64 0, i32 0
     %.47 = load i32* %gen.resume_index, align 4
     switch i32 %.47, label %stop_iteration [
       i32 0, label %B0
       i32 1, label %generator_resume1
       i32 2, label %generator_resume2
     ]

  ; rest of the function snipped
```

(uninteresting stuff trimmed from the LLVM IR to make it more readable)

We recognize the pointer to the generator structure in `%arg.gen`. The trampoline switch has three targets (one for each *resume index* 0, 1 and 2), and a fallback target label named `stop_iteration`. Label `B0` represents the function's start, `generator_resume1` (resp. `generator_resume2`) is the resumption point after the first (resp. second) yield point.

After generation by LLVM, the whole native assembly code for this function may look like this (on x86-64):

```
        .globl  __main__.gen.next
        .align  16, 0x90
__main__.gen.next:
        movl    (%rcx), %eax
        cmpl    $2, %eax
        je      .LBB1_5
        cmpl    $1, %eax
        jne     .LBB1_2
        movsd   40(%rcx), %xmm0
        subsd   48(%rcx), %xmm0
        movl    $2, (%rcx)
        movsd   %xmm0, (%rdi)
        xorl    %eax, %eax
        retq
.LBB1_5:
        movl    $-1, (%rcx)
        jmp     .LBB1_6
.LBB1_2:
        testl   %eax, %eax
        jne     .LBB1_6
        movsd   8(%rcx), %xmm0
        movsd   16(%rcx), %xmm1
        movaps  %xmm0, %xmm2
        addsd   %xmm1, %xmm2
```

(continues on next page)

```
        movsd    %xmm1, 48(%rcx)
        movsd    %xmm0, 40(%rcx)
        movl     $1, (%rcx)
        movsd    %xmm2, (%rdi)
        xorl     %eax, %eax
        retq
.LBB1_6:
        movl     $-3, %eax
        retq
```

Note the function returns 0 to indicate a value is yielded, -3 to indicate StopIteration. `%rcx` points to the start of the generator structure, where the resume index is stored.

7.6 Notes on Numba Runtime

The *Numba Runtime (NRT)* provides the language runtime to the *nopython mode* Python subset. NRT is a standalone C library with a Python binding. This allows NPM runtime feature to be used without the GIL. Currently, the only language feature implemented in NRT is memory management.

7.6.1 Memory Management

NRT implements memory management for NPM code. It uses *atomic reference count* for threadsafe, deterministic memory management. NRT maintains a separate `MemInfo` structure for storing information about each allocation.

Cooperating with CPython

For NRT to cooperate with CPython, the NRT python binding provides adaptors for converting python objects that export a memory region. When such an object is used as an argument to a NPM function, a new `MemInfo` is created and it acquires a reference to the Python object. When a NPM value is returned to the Python interpreter, the associated `MemInfo` (if any) is checked. If the `MemInfo` references a Python object, the underlying Python object is released and returned instead. Otherwise, the `MemInfo` is wrapped in a Python object and returned. Additional process maybe required depending on the type.

The current implementation supports Numpy array and any buffer-exporting types.

Compiler-side Cooperation

NRT reference counting requires the compiler to emit incref/decref operations according to the usage. When the reference count drops to zero, the compiler must call the destructor routine in NRT.

Optimizations

The compiler is allowed to emit incref/decref operations naively. It relies on an optimization pass that to remove the redundant reference count operations.

The optimization pass runs on block level to avoid control flow analysis. It depends on LLVM function optimization pass to simplify the control flow, stack-to-register, and simplify instructions. It works by matching and removing incref and decref pairs within each block.

Quirks

Since the *refcount optimization pass* requires LLVM function optimization pass, the pass works on the LLVM IR as text. The optimized IR is then materialized again as a new LLVM in-memory bitcode object.

Debugging Leaks

To debug reference leaks in NRT MemInfo, each MemInfo python object has a `.refcount` attribute for inspection. To get the MemInfo from a ndarray allocated by NRT, use the `.base` attribute.

To debug memory leaks in NRT, the `numba.runtime.rtsys` defines `.get_allocation_stats()`. It returns a namedtuple containing the number of allocation and deallocation since the start of the program. Checking that the allocation and deallocation counters are matching is the simplest way to know if the NRT is leaking.

Debugging Leaks in C

The start of numba/runtime/nrt.h has these lines:

```
/* Debugging facilities - enabled at compile-time */
/* #undef NDEBUG */
#if 0
#   define NRT_Debug(X) X
#else
#   define NRT_Debug(X) if (0) { X; }
#endif
```

Undefining NDEBUG (uncomment the `#undef NDEBUG` line) enables the assertion check in NRT.

Enabling the NRT_Debug (replace `#if 0` with `#if 1`) turns on debug print inside NRT.

7.6.2 Recursion Support

During the compilation of a pair of mutually recursive functions, one of the functions will contain unresolved symbol references since the compiler handles one function at a time. The memory for the unresolved symbols is allocated and initialized to the address of the *unresolved symbol abort* function (`nrt_unresolved_abort`) just before the machine code is generated by LLVM. These symbols are tracked and resolved as new functions are compiled. If a bug prevents the resolution of these symbols, the abort function will be called, raising a `RuntimeError` exception.

The *unresolved symbol abort* function is defined in the NRT with a zero-argument signature. The caller is safe to call it with arbitrary number of arguments. Therefore, it is safe to be used inplace of the intended callee.

7.6.3 Using the NRT from C code

Externally compiled C code should use the `NRT_api_functions` struct as a function table to access the NRT API. The struct is defined in numba/core/runtime/nrt_external.h. Users can use the utility function `numba.extending.include_path()` to determine the include directory for Numba provided C headers.

Listing 1: *numba/core/runtime/nrt_external.h*

```
#ifndef NUMBA_NRT_EXTERNAL_H_
#define NUMBA_NRT_EXTERNAL_H_

#include <stdlib.h>
```

(continues on next page)

```c
typedef struct MemInfo NRT_MemInfo;

typedef void NRT_managed_dtor(void *data);


typedef struct {
    /* Methods to create MemInfos.

    MemInfos are like smart pointers for objects that are managed by the Numba.
    */

    /* Allocate memory

    *nbytes* is the number of bytes to be allocated

    Returning a new reference.
    */
    NRT_MemInfo* (*allocate)(size_t nbytes);

    /* Convert externally allocated memory into a MemInfo.

    *data* is the memory pointer
    *dtor* is the deallocator of the memory
    */
    NRT_MemInfo* (*manage_memory)(void *data, NRT_managed_dtor dtor);

    /* Acquire a reference */
    void (*acquire)(NRT_MemInfo* mi);

    /* Release a reference */
    void (*release)(NRT_MemInfo* mi);

    /* Get MemInfo data pointer */
    void* (*get_data)(NRT_MemInfo* mi);

} NRT_api_functions;



#endif /* NUMBA_NRT_EXTERNAL_H_ */
```

Inside Numba compiled code, the `numba.core.unsafe.nrt.NRT_get_api()` intrinsic can be used to obtain a pointer to the `NRT_api_functions`.

Here is an example that uses the `nrt_external.h`:

```c
#include <stdio.h>
#include "numba/core/runtime/nrt_external.h"

void my_dtor(void *ptr) {
    free(ptr);
}

NRT_MemInfo* my_allocate(NRT_api_functions *nrt) {
    /* heap allocate some memory */
    void * data = malloc(10);
```

```
    /* wrap the allocated memory; yield a new reference */
    NRT_MemInfo *mi = nrt->manage_memory(data, my_dtor);
    /* acquire reference */
    nrt->acquire(mi);
    /* release reference */
    nrt->release(mi);
    return mi;
}
```

7.6.4 Future Plan

The plan for NRT is to make a standalone shared library that can be linked to Numba compiled code, including use within the Python interpreter and without the Python interpreter. To make that work, we will be doing some refactoring:

- numba NPM code references statically compiled code in "helperlib.c". Those functions should be moved to NRT.

7.7 Using the Numba Rewrite Pass for Fun and Optimization

7.7.1 Overview

This section introduces intermediate representation (IR) rewrites, and how they can be used to implement optimizations.

As discussed earlier in "*Stage 6a: Rewrite typed IR*", rewriting the Numba IR allows us to perform optimizations that would be much more difficult to perform at the lower LLVM level. Similar to the Numba type and lowering subsystems, the rewrite subsystem is user extensible. This extensibility affords Numba the possibility of supporting a wide variety of domain-specific optimizations (DSO's).

The remaining subsections detail the mechanics of implementing a rewrite, registering a rewrite with the rewrite registry, and provide examples of adding new rewrites, as well as internals of the array expression optimization pass. We conclude by reviewing some use cases exposed in the examples, as well as reviewing any points where developers should take care.

7.7.2 Rewriting Passes

Rewriting passes have a simple `match()` and `apply()` interface. The division between matching and rewriting follows how one would define a term rewrite in a declarative domain-specific languages (DSL's). In such DSL's, one may write a rewrite as follows:

```
<match> => <replacement>
```

The `<match>` and `<replacement>` symbols represent IR term expressions, where the left-hand side presents a pattern to match, and the right-hand side an IR term constructor to build upon matching. Whenever the rewrite matches an IR pattern, any free variables in the left-hand side are bound within a custom environment. When applied, the rewrite uses the pattern matching environment to bind any free variables in the right-hand side.

As Python is not commonly used in a declarative capacity, Numba uses object state to handle the transfer of information between the matching and application steps.

The `Rewrite` Base Class

class Rewrite

> The *Rewrite* class simply defines an abstract base class for Numba rewrites. Developers should define rewrites as subclasses of this base type, overloading the *match()* and *apply()* methods.
>
> **pipeline**
>
> > The pipeline attribute contains the `numba.compiler.Pipeline` instance that is currently compiling the function under consideration for rewriting.
>
> **__init__**(*self*, *pipeline*, **args*, ***kws*)
>
> > The base constructor for rewrites simply stashes its arguments into attributes of the same name. Unless being used in debugging or testing, rewrites should only be constructed by the `RewriteRegistry` in the `RewriteRegistry.apply()` method, and the construction interface should remain stable (though the pipeline will commonly contain just about everything there is to know).
>
> **match**(*self*, *block*, *typemap*, *callmap*)
>
> > The *match()* method takes four arguments other than *self*:
> >
> > - *func_ir*: This is an instance of `numba.ir.FunctionIR` for the function being rewritten.
> >
> > - *block*: This is an instance of `numba.ir.Block`. The matching method should iterate over the instructions contained in the `numba.ir.Block.body` member.
> >
> > - *typemap*: This is a Python `dict` instance mapping from symbol names in the IR, represented as strings, to Numba types.
> >
> > - *callmap*: This is another `dict` instance mapping from calls, represented as `numba.ir.Expr` instances, to their corresponding call site type signatures, represented as a `numba.typing.templates.Signature` instance.
> >
> > The *match()* method should return a `bool` result. A `True` result should indicate that one or more matches were found, and the *apply()* method will return a new replacement `numba.ir.Block` instance. A `False` result should indicate that no matches were found, and subsequent calls to *apply()* will return undefined or invalid results.
>
> **apply**(*self*)
>
> > The *apply()* method should only be invoked following a successful call to *match()*. This method takes no additional parameters other than *self*, and should return a replacement `numba.ir.Block` instance.
> >
> > As mentioned above, the behavior of calling *apply()* is undefined unless *match()* has already been called and returned `True`.

Subclassing `Rewrite`

Before going into the expectations for the overloaded methods any *Rewrite* subclass must have, let's step back a minute to review what is taking place here. By providing an extensible compiler, Numba opens itself to user-defined code generators which may be incomplete, or worse, incorrect. When a code generator goes awry, it can cause abnormal program behavior or early termination. User-defined rewrites add a new level of complexity because they must not only generate correct code, but the code they generate should ensure that the compiler does not get stuck in a match/apply loop. Non-termination by the compiler will directly lead to non-termination of user function calls.

There are several ways to help ensure that a rewrite terminates:

- *Typing*: A rewrite should generally attempt to decompose composite types, and avoid composing new types. If the rewrite is matching a specific type, changing expression types to a lower-level type will ensure they will no long match after the rewrite is applied.

- *Special instructions*: A rewrite may synthesize custom operators or use special functions in the target IR. This technique again generates code that is no longer within the domain of the original match, and the rewrite will terminate.

In the "*Case study: Array Expressions*" subsection, below, we'll see how the array expression rewriter uses both of these techniques.

Overloading `Rewrite.match()`

Every rewrite developer should seek to have their implementation of *match()* return a `False` value as quickly as possible. Numba is a just-in-time compiler, and adding compilation time ultimately adds to the user's run time. When a rewrite returns `False` for a given block, the registry will no longer process that block with that rewrite, and the compiler is that much closer to proceeding to lowering.

This need for timeliness has to be balanced against collecting the necessary information to make a match for a rewrite. Rewrite developers should be comfortable adding dynamic attributes to their subclasses, and then having these new attributes guide construction of the replacement basic block.

Overloading `Rewrite.apply()`

The *apply()* method should return a replacement `numba.ir.Block` instance to replace the basic block that contained a match for the rewrite. As mentioned above, the IR built by *apply()* methods should preserve the semantics of the user's code, but also seek to avoid generating another match for the same rewrite or set of rewrites.

7.7.3 The Rewrite Registry

When you want to include a rewrite in the rewrite pass, you should register it with the rewrite registry. The `numba.rewrites` module provides both the abstract base class and a class decorator for hooking into the Numba rewrite subsystem. The following illustrates a stub definition of a new rewrite:

```python
from numba import rewrites

@rewrites.register_rewrite
class MyRewrite(rewrites.Rewrite):

    def match(self, block, typemap, calltypes):
        raise NotImplementedError("FIXME")

    def apply(self):
        raise NotImplementedError("FIXME")
```

Developers should note that using the class decorator as shown above will register a rewrite at import time. It is the developer's responsibility to ensure their extensions are loaded before compilation starts.

7.7.4 Case study: Array Expressions

This subsection looks at the array expression rewriter in more depth. The array expression rewriter, and most of its support functionality, are found in the `numba.npyufunc.array_exprs` module. The rewriting pass itself is implemented in the `RewriteArrayExprs` class. In addition to the rewriter, the `array_exprs` module includes a function for lowering array expressions, `_lower_array_expr()`. The overall optimization process is as follows:

- `RewriteArrayExprs.match()`: The rewrite pass looks for two or more array operations that form an array expression.

- `RewriteArrayExprs.apply()`: Once an array expression is found, the rewriter replaces the individual array operations with a new kind of IR expression, the `arrayexpr`.

- `numba.npyufunc.array_exprs._lower_array_expr()`: During lowering, the code generator calls `_lower_array_expr()` whenever it finds an `arrayexpr` IR expression.

More details on each step of the optimization are given below.

The `RewriteArrayExprs.match()` method

The array expression optimization pass starts by looking for array operations, including calls to supported `ufunc`'s and user-defined *DUFunc*'s. Numba IR follows the conventions of a static single assignment (SSA) language, meaning that the search for array operators begins with looking for assignment instructions.

When the rewriting pass calls the `RewriteArrayExprs.match()` method, it first checks to see if it can trivially reject the basic block. If the method determines the block to be a candidate for matching, it sets up the following state variables in the rewrite object:

- *crnt_block*: The current basic block being matched.

- *typemap*: The *typemap* for the function being matched.

- *matches*: A list of variable names that reference array expressions.

- *array_assigns*: A map from assignment variable names to the actual assignment instructions that define the given variable.

- *const_assigns*: A map from assignment variable names to the constant valued expression that defines the constant variable.

At this point, the match method iterates iterates over the assignment instructions in the input basic block. For each assignment instruction, the matcher looks for one of two things:

- Array operations: If the right-hand side of the assignment instruction is an expression, and the result of that expression is an array type, the matcher checks to see if the expression is either a known array operation, or a call to a universal function. If an array operator is found, the matcher stores the left-hand variable name and the whole instruction in the *array_assigns* member. Finally, the matcher tests to see if any operands of the array operation have also been identified as targets of other array operations. If one or more operands are also targets of array operations, then the matcher will also append the left-hand side variable name to the *matches* member.

- Constants: Constants (even scalars) can be operands to array operations. Without worrying about the constant being apart of an array expression, the matcher stores constant names and values in the *const_assigns* member.

The end of the matching method simply checks for a non-empty *matches* list, returning `True` if there were one or more matches, and `False` when *matches* is empty.

The `RewriteArrayExprs.apply()` method

When one or matching array expressions are found by `RewriteArrayExprs.match()`, the rewriting pass will call `RewriteArrayExprs.apply()`. The apply method works in two passes. The first pass iterates over the matches found, and builds a map from instructions in the old basic block to new instructions in the new basic block. The second pass iterates over the instructions in the old basic block, copying instructions that are not changed by the rewrite, and replacing or deleting instructions that were identified by the first pass.

The `RewriteArrayExprs._handle_matches()` implements the first pass of the code generation portion of the rewrite. For each match, this method builds a special IR expression that contains an expression tree for the array expression. To compute the leaves of the expression tree, the `_handle_matches()` method iterates over the operands of the identified root operation. If the operand is another array operation, it is translated into an expression sub-tree. If the operand is a constant, `_handle_matches()` copies the constant value. Otherwise, the operand

---

is marked as being used by an array expression. As the method builds array expression nodes, it builds a map from old instructions to new instructions (*replace_map*), as well as sets of variables that may have moved (*used_vars*), and variables that should be removed altogether (*dead_vars*). These three data structures are returned back to the calling `RewriteArrayExprs.apply()` method.

The remaining part of the `RewriteArrayExprs.apply()` method iterates over the instructions in the old basic block. For each instruction, this method either replaces, deletes, or duplicates that instruction based on the results of `RewriteArrayExprs._handle_matches()`. The following list describes how the optimization handles individual instructions:

- When an instruction is an assignment, `apply()` checks to see if it is in the replacement instruction map. When an assignment instruction is found in the instruction map, `apply()` must then check to see if the replacement instruction is also in the replacement map. The optimizer continues this check until it either arrives at a `None` value or an instruction that isn't in the replacement map. Instructions that have a replacement that is `None` are deleted. Instructions that have a non-`None` replacement are replaced. Assignment instructions not in the replacement map are appended to the new basic block with no changes made.

- When the instruction is a delete instruction, the rewrite checks to see if it deletes a variable that may still be used by a later array expression, or if it deletes a dead variable. Delete instructions for used variables are added to a map of deferred delete instructions that `apply()` uses to move them past any uses of that variable. The loop copies delete instructions for non-dead variables, and ignores delete instructions for dead variables (effectively removing them from the basic block).

- All other instructions are appended to the new basic block.

Finally, the `apply()` method returns the new basic block for lowering.

The `_lower_array_expr()` function

If we left things at just the rewrite, then the lowering stage of the compiler would fail, complaining it doesn't know how to lower `arrayexpr` operations. We start by hooking a lowering function into the target context whenever the `RewriteArrayExprs` class is instantiated by the compiler. This hook causes the lowering pass to call `_lower_array_expr()` whenever it encounters an `arrayexr` operator.

This function has two steps:

- Synthesize a Python function that implements the array expression: This new Python function essentially behaves like a Numpy `ufunc`, returning the result of the expression on scalar values in the broadcasted array arguments. The lowering function accomplishes this by translating from the array expression tree into a Python AST.

- Compile the synthetic Python function into a kernel: At this point, the lowering function relies on existing code for lowering ufunc and DUFunc kernels, calling `numba.targets.numpyimpl.numpy_ufunc_kernel()` after defining how to lower calls to the synthetic function.

The end result is similar to loop lifting in Numba's object mode.

7.7.5 Conclusions and Caveats

We have seen how to implement rewrites in Numba, starting with the interface, and ending with an actual optimization. The key points of this section are:

- When writing a good plug-in, the matcher should try to get a go/no-go result as soon as possible.

- The rewrite application portion can be more computationally expensive, but should still generate code that won't cause infinite loops in the compiler.

- We use object state to communicate any results of matching to the rewrite application pass.

7.8 Live Variable Analysis

(Related issue https://github.com/numba/numba/pull/1611)

Numba uses reference-counting for garbage collection, a technique that requires cooperation by the compiler. The Numba IR encodes the location where a decref must be inserted. These locations are determined by live variable analysis. The corresponding source code is the `_insert_var_dels()` method in https://github.com/numba/numba/blob/master/numba/interpreter.py.

In Python semantic, once a variable is defined inside a function, it is alive until the variable is explicitly deleted or the function scope is ended. However, Numba analyzes the code to determine the minimum bound of the lifetime of each variable by its definition and usages during compilation. As soon as a variable is unreachable, a `del` instruction is inserted at the closest basic-block (either at the start of the next block(s) or at the end of the current block). This means variables can be released earlier than in regular Python code.

The behavior of the live variable analysis affects memory usage of the compiled code. Internally, Numba does not differentiate temporary variables and user variables. Since each operation generates at least one temporary variable, a function can accumulate a high number of temporary variables if they are not released as soon as possible. Our generator implementation can benefit from early releasing of variables, which reduces the size of the state to suspend at each yield point.

7.8.1 Notes on behavior of the live variable analysis

Variable deleted before definition

(Related issue: https://github.com/numba/numba/pull/1738)

When a variable lifetime is confined within the loop body (its definition and usage does not escape the loop body), like:

```python
def f(arr):
    # BB 0
    res = 0
    # BB 1
    for i in (0, 1):
        # BB 2
        t = arr[i]
        if t[i] > 1:
            # BB 3
            res += t[i]
    # BB 4
    return res
```

Variable `t` is never referenced outside of the loop. A `del` instruction is emitted for `t` at the head of the loop (BB 1) before a variable is defined. The reason is obvious once we know the control flow graph:

```
            +-----------------------------> BB4
            |
            |
BB 0 --> BB 1  -->  BB 2 ---> BB 3
            ^            |          |
            |            V          V
            +--------------------+
```
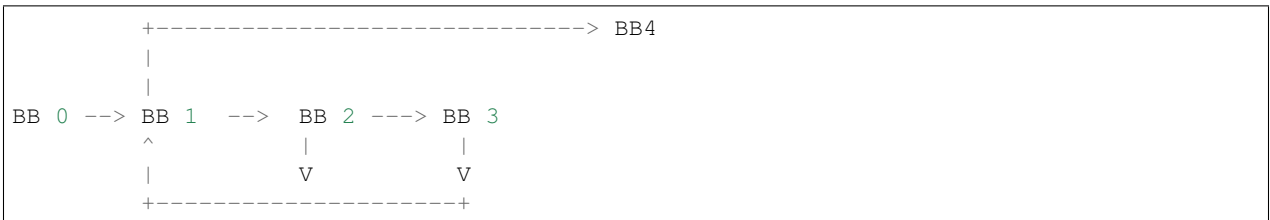
Variable `t` is defined in BB 1. In BB 2, the evaluation of `t[i] > 1` uses `t`, which is the last use if execution takes the false branch and goto BB 1. In BB 3, `t` is only used in `res += t[i]`, which is the last use if execution takes the

true branch. Because BB 3, an outgoing branch of BB 2 uses `t`, `t` must be deleted at the common predecessor. The closest point is BB 1, which does not have `t` defined from the incoming edge of BB 0.

Alternatively, if `t` is deleted at BB 4, we will still have to delete the variable before its definition because BB4 can be executed without executing the loop body (BB 2 and BB 3), where the variable is defined.

7.9 Listings

This shows listings from compiler internal registries (e.g. lowering definitions). The information is provided as developer reference. When possible, links to source code are provided via github links.

7.9.1 New style listings

The following listings are generated from `numba.help.inspector.write_listings()`. Users can run `python -m numba.help.inspector --format=rst <package>` to recreate the the documentation.

Listings for builtins

builtins

`builtins.`**`abs`**`()`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`
- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`builtins.`**`chr`**`()`

- defined by `ol_chr(i)` at numba/cpython/unicode.py:2338-2343

`builtins.`**`divmod`**`()`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`builtins.`**`hash`**`()`

- defined by `hash_overload(obj)` at numba/cpython/hashing.py:46-50

`builtins.`**`iter`**`()`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`builtins.`**`len`**`()`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`
- defined by `literal_list_len(lst)` at numba/cpython/listobj.py:1218-1223
- defined by `unicode_len(s)` at numba/cpython/unicode.py:442-447
- defined by `charseq_len(s)` at numba/cpython/charseq.py:348-369
- defined by `impl_len(d)` at numba/typed/dictobject.py:658-668

- defined by `literalstrkeydict_impl_len(d)` at numba/typed/dictobject.py:1250-1255

- defined by `impl_len(l)` at numba/typed/listobject.py:402-410

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

builtins.**max**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

- defined by `indval_max(indval1, indval2)` at numba/cpython/builtins.py:515-523

- defined by `iterable_max(iterable)` at numba/cpython/builtins.py:548-550

builtins.**min**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

- defined by `indval_min(indval1, indval2)` at numba/cpython/builtins.py:504-512

- defined by `iterable_min(iterable)` at numba/cpython/builtins.py:543-545

builtins.**next**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

builtins.**ord**()

- defined by `ol_ord(c)` at numba/cpython/unicode.py:2298-2307

builtins.**pow**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

builtins.**print**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

builtins.**round**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

builtins.**sorted**()

- defined by `ol_sorted(iterable, key=None, reverse=False)` at numba/cpython/listobj.py:1100-1113

Not showing 27 unsupported functions.

supported = 14 / 41 = 34.15%

Listings for math

math

math.**acos**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**acosh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**asin**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**asinh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**atan**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**atan2**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**atanh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**ceil**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**copysign**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**cos**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**cosh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**degrees**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**erf**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**erfc**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**exp**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**expm1**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**fabs**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**floor**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**frexp**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**gamma**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**gcd**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**hypot**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**isfinite**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**isinf**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**isnan**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**ldexp**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**lgamma**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

math.**log**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**log10**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**log1p**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**pow**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**radians**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**sin**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**sinh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**sqrt**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**tan**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**tanh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

math.**trunc**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

Not showing 7 unsupported functions.

supported = 38 / 45 = 84.44%

Listings for cmath

cmath

cmath.**acos**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**acosh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**asin**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**asinh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**atan**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**atanh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**cos**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**cosh**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**exp**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**isfinite**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**isinf**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**isnan**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**log**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**log10**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

cmath.**phase**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**polar**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**rect**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**sin**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**sinh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**sqrt**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**tan**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

cmath.**tanh**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 1 unsupported functions.

supported = 22 / 23 = 95.65%

Listings for numpy

numpy

numpy.**all**()

> • defined by np_all(a) at numba/np/arraymath.py:776-785

numpy.**amax**()

> • defined by <class 'abc.Numpy_redirect_max'>

numpy.**amin**()

> • defined by <class 'abc.Numpy_redirect_min'>

numpy.**angle**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

---

numpy.**any**()

> • defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.**append**()

> • defined by np_append(arr, values, axis=None) at numba/np/arrayobj.py:1709-1732

numpy.**arange**()

> • defined   by   np_arange(start, stop=None, step=None, dtype=None)   at
> numba/np/arrayobj.py:3842-3908

numpy.**argmax**()

> • defined by <class 'abc.Numpy_redirect_argmax'>

numpy.**argmin**()

> • defined by <class 'abc.Numpy_redirect_argmin'>

numpy.**argsort**()

> • defined by <class 'abc.Numpy_redirect_argsort'>

numpy.**argwhere**()

> • defined by np_argwhere(a) at numba/np/arraymath.py:2789-2821

numpy.**around**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**array**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**array_equal**()

> • defined by np_array_equal(a, b) at numba/np/arraymath.py:3413-3432

numpy.**asarray**()

> • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.**ascontiguousarray**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**asfortranarray**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**atleast_1d**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**atleast_2d**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**atleast_3d**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**bartlett**()

- defined by `window_generator.<locals>.window_overload(M)` at numba/np/arraymath.py:4183-4195

numpy.**bincount**()

- defined by `np_bincount(a, weights=None, minlength=0)` at numba/np/arraymath.py:3445-3500

numpy.**blackman**()

- defined by `window_generator.<locals>.window_overload(M)` at numba/np/arraymath.py:4183-4195

numpy.**column_stack**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**concatenate**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**convolve**()

- defined by `np_convolve(a, v)` at numba/np/arraymath.py:3994-4015

numpy.**copy**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**corrcoef**()

- defined by `np_corrcoef(x, y=None, rowvar=True)` at numba/np/arraymath.py:2752-2782

numpy.**correlate**()

- defined by `_np_correlate(a, v)` at numba/np/arraymath.py:3946-3991

numpy.**count_nonzero**()

- defined by `np_count_nonzero(arr, axis=None)` at numba/np/arraymath.py:3306-3320

numpy.**cov**()

- defined by `np_cov(m, y=None, rowvar=True, bias=False, ddof=None)` at numba/np/arraymath.py:2693-2749

numpy.**cross**()

- defined by `np_cross(a, b)` at numba/np/arraymath.py:4364-4386

numpy.**cumprod**()

- defined by `<class 'abc.Numpy_redirect_cumprod'>`

numpy.**cumsum**()

- defined by `<class 'abc.Numpy_redirect_cumsum'>`

numpy.**delete**()

- defined by `np_delete(arr, obj)` at numba/np/arraymath.py:3327-3371

numpy.**diag**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**diff**()

> • defined by np_diff_impl(a, n=1) at numba/np/arraymath.py:3374-3410

numpy.**digitize**()

> • defined by np_digitize(x, bins, right=False) at numba/np/arraymath.py:3568-3694

numpy.**dot**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**dstack**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**ediff1d**()

> • defined   by   np_ediff1d(ary, to_end=None, to_begin=None)   at
> numba/np/arraymath.py:1741-1796

numpy.**empty**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**empty_like**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**expand_dims**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.**extract**()

> • defined by np_extract(condition, arr) at numba/np/arraymath.py:4073-4096

numpy.**eye**()

> • defined   by   numpy_eye(N, M=None, k=0, dtype=<class 'float'>)   at
> numba/np/arrayobj.py:3681-3704

numpy.**fill_diagonal**()

> • defined by np_fill_diagonal(a, val, wrap=False) at numba/np/arraymath.py:2926-2956

numpy.**flatnonzero**()

> • defined by np_flatnonzero(a) at numba/np/arraymath.py:2824-2839

numpy.**flip**()

> • defined by np_flip(a) at numba/np/arrayobj.py:4953-4964

numpy.**fliplr**()

> • defined by np_flip_lr(a) at numba/np/arrayobj.py:4891-4905

numpy.**flipud**()

- defined by `np_flip_ud(a)` at [numba/np/arrayobj.py:4908-4922](#)

numpy.**frombuffer**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**full**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**full_like**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**hamming**()

- defined   by   `window_generator.<locals>.window_overload(M)`   at
  [numba/np/arraymath.py:4183-4195](#)

numpy.**hanning**()

- defined   by   `window_generator.<locals>.window_overload(M)`   at
  [numba/np/arraymath.py:4183-4195](#)

numpy.**histogram**()

- defined by `np_histogram(a, bins=10, range=None)` at [numba/np/arraymath.py:3700-3780](#)

numpy.**hstack**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**identity**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**imag**()

- defined by `np_imag(a)` at [numba/np/arraymath.py:3295-3300](#)

numpy.**interp**()

- defined by `np_interp(x, xp, fp)` at [numba/np/arraymath.py:2444-2505](#)

numpy.**kaiser**()

- defined by `np_kaiser(M, beta)` at [numba/np/arraymath.py:4301-4320](#)

numpy.**kron**()

- defined by `kron_impl(a, b)` at [numba/np/linalg.py:2722-2776](#)

numpy.**linspace**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

numpy.**max**()
  Alias to: `numpy.amax`

- defined by `<class 'abc.Numpy_redirect_max'>`

numpy.**mean**()

- defined by <class 'abc.Numpy_redirect_mean'>

numpy.**median**()

- defined by np_median(a) at numba/np/arraymath.py:1254-1266

numpy.**min**()

    Alias to: numpy.amin

- defined by <class 'abc.Numpy_redirect_min'>

numpy.**nancumprod**()

- defined by np_nancumprod(a) at numba/np/arraymath.py:1000-1022

numpy.**nancumsum**()

- defined by np_nancumsum(a) at numba/np/arraymath.py:1025-1047

numpy.**nanmax**()

- defined by np_nanmax(a) at numba/np/arraymath.py:891-897

numpy.**nanmean**()

- defined by np_nanmean(a) at numba/np/arraymath.py:900-917

numpy.**nanmedian**()

- defined by np_nanmedian(a) at numba/np/arraymath.py:1438-1460

numpy.**nanmin**()

- defined by np_nanmin(a) at numba/np/arraymath.py:882-888

numpy.**nanpercentile**()

- defined by np_nanpercentile(a, q) at numba/np/arraymath.py:1417-1421

numpy.**nanprod**()

- defined by np_nanprod(a) at numba/np/arraymath.py:978-997

numpy.**nanquantile**()

- defined by np_nanquantile(a, q) at numba/np/arraymath.py:1431-1435

numpy.**nanstd**()

- defined by np_nanstd(a) at numba/np/arraymath.py:945-953

numpy.**nansum**()

- defined by np_nansum(a) at numba/np/arraymath.py:956-975

numpy.**nanvar**()

- defined by np_nanvar(a) at numba/np/arraymath.py:920-942

numpy.**nonzero**()

- defined by <class 'abc.Numpy_redirect_nonzero'>

numpy.**ones**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.**ones_like**()

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.**outer**()

- defined by outer_impl(a, b, out=None) at numba/np/linalg.py:2645-2656

numpy.**partition**()

- defined by np_partition(a, kth) at numba/np/arraymath.py:1519-1541

numpy.**percentile**()

- defined by np_percentile(a, q) at numba/np/arraymath.py:1410-1414

numpy.**prod**()

- defined by <class 'abc.Numpy_redirect_prod'>

numpy.**ptp**()

- defined by np_ptp(a) at numba/np/arraymath.py:1109-1134

numpy.**quantile**()

- defined by np_quantile(a, q) at numba/np/arraymath.py:1424-1428

numpy.**ravel**()

- defined by <class 'abc.Numpy_redirect_ravel'>

numpy.**real**()

- defined by np_real(a) at numba/np/arraymath.py:3287-3292

numpy.**repeat**()

- defined by np_repeat(a, repeats) at numba/np/arrayobj.py:1925-1967

numpy.**reshape**()

- defined by np_reshape(a, shape) at numba/np/arrayobj.py:1702-1706

numpy.**roll**()

- defined by np_roll(a, shift) at numba/np/arraymath.py:1931-1951

numpy.**roots**()

- defined by roots_impl(p) at numba/np/polynomial.py:14-59

numpy.**round**()

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.**round_**()
    Alias to: numpy.round

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.**row_stack**()

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.**searchsorted**()

- defined by searchsorted(a, v, side='left') at numba/np/arraymath.py:3533-3565

numpy.**select**()

> • defined   by   np_select(condlist, choicelist, default=0)   at
>   numba/np/arraymath.py:4099-4147

numpy.**shape**()

> • defined by np_shape(a) at numba/np/arrayobj.py:1903-1910

numpy.**sinc**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.**sort**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.**stack**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.**std**()

> • defined by <class 'abc.Numpy_redirect_std'>

numpy.**sum**()

> • defined by <class 'abc.Numpy_redirect_sum'>

numpy.**take**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.**trace**()

> • defined by matrix_trace_impl(a, offset=0) at numba/np/linalg.py:2567-2595

numpy.**transpose**()

> • defined by numpy_transpose(a, axes=None) at numba/np/arrayobj.py:1530-1542

numpy.**trapz**()

> • defined by np_trapz(y, x=None, dx=1.0) at numba/np/arraymath.py:1833-1852

numpy.**tri**()

> • defined by np_tri(N, M=None, k=0) at numba/np/arraymath.py:1560-1571

numpy.**tril**()

> • defined by my_tril(m, k=0) at numba/np/arraymath.py:1597-1621

numpy.**tril_indices**()

> • defined by np_tril_indices(n, k=0, m=None) at numba/np/arraymath.py:1624-1635

numpy.**tril_indices_from**()

> • defined by np_tril_indices_from(arr, k=0) at numba/np/arraymath.py:1638-1649

numpy.**triu**()

> • defined by my_triu(m, k=0) at numba/np/arraymath.py:1658-1681

numpy.**triu_indices**()

• defined by `np_triu_indices(n, k=0, m=None)` at [numba/np/arraymath.py:1684-1695](#)

numpy.**triu_indices_from**()

• defined by `np_triu_indices_from(arr, k=0)` at [numba/np/arraymath.py:1698-1709](#)

numpy.**unique**()

• defined by `np_unique(a)` at [numba/np/arrayobj.py:1915-1922](#)

numpy.**vander**()

• defined by `np_vander(x, N=None, increasing=False)` at [numba/np/arraymath.py:1891-1928](#)

numpy.**var**()

• defined by `<class 'abc.Numpy_redirect_var'>`

numpy.**vdot**()

• defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**vstack**()
    Alias to: `numpy.row_stack`

• defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**where**()

• defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**zeros**()

• defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.**zeros_like**()

• defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 194 unsupported functions.

supported = 125 / 319 = 39.18%

numpy.compat

This module is not supported.

numpy.compat.py3k

This module is not supported.

numpy.compat.setup

This module is not supported.

numpy.core

numpy.core.**all**()
    Alias to: numpy.all

        • defined by np_all(a) at numba/np/arraymath.py:776-785

numpy.core.**amax**()
    Alias to: numpy.amax

        • defined by <class 'abc.Numpy_redirect_max'>

numpy.core.**amin**()
    Alias to: numpy.amin

        • defined by <class 'abc.Numpy_redirect_min'>

numpy.core.**any**()
    Alias to: numpy.any

        • defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.core.**arange**()
    Alias to: numpy.arange

        • defined   by   np_arange(start, stop=None, step=None, dtype=None)   at
          numba/np/arrayobj.py:3842-3908

numpy.core.**argmax**()
    Alias to: numpy.argmax

        • defined by <class 'abc.Numpy_redirect_argmax'>

numpy.core.**argmin**()
    Alias to: numpy.argmin

        • defined by <class 'abc.Numpy_redirect_argmin'>

numpy.core.**argsort**()
    Alias to: numpy.argsort

        • defined by <class 'abc.Numpy_redirect_argsort'>

numpy.core.**argwhere**()
    Alias to: numpy.argwhere

        • defined by np_argwhere(a) at numba/np/arraymath.py:2789-2821

numpy.core.**around**()
    Alias to: numpy.around

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.core.**array**()
    Alias to: numpy.array

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.core.**array_equal**()
    Alias to: numpy.array_equal

        • defined by np_array_equal(a, b) at numba/np/arraymath.py:3413-3432

numpy.core.**asarray**()
      Alias to: numpy.asarray

            • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.core.**ascontiguousarray**()
      Alias to: numpy.ascontiguousarray

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**asfortranarray**()
      Alias to: numpy.asfortranarray

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**atleast_1d**()
      Alias to: numpy.atleast_1d

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**atleast_2d**()
      Alias to: numpy.atleast_2d

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**atleast_3d**()
      Alias to: numpy.atleast_3d

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**concatenate**()
      Alias to: numpy.concatenate

            • defined by <class 'numba.core.typing.templates.Registry.register_global.
              <locals>.decorate.<locals>.Template'>

numpy.core.**convolve**()
      Alias to: numpy.convolve

            • defined by np_convolve(a, v) at numba/np/arraymath.py:3994-4015

numpy.core.**correlate**()
      Alias to: numpy.correlate

            • defined by _np_correlate(a, v) at numba/np/arraymath.py:3946-3991

numpy.core.**count_nonzero**()
      Alias to: numpy.count_nonzero

            • defined by np_count_nonzero(arr, axis=None) at numba/np/arraymath.py:3306-3320

numpy.core.**cross**()
      Alias to: numpy.cross

            • defined by np_cross(a, b) at numba/np/arraymath.py:4364-4386

numpy.core.**cumprod**()
      Alias to: numpy.cumprod

            • defined by <class 'abc.Numpy_redirect_cumprod'>

numpy.core.**cumsum**()
    Alias to: numpy.cumsum

    • defined by `<class 'abc.Numpy_redirect_cumsum'>`

numpy.core.**dot**()
    Alias to: numpy.dot

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**empty**()
    Alias to: numpy.empty

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**empty_like**()
    Alias to: numpy.empty_like

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**flatnonzero**()
    Alias to: numpy.flatnonzero

    • defined by `np_flatnonzero(a)` at numba/np/arraymath.py:2824-2839

numpy.core.**frombuffer**()
    Alias to: numpy.frombuffer

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**full**()
    Alias to: numpy.full

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**full_like**()
    Alias to: numpy.full_like

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**hstack**()
    Alias to: numpy.hstack

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**identity**()
    Alias to: numpy.identity

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**linspace**()
    Alias to: numpy.linspace

    • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
      `<locals>.decorate.<locals>.Template'>`

numpy.core.**max**()
    Alias to: numpy.amax

        • defined by <class 'abc.Numpy_redirect_max'>

numpy.core.**mean**()
    Alias to: numpy.mean

        • defined by <class 'abc.Numpy_redirect_mean'>

numpy.core.**min**()
    Alias to: numpy.amin

        • defined by <class 'abc.Numpy_redirect_min'>

numpy.core.**nonzero**()
    Alias to: numpy.nonzero

        • defined by <class 'abc.Numpy_redirect_nonzero'>

numpy.core.**ones**()
    Alias to: numpy.ones

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.core.**ones_like**()
    Alias to: numpy.ones_like

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.core.**outer**()
    Alias to: numpy.outer

        • defined by outer_impl(a, b, out=None) at numba/np/linalg.py:2645-2656

numpy.core.**partition**()
    Alias to: numpy.partition

        • defined by np_partition(a, kth) at numba/np/arraymath.py:1519-1541

numpy.core.**prod**()
    Alias to: numpy.prod

        • defined by <class 'abc.Numpy_redirect_prod'>

numpy.core.**ptp**()
    Alias to: numpy.ptp

        • defined by np_ptp(a) at numba/np/arraymath.py:1109-1134

numpy.core.**ravel**()
    Alias to: numpy.ravel

        • defined by <class 'abc.Numpy_redirect_ravel'>

numpy.core.**repeat**()
    Alias to: numpy.repeat

        • defined by np_repeat(a, repeats) at numba/np/arrayobj.py:1925-1967

numpy.core.**reshape**()
    Alias to: numpy.reshape

        • defined by np_reshape(a, shape) at numba/np/arrayobj.py:1702-1706

numpy.core.**roll**()
    Alias to: `numpy.roll`

        • defined by `np_roll(a, shift)` at numba/np/arraymath.py:1931-1951

numpy.core.**round**()
    Alias to: `numpy.round`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
         `<locals>.decorate.<locals>.Template'>`

numpy.core.**round_**()
    Alias to: `numpy.round`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
         `<locals>.decorate.<locals>.Template'>`

numpy.core.**searchsorted**()
    Alias to: `numpy.searchsorted`

        • defined by `searchsorted(a, v, side='left')` at numba/np/arraymath.py:3533-3565

numpy.core.**shape**()
    Alias to: `numpy.shape`

        • defined by `np_shape(a)` at numba/np/arrayobj.py:1903-1910

numpy.core.**sort**()
    Alias to: `numpy.sort`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
         `<locals>.decorate.<locals>.Template'>`

numpy.core.**stack**()
    Alias to: `numpy.stack`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
         `<locals>.decorate.<locals>.Template'>`

numpy.core.**std**()
    Alias to: `numpy.std`

        • defined by `<class 'abc.Numpy_redirect_std'>`

numpy.core.**sum**()
    Alias to: `numpy.sum`

        • defined by `<class 'abc.Numpy_redirect_sum'>`

numpy.core.**take**()
    Alias to: `numpy.take`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
         `<locals>.decorate.<locals>.Template'>`

numpy.core.**trace**()
    Alias to: `numpy.trace`

        • defined by `matrix_trace_impl(a, offset=0)` at numba/np/linalg.py:2567-2595

numpy.core.**transpose**()
    Alias to: `numpy.transpose`

        • defined by `numpy_transpose(a, axes=None)` at numba/np/arrayobj.py:1530-1542

numpy.core.**var**()
    Alias to: numpy.var

> • defined by <class 'abc.Numpy_redirect_var'>

numpy.core.**vdot**()
    Alias to: numpy.vdot

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.**vstack**()
    Alias to: numpy.row_stack

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.**where**()
    Alias to: numpy.where

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.**zeros**()
    Alias to: numpy.zeros

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.**zeros_like**()
    Alias to: numpy.zeros_like

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

Not showing 81 unsupported functions.

supported = 66 / 147 = 44.90%

numpy.core.arrayprint

numpy.core.arrayprint.**any**()
    Alias to: numpy.any

> • defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.core.arrayprint.**array**()
    Alias to: numpy.array

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.arrayprint.**asarray**()
    Alias to: numpy.asarray

> • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.core.arrayprint.**concatenate**()
    Alias to: numpy.concatenate

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

Not showing 21 unsupported functions.

supported = 4 / 25 = 16.00%

numpy.core.defchararray

numpy.core.defchararray.**narray**()
    Alias to: numpy.array

- defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

Not showing 54 unsupported functions.

supported = 1 / 55 = 1.82%

numpy.core.einsumfunc

This module is not supported.

numpy.core.fromnumeric

numpy.core.fromnumeric.**all**()
    Alias to: numpy.all

- defined by np_all(a) at numba/np/arraymath.py:776-785

numpy.core.fromnumeric.**amax**()
    Alias to: numpy.amax

- defined by <class 'abc.Numpy_redirect_max'>

numpy.core.fromnumeric.**amin**()
    Alias to: numpy.amin

- defined by <class 'abc.Numpy_redirect_min'>

numpy.core.fromnumeric.**any**()
    Alias to: numpy.any

- defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.core.fromnumeric.**argmax**()
    Alias to: numpy.argmax

- defined by <class 'abc.Numpy_redirect_argmax'>

numpy.core.fromnumeric.**argmin**()
    Alias to: numpy.argmin

- defined by <class 'abc.Numpy_redirect_argmin'>

numpy.core.fromnumeric.**argsort**()
    Alias to: numpy.argsort

- defined by <class 'abc.Numpy_redirect_argsort'>

numpy.core.fromnumeric.**around**()
    Alias to: numpy.around

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.core.fromnumeric.**array**()
    Alias to: numpy.array

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.core.fromnumeric.**asarray**()
    Alias to: numpy.asarray

- defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.core.fromnumeric.**concatenate**()
    Alias to: numpy.concatenate

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.core.fromnumeric.**cumprod**()
    Alias to: numpy.cumprod

- defined by <class 'abc.Numpy_redirect_cumprod'>

numpy.core.fromnumeric.**cumsum**()
    Alias to: numpy.cumsum

- defined by <class 'abc.Numpy_redirect_cumsum'>

numpy.core.fromnumeric.**mean**()
    Alias to: numpy.mean

- defined by <class 'abc.Numpy_redirect_mean'>

numpy.core.fromnumeric.**nonzero**()
    Alias to: numpy.nonzero

- defined by <class 'abc.Numpy_redirect_nonzero'>

numpy.core.fromnumeric.**partition**()
    Alias to: numpy.partition

- defined by np_partition(a, kth) at numba/np/arraymath.py:1519-1541

numpy.core.fromnumeric.**prod**()
    Alias to: numpy.prod

- defined by <class 'abc.Numpy_redirect_prod'>

numpy.core.fromnumeric.**ptp**()
    Alias to: numpy.ptp

- defined by np_ptp(a) at numba/np/arraymath.py:1109-1134

numpy.core.fromnumeric.**ravel**()
    Alias to: numpy.ravel

- defined by <class 'abc.Numpy_redirect_ravel'>

numpy.core.fromnumeric.**repeat**()
    Alias to: numpy.repeat

- defined by np_repeat(a, repeats) at numba/np/arrayobj.py:1925-1967

`numpy.core.fromnumeric.`**`reshape`**`()`
>   Alias to: `numpy.reshape`

>   • defined by `np_reshape(a, shape)` at numba/np/arrayobj.py:1702-1706

`numpy.core.fromnumeric.`**`round_`**`()`
>   Alias to: `numpy.round`

>   • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

`numpy.core.fromnumeric.`**`searchsorted`**`()`
>   Alias to: `numpy.searchsorted`

>   • defined by `searchsorted(a, v, side='left')` at numba/np/arraymath.py:3533-3565

`numpy.core.fromnumeric.`**`shape`**`()`
>   Alias to: `numpy.shape`

>   • defined by `np_shape(a)` at numba/np/arrayobj.py:1903-1910

`numpy.core.fromnumeric.`**`sort`**`()`
>   Alias to: `numpy.sort`

>   • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

`numpy.core.fromnumeric.`**`std`**`()`
>   Alias to: `numpy.std`

>   • defined by `<class 'abc.Numpy_redirect_std'>`

`numpy.core.fromnumeric.`**`sum`**`()`
>   Alias to: `numpy.sum`

>   • defined by `<class 'abc.Numpy_redirect_sum'>`

`numpy.core.fromnumeric.`**`take`**`()`
>   Alias to: `numpy.take`

>   • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

`numpy.core.fromnumeric.`**`trace`**`()`
>   Alias to: `numpy.trace`

>   • defined by `matrix_trace_impl(a, offset=0)` at numba/np/linalg.py:2567-2595

`numpy.core.fromnumeric.`**`transpose`**`()`
>   Alias to: `numpy.transpose`

>   • defined by `numpy_transpose(a, axes=None)` at numba/np/arrayobj.py:1530-1542

`numpy.core.fromnumeric.`**`var`**`()`
>   Alias to: `numpy.var`

>   • defined by `<class 'abc.Numpy_redirect_var'>`

Not showing 17 unsupported functions.

supported = 31 / 48 = 64.58%

numpy.core.function_base

numpy.core.function_base.**linspace**()
> Alias to: `numpy.linspace`
>
> > • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
> > `<locals>.decorate.<locals>.Template'>`

Not showing 7 unsupported functions.

supported = 1 / 8 = 12.50%

numpy.core.getlimits

numpy.core.getlimits.**array**()
> Alias to: `numpy.array`
>
> > • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
> > `<locals>.decorate.<locals>.Template'>`

Not showing 1 unsupported functions.

supported = 1 / 2 = 50.00%

numpy.core.machar

numpy.core.machar.**any**()
> Alias to: `numpy.any`
>
> > • defined by `np_any(a)` at numba/np/arraymath.py:788-797

Not showing 1 unsupported functions.

supported = 1 / 2 = 50.00%

numpy.core.multiarray

numpy.core.multiarray.**arange**()
> Alias to: `numpy.arange`
>
> > • defined by `np_arange(start, stop=None, step=None, dtype=None)` at
> > numba/np/arrayobj.py:3842-3908

numpy.core.multiarray.**array**()
> Alias to: `numpy.array`
>
> > • defined by `<class 'numba.core.typing.templates.Registry.register_global.`
> > `<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**bincount**()
> Alias to: `numpy.bincount`
>
> > • defined by `np_bincount(a, weights=None, minlength=0)` at numba/np/arraymath.py:3445-
> > 3500

numpy.core.multiarray.**concatenate**()
> Alias to: `numpy.concatenate`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**dot**()
    Alias to: `numpy.dot`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**empty**()
    Alias to: `numpy.empty`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**empty_like**()
    Alias to: `numpy.empty_like`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**frombuffer**()
    Alias to: `numpy.frombuffer`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**vdot**()
    Alias to: `numpy.vdot`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**where**()
    Alias to: `numpy.where`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.multiarray.**zeros**()
    Alias to: `numpy.zeros`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 46 unsupported functions.

supported = 11 / 57 = 19.30%

numpy.core.numeric

numpy.core.numeric.**all**()
    Alias to: `numpy.all`

- defined by `np_all(a)` at numba/np/arraymath.py:776-785

numpy.core.numeric.**amax**()
    Alias to: `numpy.amax`

- defined by `<class 'abc.Numpy_redirect_max'>`

numpy.core.numeric.**amin**()
    Alias to: `numpy.amin`

- defined by `<class 'abc.Numpy_redirect_min'>`

`numpy.core.numeric.`**`any`**`()`

    Alias to: `numpy.any`

- defined by `np_any(a)` at numba/np/arraymath.py:788-797

`numpy.core.numeric.`**`arange`**`()`

    Alias to: `numpy.arange`

- defined   by   `np_arange(start, stop=None, step=None, dtype=None)`   at numba/np/arrayobj.py:3842-3908

`numpy.core.numeric.`**`argmax`**`()`

    Alias to: `numpy.argmax`

- defined by `<class 'abc.Numpy_redirect_argmax'>`

`numpy.core.numeric.`**`argmin`**`()`

    Alias to: `numpy.argmin`

- defined by `<class 'abc.Numpy_redirect_argmin'>`

`numpy.core.numeric.`**`argsort`**`()`

    Alias to: `numpy.argsort`

- defined by `<class 'abc.Numpy_redirect_argsort'>`

`numpy.core.numeric.`**`argwhere`**`()`

    Alias to: `numpy.argwhere`

- defined by `np_argwhere(a)` at numba/np/arraymath.py:2789-2821

`numpy.core.numeric.`**`around`**`()`

    Alias to: `numpy.around`

- defined by `<class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>`

`numpy.core.numeric.`**`array`**`()`

    Alias to: `numpy.array`

- defined by `<class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>`

`numpy.core.numeric.`**`array_equal`**`()`

    Alias to: `numpy.array_equal`

- defined by `np_array_equal(a, b)` at numba/np/arraymath.py:3413-3432

`numpy.core.numeric.`**`asarray`**`()`

    Alias to: `numpy.asarray`

- defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

`numpy.core.numeric.`**`ascontiguousarray`**`()`

    Alias to: `numpy.ascontiguousarray`

- defined by `<class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>`

`numpy.core.numeric.`**`asfortranarray`**`()`

    Alias to: `numpy.asfortranarray`

- defined by `<class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>`

numpy.core.numeric.**concatenate**()
    Alias to: numpy.concatenate

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**convolve**()
    Alias to: numpy.convolve

> • defined by np_convolve(a, v) at numba/np/arraymath.py:3994-4015

numpy.core.numeric.**correlate**()
    Alias to: numpy.correlate

> • defined by _np_correlate(a, v) at numba/np/arraymath.py:3946-3991

numpy.core.numeric.**count_nonzero**()
    Alias to: numpy.count_nonzero

> • defined by np_count_nonzero(arr, axis=None) at numba/np/arraymath.py:3306-3320

numpy.core.numeric.**cross**()
    Alias to: numpy.cross

> • defined by np_cross(a, b) at numba/np/arraymath.py:4364-4386

numpy.core.numeric.**cumprod**()
    Alias to: numpy.cumprod

> • defined by <class 'abc.Numpy_redirect_cumprod'>

numpy.core.numeric.**cumsum**()
    Alias to: numpy.cumsum

> • defined by <class 'abc.Numpy_redirect_cumsum'>

numpy.core.numeric.**dot**()
    Alias to: numpy.dot

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**empty**()
    Alias to: numpy.empty

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**empty_like**()
    Alias to: numpy.empty_like

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**flatnonzero**()
    Alias to: numpy.flatnonzero

> • defined by np_flatnonzero(a) at numba/np/arraymath.py:2824-2839

numpy.core.numeric.**frombuffer**()
    Alias to: numpy.frombuffer

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**full**()
    Alias to: numpy.full

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**full_like**()
    Alias to: numpy.full_like

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**identity**()
    Alias to: numpy.identity

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**mean**()
    Alias to: numpy.mean

    • defined by <class 'abc.Numpy_redirect_mean'>

numpy.core.numeric.**nonzero**()
    Alias to: numpy.nonzero

    • defined by <class 'abc.Numpy_redirect_nonzero'>

numpy.core.numeric.**ones**()
    Alias to: numpy.ones

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**ones_like**()
    Alias to: numpy.ones_like

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.core.numeric.**outer**()
    Alias to: numpy.outer

    • defined by outer_impl(a, b, out=None) at numba/np/linalg.py:2645-2656

numpy.core.numeric.**partition**()
    Alias to: numpy.partition

    • defined by np_partition(a, kth) at numba/np/arraymath.py:1519-1541

numpy.core.numeric.**prod**()
    Alias to: numpy.prod

    • defined by <class 'abc.Numpy_redirect_prod'>

numpy.core.numeric.**ptp**()
    Alias to: numpy.ptp

    • defined by np_ptp(a) at numba/np/arraymath.py:1109-1134

numpy.core.numeric.**ravel**()
    Alias to: numpy.ravel

    • defined by <class 'abc.Numpy_redirect_ravel'>

numpy.core.numeric.**repeat**()
    Alias to: numpy.repeat

       • defined by np_repeat(a, repeats) at numba/np/arrayobj.py:1925-1967

numpy.core.numeric.**reshape**()
    Alias to: numpy.reshape

       • defined by np_reshape(a, shape) at numba/np/arrayobj.py:1702-1706

numpy.core.numeric.**roll**()
    Alias to: numpy.roll

       • defined by np_roll(a, shift) at numba/np/arraymath.py:1931-1951

numpy.core.numeric.**round_**()
    Alias to: numpy.round

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.core.numeric.**searchsorted**()
    Alias to: numpy.searchsorted

       • defined by searchsorted(a, v, side='left') at numba/np/arraymath.py:3533-3565

numpy.core.numeric.**shape**()
    Alias to: numpy.shape

       • defined by np_shape(a) at numba/np/arrayobj.py:1903-1910

numpy.core.numeric.**sort**()
    Alias to: numpy.sort

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.core.numeric.**std**()
    Alias to: numpy.std

       • defined by <class 'abc.Numpy_redirect_std'>

numpy.core.numeric.**sum**()
    Alias to: numpy.sum

       • defined by <class 'abc.Numpy_redirect_sum'>

numpy.core.numeric.**take**()
    Alias to: numpy.take

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.core.numeric.**trace**()
    Alias to: numpy.trace

       • defined by matrix_trace_impl(a, offset=0) at numba/np/linalg.py:2567-2595

numpy.core.numeric.**transpose**()
    Alias to: numpy.transpose

       • defined by numpy_transpose(a, axes=None) at numba/np/arrayobj.py:1530-1542

numpy.core.numeric.**var**()
    Alias to: numpy.var

- defined by `<class 'abc.Numpy_redirect_var'>`

numpy.core.numeric.**vdot**()
    Alias to: `numpy.vdot`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.numeric.**where**()
    Alias to: `numpy.where`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.numeric.**zeros**()
    Alias to: `numpy.zeros`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.numeric.**zeros_like**()
    Alias to: `numpy.zeros_like`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 80 unsupported functions.

supported = 56 / 136 = 41.18%

numpy.core.numerictypes

numpy.core.numerictypes.**array**()
    Alias to: `numpy.array`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.core.numerictypes.**empty**()
    Alias to: `numpy.empty`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 18 unsupported functions.

supported = 2 / 20 = 10.00%

numpy.core.overrides

This module is not supported.

numpy.core.records

This module is not supported.

numpy.core.setup_common

This module is not supported.

numpy.core.shape_base

numpy.core.shape_base.**array**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**atleast_1d**()
    Alias to: numpy.atleast_1d

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**atleast_2d**()
    Alias to: numpy.atleast_2d

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**atleast_3d**()
    Alias to: numpy.atleast_3d

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**hstack**()
    Alias to: numpy.hstack

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**stack**()
    Alias to: numpy.stack

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.shape_base.**vstack**()
    Alias to: numpy.row_stack

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 3 unsupported functions.

supported = 7 / 10 = 70.00%

numpy.core.umath

numpy.core.umath.**arange**()
    Alias to: numpy.arange

> • defined by np_arange(start, stop=None, step=None, dtype=None) at
> numba/np/arrayobj.py:3842-3908

---

numpy.core.umath.**array**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.umath.**empty**()
    Alias to: numpy.empty

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.umath.**frombuffer**()
    Alias to: numpy.frombuffer

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.core.umath.**zeros**()
    Alias to: numpy.zeros

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 52 unsupported functions.

supported = 5 / 57 = 8.77%

numpy.core.umath_tests

This module is not supported.

numpy.ctypeslib

numpy.ctypeslib.**array**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.ctypeslib.**frombuffer**()
    Alias to: numpy.frombuffer

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 7 unsupported functions.

supported = 2 / 9 = 22.22%

numpy.distutils

This module is not supported.

numpy.distutils.ccompiler

This module is not supported.

---

numpy.distutils.command

This module is not supported.

numpy.distutils.command.autodist

This module is not supported.

numpy.distutils.command.bdist_rpm

This module is not supported.

numpy.distutils.command.build

This module is not supported.

numpy.distutils.command.build_clib

This module is not supported.

numpy.distutils.command.build_ext

This module is not supported.

numpy.distutils.command.build_py

This module is not supported.

numpy.distutils.command.build_scripts

This module is not supported.

numpy.distutils.command.build_src

This module is not supported.

numpy.distutils.command.config

This module is not supported.

numpy.distutils.command.config_compiler

This module is not supported.

numpy.distutils.command.develop

This module is not supported.

numpy.distutils.command.egg_info

This module is not supported.

numpy.distutils.command.install

This module is not supported.

numpy.distutils.command.install_clib

This module is not supported.

numpy.distutils.command.install_data

This module is not supported.

numpy.distutils.command.install_headers

This module is not supported.

numpy.distutils.command.sdist

This module is not supported.

numpy.distutils.conv_template

This module is not supported.

numpy.distutils.core

This module is not supported.

numpy.distutils.cpuinfo

This module is not supported.

numpy.distutils.exec_command

This module is not supported.

numpy.distutils.extension

This module is not supported.

numpy.distutils.fcompiler

This module is not supported.

numpy.distutils.fcompiler.absoft

This module is not supported.

numpy.distutils.fcompiler.compaq

This module is not supported.

numpy.distutils.fcompiler.environment

This module is not supported.

numpy.distutils.fcompiler.g95

This module is not supported.

numpy.distutils.fcompiler.gnu

This module is not supported.

numpy.distutils.fcompiler.hpux

This module is not supported.

numpy.distutils.fcompiler.ibm

This module is not supported.

numpy.distutils.fcompiler.intel

This module is not supported.

numpy.distutils.fcompiler.lahey

This module is not supported.

numpy.distutils.fcompiler.mips

This module is not supported.

numpy.distutils.fcompiler.nag

This module is not supported.

numpy.distutils.fcompiler.none

This module is not supported.

numpy.distutils.fcompiler.pathf95

This module is not supported.

numpy.distutils.fcompiler.pg

This module is not supported.

numpy.distutils.fcompiler.sun

This module is not supported.

numpy.distutils.fcompiler.vast

This module is not supported.

numpy.distutils.from_template

This module is not supported.

numpy.distutils.intelccompiler

This module is not supported.

numpy.distutils.lib2def

This module is not supported.

numpy.distutils.line_endings

This module is not supported.

numpy.distutils.log

This module is not supported.

numpy.distutils.mingw32ccompiler

This module is not supported.

numpy.distutils.misc_util

This module is not supported.

numpy.distutils.msvccompiler

This module is not supported.

numpy.distutils.npy_pkg_config

This module is not supported.

numpy.distutils.numpy_distribution

This module is not supported.

numpy.distutils.pathccompiler

This module is not supported.

numpy.distutils.setup

This module is not supported.

numpy.distutils.system_info

numpy.distutils.system_info.**reduce**()

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 16 unsupported functions.

supported = 1 / 17 = 5.88%

numpy.distutils.unixccompiler

This module is not supported.

numpy.doc

This module is not supported.

numpy.doc.basics

This module is not supported.

numpy.doc.broadcasting

This module is not supported.

numpy.doc.byteswapping

This module is not supported.

numpy.doc.constants

This module is not supported.

numpy.doc.creation

This module is not supported.

numpy.doc.dispatch

This module is not supported.

numpy.doc.glossary

This module is not supported.

numpy.doc.indexing

This module is not supported.

numpy.doc.internals

This module is not supported.

numpy.doc.misc

This module is not supported.

numpy.doc.structured_arrays

This module is not supported.

numpy.doc.subclassing

This module is not supported.

numpy.doc.ufuncs

This module is not supported.

numpy.dual

numpy.dual.**cholesky**()

> • defined by cho_impl(a) at numba/np/linalg.py:901-943

numpy.dual.**det**()

> • defined by det_impl(a) at numba/np/linalg.py:2018-2029

numpy.dual.**eig**()

> • defined by eig_impl(a) at numba/np/linalg.py:945-1058

numpy.dual.**eigh**()

> • defined by eigh_impl(a) at numba/np/linalg.py:1179-1226

numpy.dual.**eigvals**()

> • defined by eigvals_impl(a) at numba/np/linalg.py:1060-1177

numpy.dual.**eigvalsh**()

> • defined by eigvalsh_impl(a) at numba/np/linalg.py:1228-1275

numpy.dual.**inv**()

> • defined by inv_impl(a) at numba/np/linalg.py:833-870

numpy.dual.**lstsq**()

> • defined by lstsq_impl(a, b, rcond=-1.0) at numba/np/linalg.py:1583-1675

numpy.dual.**norm**()

> • defined by norm_impl(a, ord=None) at numba/np/linalg.py:2352-2358

numpy.dual.**pinv**()

> • defined by pinv_impl(a, rcond=1e-15) at numba/np/linalg.py:1759-1920

numpy.dual.**solve**()

> • defined by solve_impl(a, b) at numba/np/linalg.py:1698-1756

numpy.dual.**svd**()

> • defined by svd_impl(a, full_matrices=1) at numba/np/linalg.py:1277-1340

Not showing 10 unsupported functions.

supported = 12 / 22 = 54.55%

numpy.f2py

This module is not supported.

numpy.f2py.auxfuncs

numpy.f2py.auxfuncs.**reduce**()
    Alias to: `numpy.distutils.system_info.reduce`

> • defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 102 unsupported functions.

supported = 1 / 103 = 0.97%

numpy.f2py.capi_maps

This module is not supported.

numpy.f2py.cb_rules

This module is not supported.

numpy.f2py.cfuncs

This module is not supported.

numpy.f2py.common_rules

This module is not supported.

numpy.f2py.crackfortran

This module is not supported.

numpy.f2py.diagnose

This module is not supported.

numpy.f2py.f2py2e

This module is not supported.

numpy.f2py.f2py_testing

This module is not supported.

numpy.f2py.f90mod_rules

This module is not supported.

numpy.f2py.func2subr

This module is not supported.

numpy.f2py.rules

This module is not supported.

numpy.f2py.use_rules

This module is not supported.

numpy.fft

This module is not supported.

numpy.fft.helper

numpy.fft.helper.**arange**()
> Alias to: `numpy.arange`

> > • defined by `np_arange(start, stop=None, step=None, dtype=None)` at numba/np/arrayobj.py:3842-3908

numpy.fft.helper.**asarray**()
> Alias to: `numpy.asarray`

> > • defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

numpy.fft.helper.**empty**()
> Alias to: `numpy.empty`

> > • defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.fft.helper.**roll**()
> Alias to: `numpy.roll`

> > • defined by `np_roll(a, shift)` at numba/np/arraymath.py:1931-1951

Not showing 6 unsupported functions.

supported = 4 / 10 = 40.00%

numpy.fft.setup

This module is not supported.

numpy.lib

numpy.lib.**angle**()
     Alias to: numpy.angle

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.**append**()
     Alias to: numpy.append

          • defined by np_append(arr, values, axis=None) at numba/np/arrayobj.py:1709-1732

numpy.lib.**bartlett**()
     Alias to: numpy.bartlett

          • defined   by   window_generator.<locals>.window_overload(M)   at
            numba/np/arraymath.py:4183-4195

numpy.lib.**bincount**()
     Alias to: numpy.bincount

          • defined by np_bincount(a, weights=None, minlength=0) at [numba/np/arraymath.py:3445-](numba/np/arraymath.py:3445-3500)
            [3500](numba/np/arraymath.py:3445-3500)

numpy.lib.**blackman**()
     Alias to: numpy.blackman

          • defined   by   window_generator.<locals>.window_overload(M)   at
            numba/np/arraymath.py:4183-4195

numpy.lib.**column_stack**()
     Alias to: numpy.column_stack

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.**copy**()
     Alias to: numpy.copy

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.**corrcoef**()
     Alias to: numpy.corrcoef

          • defined by np_corrcoef(x, y=None, rowvar=True) at numba/np/arraymath.py:2752-2782

numpy.lib.**cov**()
     Alias to: numpy.cov

          • defined   by   np_cov(m, y=None, rowvar=True, bias=False, ddof=None)   at
            numba/np/arraymath.py:2693-2749

numpy.lib.**delete**()
     Alias to: numpy.delete

          • defined by np_delete(arr, obj) at numba/np/arraymath.py:3327-3371

numpy.lib.**diag**()
    Alias to: numpy.diag

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.**diff**()
    Alias to: numpy.diff

- defined by np_diff_impl(a, n=1) at [numba/np/arraymath.py:3374-3410](#)

numpy.lib.**digitize**()
    Alias to: numpy.digitize

- defined by np_digitize(x, bins, right=False) at [numba/np/arraymath.py:3568-3694](#)

numpy.lib.**dstack**()
    Alias to: numpy.dstack

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.**ediff1d**()
    Alias to: numpy.ediff1d

- defined by np_ediff1d(ary, to_end=None, to_begin=None) at
  [numba/np/arraymath.py:1741-1796](#)

numpy.lib.**expand_dims**()
    Alias to: numpy.expand_dims

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.**extract**()
    Alias to: numpy.extract

- defined by np_extract(condition, arr) at [numba/np/arraymath.py:4073-4096](#)

numpy.lib.**eye**()
    Alias to: numpy.eye

- defined by numpy_eye(N, M=None, k=0, dtype=<class 'float'>) at
  [numba/np/arrayobj.py:3681-3704](#)

numpy.lib.**fill_diagonal**()
    Alias to: numpy.fill_diagonal

- defined by np_fill_diagonal(a, val, wrap=False) at [numba/np/arraymath.py:2926-2956](#)

numpy.lib.**flip**()
    Alias to: numpy.flip

- defined by np_flip(a) at [numba/np/arrayobj.py:4953-4964](#)

numpy.lib.**fliplr**()
    Alias to: numpy.fliplr

- defined by np_flip_lr(a) at [numba/np/arrayobj.py:4891-4905](#)

numpy.lib.**flipud**()
    Alias to: numpy.flipud

- defined by np_flip_ud(a) at [numba/np/arrayobj.py:4908-4922](#)

numpy.lib.**hamming**()
    Alias to: numpy.hamming

  - defined   by   window_generator.<locals>.window_overload(M)   at
    numba/np/arraymath.py:4183-4195

numpy.lib.**hanning**()
    Alias to: numpy.hanning

  - defined   by   window_generator.<locals>.window_overload(M)   at
    numba/np/arraymath.py:4183-4195

numpy.lib.**histogram**()
    Alias to: numpy.histogram

  - defined by np_histogram(a, bins=10, range=None) at numba/np/arraymath.py:3700-3780

numpy.lib.**imag**()
    Alias to: numpy.imag

  - defined by np_imag(a) at numba/np/arraymath.py:3295-3300

numpy.lib.**interp**()
    Alias to: numpy.interp

  - defined by np_interp(x, xp, fp) at numba/np/arraymath.py:2444-2505

numpy.lib.**kaiser**()
    Alias to: numpy.kaiser

  - defined by np_kaiser(M, beta) at numba/np/arraymath.py:4301-4320

numpy.lib.**kron**()
    Alias to: numpy.kron

  - defined by kron_impl(a, b) at numba/np/linalg.py:2722-2776

numpy.lib.**median**()
    Alias to: numpy.median

  - defined by np_median(a) at numba/np/arraymath.py:1254-1266

numpy.lib.**nancumprod**()
    Alias to: numpy.nancumprod

  - defined by np_nancumprod(a) at numba/np/arraymath.py:1000-1022

numpy.lib.**nancumsum**()
    Alias to: numpy.nancumsum

  - defined by np_nancumsum(a) at numba/np/arraymath.py:1025-1047

numpy.lib.**nanmax**()
    Alias to: numpy.nanmax

  - defined by np_nanmax(a) at numba/np/arraymath.py:891-897

numpy.lib.**nanmean**()
    Alias to: numpy.nanmean

  - defined by np_nanmean(a) at numba/np/arraymath.py:900-917

numpy.lib.**nanmedian**()
    Alias to: numpy.nanmedian

  - defined by np_nanmedian(a) at numba/np/arraymath.py:1438-1460

numpy.lib.**nanmin**()
    Alias to: numpy.nanmin

    • defined by np_nanmin(a) at numba/np/arraymath.py:882-888

numpy.lib.**nanpercentile**()
    Alias to: numpy.nanpercentile

    • defined by np_nanpercentile(a, q) at numba/np/arraymath.py:1417-1421

numpy.lib.**nanprod**()
    Alias to: numpy.nanprod

    • defined by np_nanprod(a) at numba/np/arraymath.py:978-997

numpy.lib.**nanquantile**()
    Alias to: numpy.nanquantile

    • defined by np_nanquantile(a, q) at numba/np/arraymath.py:1431-1435

numpy.lib.**nanstd**()
    Alias to: numpy.nanstd

    • defined by np_nanstd(a) at numba/np/arraymath.py:945-953

numpy.lib.**nansum**()
    Alias to: numpy.nansum

    • defined by np_nansum(a) at numba/np/arraymath.py:956-975

numpy.lib.**nanvar**()
    Alias to: numpy.nanvar

    • defined by np_nanvar(a) at numba/np/arraymath.py:920-942

numpy.lib.**percentile**()
    Alias to: numpy.percentile

    • defined by np_percentile(a, q) at numba/np/arraymath.py:1410-1414

numpy.lib.**quantile**()
    Alias to: numpy.quantile

    • defined by np_quantile(a, q) at numba/np/arraymath.py:1424-1428

numpy.lib.**real**()
    Alias to: numpy.real

    • defined by np_real(a) at numba/np/arraymath.py:3287-3292

numpy.lib.**roots**()
    Alias to: numpy.roots

    • defined by roots_impl(p) at numba/np/polynomial.py:14-59

numpy.lib.**row_stack**()
    Alias to: numpy.row_stack

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.**select**()
    Alias to: numpy.select

    • defined by np_select(condlist, choicelist, default=0) at
      numba/np/arraymath.py:4099-4147

---

numpy.lib.**sinc**()
    Alias to: numpy.sinc

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.lib.**trapz**()
    Alias to: numpy.trapz

        • defined by np_trapz(y, x=None, dx=1.0) at [numba/np/arraymath.py:1833-1852](#)

numpy.lib.**tri**()
    Alias to: numpy.tri

        • defined by np_tri(N, M=None, k=0) at [numba/np/arraymath.py:1560-1571](#)

numpy.lib.**tril**()
    Alias to: numpy.tril

        • defined by my_tril(m, k=0) at [numba/np/arraymath.py:1597-1621](#)

numpy.lib.**tril_indices**()
    Alias to: numpy.tril_indices

        • defined by np_tril_indices(n, k=0, m=None) at [numba/np/arraymath.py:1624-1635](#)

numpy.lib.**tril_indices_from**()
    Alias to: numpy.tril_indices_from

        • defined by np_tril_indices_from(arr, k=0) at [numba/np/arraymath.py:1638-1649](#)

numpy.lib.**triu**()
    Alias to: numpy.triu

        • defined by my_triu(m, k=0) at [numba/np/arraymath.py:1658-1681](#)

numpy.lib.**triu_indices**()
    Alias to: numpy.triu_indices

        • defined by np_triu_indices(n, k=0, m=None) at [numba/np/arraymath.py:1684-1695](#)

numpy.lib.**triu_indices_from**()
    Alias to: numpy.triu_indices_from

        • defined by np_triu_indices_from(arr, k=0) at [numba/np/arraymath.py:1698-1709](#)

numpy.lib.**unique**()
    Alias to: numpy.unique

        • defined by np_unique(a) at [numba/np/arrayobj.py:1915-1922](#)

numpy.lib.**vander**()
    Alias to: numpy.vander

        • defined by np_vander(x, N=None, increasing=False) at [numba/np/arraymath.py:1891-
          1928](#)

Not showing 110 unsupported functions.

supported = 59 / 169 = 34.91%

numpy.lib.arraypad

This module is not supported.

---

numpy.lib.arraysetops

numpy.lib.arraysetops.**ediff1d**()
    Alias to: numpy.ediff1d

        • defined   by   np_ediff1d(ary, to_end=None, to_begin=None)   at
        numba/np/arraymath.py:1741-1796

numpy.lib.arraysetops.**unique**()
    Alias to: numpy.unique

        • defined by np_unique(a) at numba/np/arrayobj.py:1915-1922

Not showing 6 unsupported functions.

supported = 2 / 8 = 25.00%

numpy.lib.arrayterator

numpy.lib.arrayterator.**mul**()

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
        <locals>.decorate.<locals>.Template'>

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
        <locals>.decorate.<locals>.Template'>

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
        <locals>.decorate.<locals>.Template'>

        • defined by unicode_repeat(a, b) at numba/cpython/unicode.py:1716-1725

        • defined by charseq_repeat(a, b) at numba/cpython/charseq.py:399-416

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
        <locals>.decorate.<locals>.Template'>

numpy.lib.arrayterator.**reduce**()
    Alias to: numpy.distutils.system_info.reduce

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
        <locals>.decorate.<locals>.Template'>

Not showing 0 unsupported functions.

supported = 2 / 2 = 100.00%

numpy.lib.financial

This module is not supported.

numpy.lib.format

This module is not supported.

numpy.lib.function_base

numpy.lib.function_base.**angle**()
     Alias to: numpy.angle

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**any**()
     Alias to: numpy.any

          • defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.lib.function_base.**append**()
     Alias to: numpy.append

          • defined by np_append(arr, values, axis=None) at numba/np/arrayobj.py:1709-1732

numpy.lib.function_base.**arange**()
     Alias to: numpy.arange

          • defined   by   np_arange(start, stop=None, step=None, dtype=None)   at
            numba/np/arrayobj.py:3842-3908

numpy.lib.function_base.**around**()
     Alias to: numpy.around

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**array**()
     Alias to: numpy.array

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**asarray**()
     Alias to: numpy.asarray

          • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.lib.function_base.**bartlett**()
     Alias to: numpy.bartlett

          • defined   by   window_generator.<locals>.window_overload(M)   at
            numba/np/arraymath.py:4183-4195

numpy.lib.function_base.**bincount**()
     Alias to: numpy.bincount

          • defined by np_bincount(a, weights=None, minlength=0) at numba/np/arraymath.py:3445-
            3500

numpy.lib.function_base.**blackman**()
     Alias to: numpy.blackman

          • defined   by   window_generator.<locals>.window_overload(M)   at
            numba/np/arraymath.py:4183-4195

numpy.lib.function_base.**concatenate**()
     Alias to: numpy.concatenate

          • defined by <class 'numba.core.typing.templates.Registry.register_global.
            <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**copy**()
    Alias to: numpy.copy

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**corrcoef**()
    Alias to: numpy.corrcoef

- defined by np_corrcoef(x, y=None, rowvar=True) at numba/np/arraymath.py:2752-2782

numpy.lib.function_base.**cov**()
    Alias to: numpy.cov

- defined by np_cov(m, y=None, rowvar=True, bias=False, ddof=None) at numba/np/arraymath.py:2693-2749

numpy.lib.function_base.**delete**()
    Alias to: numpy.delete

- defined by np_delete(arr, obj) at numba/np/arraymath.py:3327-3371

numpy.lib.function_base.**diag**()
    Alias to: numpy.diag

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**diff**()
    Alias to: numpy.diff

- defined by np_diff_impl(a, n=1) at numba/np/arraymath.py:3374-3410

numpy.lib.function_base.**digitize**()
    Alias to: numpy.digitize

- defined by np_digitize(x, bins, right=False) at numba/np/arraymath.py:3568-3694

numpy.lib.function_base.**dot**()
    Alias to: numpy.dot

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**empty**()
    Alias to: numpy.empty

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**extract**()
    Alias to: numpy.extract

- defined by np_extract(condition, arr) at numba/np/arraymath.py:4073-4096

numpy.lib.function_base.**flip**()
    Alias to: numpy.flip

- defined by np_flip(a) at numba/np/arrayobj.py:4953-4964

numpy.lib.function_base.**hamming**()
    Alias to: numpy.hamming

- defined by window_generator.<locals>.window_overload(M) at numba/np/arraymath.py:4183-4195

`numpy.lib.function_base.`**`hanning`**`()`
    Alias to: `numpy.hanning`

        • defined by `window_generator.<locals>.window_overload(M)` at [numba/np/arraymath.py:4183-4195](#)

`numpy.lib.function_base.`**`histogram`**`()`
    Alias to: `numpy.histogram`

        • defined by `np_histogram(a, bins=10, range=None)` at [numba/np/arraymath.py:3700-3780](#)

`numpy.lib.function_base.`**`interp`**`()`
    Alias to: `numpy.interp`

        • defined by `np_interp(x, xp, fp)` at [numba/np/arraymath.py:2444-2505](#)

`numpy.lib.function_base.`**`kaiser`**`()`
    Alias to: `numpy.kaiser`

        • defined by `np_kaiser(M, beta)` at [numba/np/arraymath.py:4301-4320](#)

`numpy.lib.function_base.`**`mean`**`()`
    Alias to: `numpy.mean`

        • defined by `<class 'abc.Numpy_redirect_mean'>`

`numpy.lib.function_base.`**`median`**`()`
    Alias to: `numpy.median`

        • defined by `np_median(a)` at [numba/np/arraymath.py:1254-1266](#)

`numpy.lib.function_base.`**`nonzero`**`()`
    Alias to: `numpy.nonzero`

        • defined by `<class 'abc.Numpy_redirect_nonzero'>`

`numpy.lib.function_base.`**`ones`**`()`
    Alias to: `numpy.ones`

        • defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.lib.function_base.`**`partition`**`()`
    Alias to: `numpy.partition`

        • defined by `np_partition(a, kth)` at [numba/np/arraymath.py:1519-1541](#)

`numpy.lib.function_base.`**`percentile`**`()`
    Alias to: `numpy.percentile`

        • defined by `np_percentile(a, q)` at [numba/np/arraymath.py:1410-1414](#)

`numpy.lib.function_base.`**`quantile`**`()`
    Alias to: `numpy.quantile`

        • defined by `np_quantile(a, q)` at [numba/np/arraymath.py:1424-1428](#)

`numpy.lib.function_base.`**`ravel`**`()`
    Alias to: `numpy.ravel`

        • defined by `<class 'abc.Numpy_redirect_ravel'>`

`numpy.lib.function_base.`**`select`**`()`
    Alias to: `numpy.select`

- defined by np_select(condlist, choicelist, default=0) at numba/np/arraymath.py:4099-4147

numpy.lib.function_base.**sinc**()
    Alias to: numpy.sinc

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**sum**()
    Alias to: numpy.sum

- defined by <class 'abc.Numpy_redirect_sum'>

numpy.lib.function_base.**take**()
    Alias to: numpy.take

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**transpose**()
    Alias to: numpy.transpose

- defined by numpy_transpose(a, axes=None) at numba/np/arrayobj.py:1530-1542

numpy.lib.function_base.**trapz**()
    Alias to: numpy.trapz

- defined by np_trapz(y, x=None, dx=1.0) at numba/np/arraymath.py:1833-1852

numpy.lib.function_base.**where**()
    Alias to: numpy.where

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.lib.function_base.**zeros**()
    Alias to: numpy.zeros

- defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

Not showing 26 unsupported functions.

supported = 43 / 69 = 62.32%

numpy.lib.histograms

numpy.lib.histograms.**histogram**()
    Alias to: numpy.histogram

- defined by np_histogram(a, bins=10, range=None) at numba/np/arraymath.py:3700-3780

Not showing 2 unsupported functions.

supported = 1 / 3 = 33.33%

numpy.lib.index_tricks

numpy.lib.index_tricks.**arange**()
    Alias to: numpy.arange

- defined by `np_arange(start, stop=None, step=None, dtype=None)` at numba/np/arrayobj.py:3842-3908

`numpy.lib.index_tricks.`**`array`**`()`
    Alias to: `numpy.array`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

`numpy.lib.index_tricks.`**`as_strided`**`()`

- defined by `as_strided(x, shape=None, strides=None)` at numba/np/arrayobj.py:5100-5126

`numpy.lib.index_tricks.`**`asarray`**`()`
    Alias to: `numpy.asarray`

- defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

`numpy.lib.index_tricks.`**`cumprod`**`()`
    Alias to: `numpy.cumprod`

- defined by `<class 'abc.Numpy_redirect_cumprod'>`

`numpy.lib.index_tricks.`**`diff`**`()`
    Alias to: `numpy.diff`

- defined by `np_diff_impl(a, n=1)` at numba/np/arraymath.py:3374-3410

`numpy.lib.index_tricks.`**`fill_diagonal`**`()`
    Alias to: `numpy.fill_diagonal`

- defined by `np_fill_diagonal(a, val, wrap=False)` at numba/np/arraymath.py:2926-2956

`numpy.lib.index_tricks.`**`linspace`**`()`
    Alias to: `numpy.linspace`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.`
  `<locals>.decorate.<locals>.Template'>`

Not showing 10 unsupported functions.

supported = 8 / 18 = 44.44%

numpy.lib.mixins

This module is not supported.

numpy.lib.nanfunctions

`numpy.lib.nanfunctions.`**`nancumprod`**`()`
    Alias to: `numpy.nancumprod`

- defined by `np_nancumprod(a)` at numba/np/arraymath.py:1000-1022

`numpy.lib.nanfunctions.`**`nancumsum`**`()`
    Alias to: `numpy.nancumsum`

- defined by `np_nancumsum(a)` at numba/np/arraymath.py:1025-1047

`numpy.lib.nanfunctions.`**`nanmax`**`()`
    Alias to: `numpy.nanmax`

  - defined by `np_nanmax(a)` at numba/np/arraymath.py:891-897

`numpy.lib.nanfunctions.`**`nanmean`**`()`
    Alias to: `numpy.nanmean`

  - defined by `np_nanmean(a)` at numba/np/arraymath.py:900-917

`numpy.lib.nanfunctions.`**`nanmedian`**`()`
    Alias to: `numpy.nanmedian`

  - defined by `np_nanmedian(a)` at numba/np/arraymath.py:1438-1460

`numpy.lib.nanfunctions.`**`nanmin`**`()`
    Alias to: `numpy.nanmin`

  - defined by `np_nanmin(a)` at numba/np/arraymath.py:882-888

`numpy.lib.nanfunctions.`**`nanpercentile`**`()`
    Alias to: `numpy.nanpercentile`

  - defined by `np_nanpercentile(a, q)` at numba/np/arraymath.py:1417-1421

`numpy.lib.nanfunctions.`**`nanprod`**`()`
    Alias to: `numpy.nanprod`

  - defined by `np_nanprod(a)` at numba/np/arraymath.py:978-997

`numpy.lib.nanfunctions.`**`nanquantile`**`()`
    Alias to: `numpy.nanquantile`

  - defined by `np_nanquantile(a, q)` at numba/np/arraymath.py:1431-1435

`numpy.lib.nanfunctions.`**`nanstd`**`()`
    Alias to: `numpy.nanstd`

  - defined by `np_nanstd(a)` at numba/np/arraymath.py:945-953

`numpy.lib.nanfunctions.`**`nansum`**`()`
    Alias to: `numpy.nansum`

  - defined by `np_nansum(a)` at numba/np/arraymath.py:956-975

`numpy.lib.nanfunctions.`**`nanvar`**`()`
    Alias to: `numpy.nanvar`

  - defined by `np_nanvar(a)` at numba/np/arraymath.py:920-942

Not showing 2 unsupported functions.

supported = 12 / 14 = 85.71%

numpy.lib.npyio

This module is not supported.

numpy.lib.polynomial

`numpy.lib.polynomial.`**`array`**`()`
    Alias to: `numpy.array`

  - defined by `<class 'numba.core.typing.templates.Registry.register_global.`
    `<locals>.decorate.<locals>.Template'>`

numpy.lib.polynomial.**atleast_1d**()
    Alias to: numpy.atleast_1d

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.polynomial.**diag**()
    Alias to: numpy.diag

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.polynomial.**dot**()
    Alias to: numpy.dot

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.polynomial.**eigvals**()
    Alias to: numpy.dual.eigvals

    • defined by eigvals_impl(a) at numba/np/linalg.py:1060-1177

numpy.lib.polynomial.**hstack**()
    Alias to: numpy.hstack

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.polynomial.**imag**()
    Alias to: numpy.imag

    • defined by np_imag(a) at numba/np/arraymath.py:3295-3300

numpy.lib.polynomial.**inv**()
    Alias to: numpy.dual.inv

    • defined by inv_impl(a) at numba/np/linalg.py:833-870

numpy.lib.polynomial.**lstsq**()
    Alias to: numpy.dual.lstsq

    • defined by lstsq_impl(a, b, rcond=-1.0) at numba/np/linalg.py:1583-1675

numpy.lib.polynomial.**ones**()
    Alias to: numpy.ones

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

numpy.lib.polynomial.**real**()
    Alias to: numpy.real

    • defined by np_real(a) at numba/np/arraymath.py:3287-3292

numpy.lib.polynomial.**roots**()
    Alias to: numpy.roots

    • defined by roots_impl(p) at numba/np/polynomial.py:14-59

numpy.lib.polynomial.**vander**()
    Alias to: numpy.vander

    • defined by np_vander(x, N=None, increasing=False) at numba/np/arraymath.py:1891-1928

Not showing 14 unsupported functions.

supported = 13 / 27 = 48.15%

numpy.lib.recfunctions

This module is not supported.

numpy.lib.scimath

numpy.lib.scimath.**any**()
>   Alias to: numpy.any
>
>   • defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.lib.scimath.**asarray**()
>   Alias to: numpy.asarray
>
>   • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

Not showing 11 unsupported functions.

supported = 2 / 13 = 15.38%

numpy.lib.setup

This module is not supported.

numpy.lib.shape_base

numpy.lib.shape_base.**array**()
>   Alias to: numpy.array
>
>   • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**asarray**()
>   Alias to: numpy.asarray
>
>   • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.lib.shape_base.**atleast_3d**()
>   Alias to: numpy.atleast_3d
>
>   • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**column_stack**()
>   Alias to: numpy.column_stack
>
>   • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**concatenate**()
>   Alias to: numpy.concatenate
>
>   • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**dstack**()
     Alias to: numpy.dstack

         • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**expand_dims**()
     Alias to: numpy.expand_dims

         • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**kron**()
     Alias to: numpy.kron

         • defined by kron_impl(a, b) at numba/np/linalg.py:2722-2776

numpy.lib.shape_base.**outer**()
     Alias to: numpy.outer

         • defined by outer_impl(a, b, out=None) at numba/np/linalg.py:2645-2656

numpy.lib.shape_base.**reshape**()
     Alias to: numpy.reshape

         • defined by np_reshape(a, shape) at numba/np/arrayobj.py:1702-1706

numpy.lib.shape_base.**row_stack**()
     Alias to: numpy.row_stack

         • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**transpose**()
     Alias to: numpy.transpose

         • defined by numpy_transpose(a, axes=None) at numba/np/arrayobj.py:1530-1542

numpy.lib.shape_base.**vstack**()
     Alias to: numpy.row_stack

         • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.lib.shape_base.**zeros**()
     Alias to: numpy.zeros

         • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

Not showing 15 unsupported functions.

supported = 14 / 29 = 48.28%

numpy.lib.stride_tricks

numpy.lib.stride_tricks.**as_strided**()
     Alias to: numpy.lib.index_tricks.as_strided

         • defined by as_strided(x, shape=None, strides=None) at numba/np/arrayobj.py:5100-5126

Not showing 3 unsupported functions.

supported = 1 / 4 = 25.00%

numpy.lib.twodim_base

numpy.lib.twodim_base.**arange**()
>   Alias to: `numpy.arange`
>
>   • defined   by   `np_arange(start, stop=None, step=None, dtype=None)`   at
>   numba/np/arrayobj.py:3842-3908

numpy.lib.twodim_base.**asarray**()
>   Alias to: `numpy.asarray`
>
>   • defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

numpy.lib.twodim_base.**diag**()
>   Alias to: `numpy.diag`
>
>   • defined   by   `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

numpy.lib.twodim_base.**empty**()
>   Alias to: `numpy.empty`
>
>   • defined   by   `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

numpy.lib.twodim_base.**eye**()
>   Alias to: `numpy.eye`
>
>   • defined   by   `numpy_eye(N, M=None, k=0, dtype=<class 'float'>)`   at
>   numba/np/arrayobj.py:3681-3704

numpy.lib.twodim_base.**fliplr**()
>   Alias to: `numpy.fliplr`
>
>   • defined by `np_flip_lr(a)` at numba/np/arrayobj.py:4891-4905

numpy.lib.twodim_base.**flipud**()
>   Alias to: `numpy.flipud`
>
>   • defined by `np_flip_ud(a)` at numba/np/arrayobj.py:4908-4922

numpy.lib.twodim_base.**nonzero**()
>   Alias to: `numpy.nonzero`
>
>   • defined by `<class 'abc.Numpy_redirect_nonzero'>`

numpy.lib.twodim_base.**ones**()
>   Alias to: `numpy.ones`
>
>   • defined   by   `<class 'numba.core.typing.templates.Registry.register_global.`
>   `<locals>.decorate.<locals>.Template'>`

numpy.lib.twodim_base.**tri**()
>   Alias to: `numpy.tri`
>
>   • defined by `np_tri(N, M=None, k=0)` at numba/np/arraymath.py:1560-1571

numpy.lib.twodim_base.**tril**()
>   Alias to: `numpy.tril`

- defined by `my_tril(m, k=0)` at [numba/np/arraymath.py:1597-1621](#)

`numpy.lib.twodim_base.`**`tril_indices`**`()`
    Alias to: `numpy.tril_indices`

- defined by `np_tril_indices(n, k=0, m=None)` at [numba/np/arraymath.py:1624-1635](#)

`numpy.lib.twodim_base.`**`tril_indices_from`**`()`
    Alias to: `numpy.tril_indices_from`

- defined by `np_tril_indices_from(arr, k=0)` at [numba/np/arraymath.py:1638-1649](#)

`numpy.lib.twodim_base.`**`triu`**`()`
    Alias to: `numpy.triu`

- defined by `my_triu(m, k=0)` at [numba/np/arraymath.py:1658-1681](#)

`numpy.lib.twodim_base.`**`triu_indices`**`()`
    Alias to: `numpy.triu_indices`

- defined by `np_triu_indices(n, k=0, m=None)` at [numba/np/arraymath.py:1684-1695](#)

`numpy.lib.twodim_base.`**`triu_indices_from`**`()`
    Alias to: `numpy.triu_indices_from`

- defined by `np_triu_indices_from(arr, k=0)` at [numba/np/arraymath.py:1698-1709](#)

`numpy.lib.twodim_base.`**`vander`**`()`
    Alias to: `numpy.vander`

- defined by `np_vander(x, N=None, increasing=False)` at [numba/np/arraymath.py:1891-1928](#)

`numpy.lib.twodim_base.`**`where`**`()`
    Alias to: `numpy.where`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.lib.twodim_base.`**`zeros`**`()`
    Alias to: `numpy.zeros`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 7 unsupported functions.

supported = 19 / 26 = 73.08%

numpy.lib.type_check

`numpy.lib.type_check.`**`asarray`**`()`
    Alias to: `numpy.asarray`

- defined by `np_asarray(a, dtype=None)` at [numba/np/arraymath.py:4018-4070](#)

`numpy.lib.type_check.`**`imag`**`()`
    Alias to: `numpy.imag`

- defined by `np_imag(a)` at [numba/np/arraymath.py:3295-3300](#)

`numpy.lib.type_check.`**`real`**`()`
    Alias to: `numpy.real`

- defined by `np_real(a)` at numba/np/arraymath.py:3287-3292

`numpy.lib.type_check.`**`zeros`**`()`
> Alias to: `numpy.zeros`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

Not showing 15 unsupported functions.

supported = 4 / 19 = 21.05%

numpy.lib.ufunclike

This module is not supported.

numpy.lib.user_array

`numpy.lib.user_array.`**`arange`**`()`
> Alias to: `numpy.arange`

> - defined by `np_arange(start, stop=None, step=None, dtype=None)` at numba/np/arrayobj.py:3842-3908

`numpy.lib.user_array.`**`array`**`()`
> Alias to: `numpy.array`

> - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.lib.user_array.`**`asarray`**`()`
> Alias to: `numpy.asarray`

> - defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

`numpy.lib.user_array.`**`reshape`**`()`
> Alias to: `numpy.reshape`

> - defined by `np_reshape(a, shape)` at numba/np/arrayobj.py:1702-1706

`numpy.lib.user_array.`**`shape`**`()`
> Alias to: `numpy.shape`

> - defined by `np_shape(a)` at numba/np/arrayobj.py:1903-1910

`numpy.lib.user_array.`**`transpose`**`()`
> Alias to: `numpy.transpose`

> - defined by `numpy_transpose(a, axes=None)` at numba/np/arrayobj.py:1530-1542

Not showing 0 unsupported functions.

supported = 6 / 6 = 100.00%

numpy.lib.utils

`numpy.lib.utils.`**`asarray`**`()`
> Alias to: `numpy.asarray`

> - defined by `np_asarray(a, dtype=None)` at numba/np/arraymath.py:4018-4070

Not showing 15 unsupported functions.

supported = 1 / 16 = 6.25%

numpy.linalg

numpy.linalg.**cholesky**()
    Alias to: numpy.dual.cholesky

- defined by cho_impl(a) at numba/np/linalg.py:901-943

numpy.linalg.**cond**()

- defined by cond_impl(a, p=None) at numba/np/linalg.py:2361-2403

numpy.linalg.**det**()
    Alias to: numpy.dual.det

- defined by det_impl(a) at numba/np/linalg.py:2018-2029

numpy.linalg.**eig**()
    Alias to: numpy.dual.eig

- defined by eig_impl(a) at numba/np/linalg.py:945-1058

numpy.linalg.**eigh**()
    Alias to: numpy.dual.eigh

- defined by eigh_impl(a) at numba/np/linalg.py:1179-1226

numpy.linalg.**eigvals**()
    Alias to: numpy.dual.eigvals

- defined by eigvals_impl(a) at numba/np/linalg.py:1060-1177

numpy.linalg.**eigvalsh**()
    Alias to: numpy.dual.eigvalsh

- defined by eigvalsh_impl(a) at numba/np/linalg.py:1228-1275

numpy.linalg.**inv**()
    Alias to: numpy.dual.inv

- defined by inv_impl(a) at numba/np/linalg.py:833-870

numpy.linalg.**lstsq**()
    Alias to: numpy.dual.lstsq

- defined by lstsq_impl(a, b, rcond=-1.0) at numba/np/linalg.py:1583-1675

numpy.linalg.**matrix_power**()

- defined by matrix_power_impl(a, n) at numba/np/linalg.py:2486-2562

numpy.linalg.**matrix_rank**()

- defined by matrix_rank_impl(a, tol=None) at numba/np/linalg.py:2420-2483

numpy.linalg.**norm**()
    Alias to: numpy.dual.norm

- defined by norm_impl(a, ord=None) at numba/np/linalg.py:2352-2358

numpy.linalg.**pinv**()
    Alias to: numpy.dual.pinv

- defined by `pinv_impl(a, rcond=1e-15)` at [numba/np/linalg.py:1759-1920](#)

numpy.linalg.**qr**()

- defined by `qr_impl(a)` at [numba/np/linalg.py:1343-1417](#)

numpy.linalg.**slogdet**()

- defined by `slogdet_impl(a)` at [numba/np/linalg.py:1959-2015](#)

numpy.linalg.**solve**()
    Alias to: `numpy.dual.solve`

- defined by `solve_impl(a, b)` at [numba/np/linalg.py:1698-1756](#)

numpy.linalg.**svd**()
    Alias to: `numpy.dual.svd`

- defined by `svd_impl(a, full_matrices=1)` at [numba/np/linalg.py:1277-1340](#)

Not showing 3 unsupported functions.

supported = 17 / 20 = 85.00%

numpy.linalg.lapack_lite

This module is not supported.

numpy.linalg.linalg

numpy.linalg.linalg.**all**()
    Alias to: `numpy.all`

- defined by `np_all(a)` at [numba/np/arraymath.py:776-785](#)

numpy.linalg.linalg.**amax**()
    Alias to: `numpy.amax`

- defined by `<class 'abc.Numpy_redirect_max'>`

numpy.linalg.linalg.**amin**()
    Alias to: `numpy.amin`

- defined by `<class 'abc.Numpy_redirect_min'>`

numpy.linalg.linalg.**argsort**()
    Alias to: `numpy.argsort`

- defined by `<class 'abc.Numpy_redirect_argsort'>`

numpy.linalg.linalg.**array**()
    Alias to: `numpy.array`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.linalg.linalg.**asarray**()
    Alias to: `numpy.asarray`

- defined by `np_asarray(a, dtype=None)` at [numba/np/arraymath.py:4018-4070](#)

numpy.linalg.linalg.**atleast_2d**()
    Alias to: `numpy.atleast_2d`

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.linalg.linalg.**cholesky**()
    Alias to: numpy.dual.cholesky

- defined by cho_impl(a) at numba/np/linalg.py:901-943

numpy.linalg.linalg.**cond**()
    Alias to: numpy.linalg.cond

- defined by cond_impl(a, p=None) at numba/np/linalg.py:2361-2403

numpy.linalg.linalg.**count_nonzero**()
    Alias to: numpy.count_nonzero

- defined by np_count_nonzero(arr, axis=None) at numba/np/arraymath.py:3306-3320

numpy.linalg.linalg.**det**()
    Alias to: numpy.dual.det

- defined by det_impl(a) at numba/np/linalg.py:2018-2029

numpy.linalg.linalg.**dot**()
    Alias to: numpy.dot

- defined by <class 'numpy.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.linalg.linalg.**eig**()
    Alias to: numpy.dual.eig

- defined by eig_impl(a) at numba/np/linalg.py:945-1058

numpy.linalg.linalg.**eigh**()
    Alias to: numpy.dual.eigh

- defined by eigh_impl(a) at numba/np/linalg.py:1179-1226

numpy.linalg.linalg.**eigvals**()
    Alias to: numpy.dual.eigvals

- defined by eigvals_impl(a) at numba/np/linalg.py:1060-1177

numpy.linalg.linalg.**eigvalsh**()
    Alias to: numpy.dual.eigvalsh

- defined by eigvalsh_impl(a) at numba/np/linalg.py:1228-1275

numpy.linalg.linalg.**empty**()
    Alias to: numpy.empty

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.linalg.linalg.**empty_like**()
    Alias to: numpy.empty_like

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.linalg.linalg.**eye**()
    Alias to: numpy.eye

- defined by numpy_eye(N, M=None, k=0, dtype=<class 'float'>) at
  numba/np/arrayobj.py:3681-3704

numpy.linalg.linalg.**inv**()
    Alias to: numpy.dual.inv

    - defined by `inv_impl(a)` at numba/np/linalg.py:833-870

numpy.linalg.linalg.**lstsq**()
    Alias to: numpy.dual.lstsq

    - defined by `lstsq_impl(a, b, rcond=-1.0)` at numba/np/linalg.py:1583-1675

numpy.linalg.linalg.**matrix_power**()
    Alias to: numpy.linalg.matrix_power

    - defined by `matrix_power_impl(a, n)` at numba/np/linalg.py:2486-2562

numpy.linalg.linalg.**matrix_rank**()
    Alias to: numpy.linalg.matrix_rank

    - defined by `matrix_rank_impl(a, tol=None)` at numba/np/linalg.py:2420-2483

numpy.linalg.linalg.**norm**()
    Alias to: numpy.dual.norm

    - defined by `norm_impl(a, ord=None)` at numba/np/linalg.py:2352-2358

numpy.linalg.linalg.**pinv**()
    Alias to: numpy.dual.pinv

    - defined by `pinv_impl(a, rcond=1e-15)` at numba/np/linalg.py:1759-1920

numpy.linalg.linalg.**qr**()
    Alias to: numpy.linalg.qr

    - defined by `qr_impl(a)` at numba/np/linalg.py:1343-1417

numpy.linalg.linalg.**slogdet**()
    Alias to: numpy.linalg.slogdet

    - defined by `slogdet_impl(a)` at numba/np/linalg.py:1959-2015

numpy.linalg.linalg.**solve**()
    Alias to: numpy.dual.solve

    - defined by `solve_impl(a, b)` at numba/np/linalg.py:1698-1756

numpy.linalg.linalg.**sort**()
    Alias to: numpy.sort

    - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.linalg.linalg.**sum**()
    Alias to: numpy.sum

    - defined by `<class 'abc.Numpy_redirect_sum'>`

numpy.linalg.linalg.**svd**()
    Alias to: numpy.dual.svd

    - defined by `svd_impl(a, full_matrices=1)` at numba/np/linalg.py:1277-1340

numpy.linalg.linalg.**triu**()
    Alias to: numpy.triu

    - defined by `my_triu(m, k=0)` at numba/np/arraymath.py:1658-1681

numpy.linalg.linalg.**zeros**()
    Alias to: numpy.zeros

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

Not showing 14 unsupported functions.

supported = 33 / 47 = 70.21%

numpy.linalg.setup

This module is not supported.

numpy.ma

numpy.ma.**amax**()
    Alias to: numpy.amax

    • defined by <class 'abc.Numpy_redirect_max'>

numpy.ma.**amin**()
    Alias to: numpy.amin

    • defined by <class 'abc.Numpy_redirect_min'>

numpy.ma.**diff**()
    Alias to: numpy.diff

    • defined by np_diff_impl(a, n=1) at numba/np/arraymath.py:3374-3410

numpy.ma.**expand_dims**()
    Alias to: numpy.expand_dims

    • defined by <class 'numba.core.typing.templates.Registry.register_global.
      <locals>.decorate.<locals>.Template'>

Not showing 106 unsupported functions.

supported = 4 / 110 = 3.64%

numpy.ma.bench

This module is not supported.

numpy.ma.core

numpy.ma.core.**amax**()
    Alias to: numpy.amax

    • defined by <class 'abc.Numpy_redirect_max'>

numpy.ma.core.**amin**()
    Alias to: numpy.amin

    • defined by <class 'abc.Numpy_redirect_min'>

numpy.ma.core.**diff**()
    Alias to: numpy.diff

> • defined by np_diff_impl(a, n=1) at numba/np/arraymath.py:3374-3410

numpy.ma.core.**expand_dims**()
    Alias to: numpy.expand_dims

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.ma.core.**narray**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.ma.core.**ones_like**()
    Alias to: numpy.ones_like

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.ma.core.**reduce**()
    Alias to: numpy.distutils.system_info.reduce

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.ma.core.**zeros_like**()
    Alias to: numpy.zeros_like

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 87 unsupported functions.

supported = 8 / 95 = 8.42%

numpy.ma.extras

numpy.ma.extras.**nxarray**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 48 unsupported functions.

supported = 1 / 49 = 2.04%

numpy.ma.mrecords

numpy.ma.mrecords.**narray**()
    Alias to: numpy.array

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

Not showing 10 unsupported functions.

supported = 1 / 11 = 9.09%

numpy.ma.setup

This module is not supported.

numpy.ma.testutils

This module is not supported.

numpy.ma.timer_comparison

numpy.ma.timer_comparison.**reduce**()
>   Alias to: numpy.distutils.system_info.reduce
>
>   - defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

Not showing 1 unsupported functions.

supported = 1 / 2 = 50.00%

numpy.matlib

numpy.matlib.**all**()
>   Alias to: numpy.all
>
>   - defined by np_all(a) at numba/np/arraymath.py:776-785

numpy.matlib.**amax**()
>   Alias to: numpy.amax
>
>   - defined by <class 'abc.Numpy_redirect_max'>

numpy.matlib.**amin**()
>   Alias to: numpy.amin
>
>   - defined by <class 'abc.Numpy_redirect_min'>

numpy.matlib.**angle**()
>   Alias to: numpy.angle
>
>   - defined by <class 'numba.core.typing.templates.Registry.register_global. <locals>.decorate.<locals>.Template'>

numpy.matlib.**any**()
>   Alias to: numpy.any
>
>   - defined by np_any(a) at numba/np/arraymath.py:788-797

numpy.matlib.**append**()
>   Alias to: numpy.append
>
>   - defined by np_append(arr, values, axis=None) at numba/np/arrayobj.py:1709-1732

numpy.matlib.**arange**()
>   Alias to: numpy.arange
>
>   - defined by np_arange(start, stop=None, step=None, dtype=None) at numba/np/arrayobj.py:3842-3908

numpy.matlib.**argmax**()
    Alias to: numpy.argmax

> • defined by <class 'abc.Numpy_redirect_argmax'>

numpy.matlib.**argmin**()
    Alias to: numpy.argmin

> • defined by <class 'abc.Numpy_redirect_argmin'>

numpy.matlib.**argsort**()
    Alias to: numpy.argsort

> • defined by <class 'abc.Numpy_redirect_argsort'>

numpy.matlib.**argwhere**()
    Alias to: numpy.argwhere

> • defined by np_argwhere(a) at numba/np/arraymath.py:2789-2821

numpy.matlib.**around**()
    Alias to: numpy.around

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**array**()
    Alias to: numpy.array

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**array_equal**()
    Alias to: numpy.array_equal

> • defined by np_array_equal(a, b) at numba/np/arraymath.py:3413-3432

numpy.matlib.**asarray**()
    Alias to: numpy.asarray

> • defined by np_asarray(a, dtype=None) at numba/np/arraymath.py:4018-4070

numpy.matlib.**ascontiguousarray**()
    Alias to: numpy.ascontiguousarray

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**asfortranarray**()
    Alias to: numpy.asfortranarray

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**atleast_1d**()
    Alias to: numpy.atleast_1d

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**atleast_2d**()
    Alias to: numpy.atleast_2d

> • defined by <class 'numpy.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**atleast_3d**()
    Alias to: numpy.atleast_3d

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.matlib.**bartlett**()
    Alias to: numpy.bartlett

> • defined   by   window_generator.<locals>.window_overload(M)   at
> numba/np/arraymath.py:4183-4195

numpy.matlib.**bincount**()
    Alias to: numpy.bincount

> • defined by np_bincount(a, weights=None, minlength=0) at numba/np/arraymath.py:3445-
> 3500

numpy.matlib.**blackman**()
    Alias to: numpy.blackman

> • defined   by   window_generator.<locals>.window_overload(M)   at
> numba/np/arraymath.py:4183-4195

numpy.matlib.**column_stack**()
    Alias to: numpy.column_stack

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.matlib.**concatenate**()
    Alias to: numpy.concatenate

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.matlib.**convolve**()
    Alias to: numpy.convolve

> • defined by np_convolve(a, v) at numba/np/arraymath.py:3994-4015

numpy.matlib.**copy**()
    Alias to: numpy.copy

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
> <locals>.decorate.<locals>.Template'>

numpy.matlib.**corrcoef**()
    Alias to: numpy.corrcoef

> • defined by np_corrcoef(x, y=None, rowvar=True) at numba/np/arraymath.py:2752-2782

numpy.matlib.**correlate**()
    Alias to: numpy.correlate

> • defined by _np_correlate(a, v) at numba/np/arraymath.py:3946-3991

numpy.matlib.**count_nonzero**()
    Alias to: numpy.count_nonzero

> • defined by np_count_nonzero(arr, axis=None) at numba/np/arraymath.py:3306-3320

numpy.matlib.**cov**()
    Alias to: numpy.cov

- defined by np_cov(m, y=None, rowvar=True, bias=False, ddof=None) at numba/np/arraymath.py:2693-2749

numpy.matlib.**cross**()
    Alias to: numpy.cross

- defined by np_cross(a, b) at numba/np/arraymath.py:4364-4386

numpy.matlib.**cumprod**()
    Alias to: numpy.cumprod

- defined by <class 'abc.Numpy_redirect_cumprod'>

numpy.matlib.**cumsum**()
    Alias to: numpy.cumsum

- defined by <class 'abc.Numpy_redirect_cumsum'>

numpy.matlib.**delete**()
    Alias to: numpy.delete

- defined by np_delete(arr, obj) at numba/np/arraymath.py:3327-3371

numpy.matlib.**diag**()
    Alias to: numpy.diag

- defined by <class 'numpy.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.matlib.**diff**()
    Alias to: numpy.diff

- defined by np_diff_impl(a, n=1) at numba/np/arraymath.py:3374-3410

numpy.matlib.**digitize**()
    Alias to: numpy.digitize

- defined by np_digitize(x, bins, right=False) at numba/np/arraymath.py:3568-3694

numpy.matlib.**dot**()
    Alias to: numpy.dot

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.matlib.**dstack**()
    Alias to: numpy.dstack

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.matlib.**ediff1d**()
    Alias to: numpy.ediff1d

- defined by np_ediff1d(ary, to_end=None, to_begin=None) at numba/np/arraymath.py:1741-1796

numpy.matlib.**empty_like**()
    Alias to: numpy.empty_like

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.matlib.**expand_dims**()
    Alias to: numpy.expand_dims

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**extract**()
    Alias to: numpy.extract

> - defined by np_extract(condition, arr) at numba/np/arraymath.py:4073-4096

numpy.matlib.**fill_diagonal**()
    Alias to: numpy.fill_diagonal

> - defined by np_fill_diagonal(a, val, wrap=False) at numba/np/arraymath.py:2926-2956

numpy.matlib.**flatnonzero**()
    Alias to: numpy.flatnonzero

> - defined by np_flatnonzero(a) at numba/np/arraymath.py:2824-2839

numpy.matlib.**flip**()
    Alias to: numpy.flip

> - defined by np_flip(a) at numba/np/arrayobj.py:4953-4964

numpy.matlib.**fliplr**()
    Alias to: numpy.fliplr

> - defined by np_flip_lr(a) at numba/np/arrayobj.py:4891-4905

numpy.matlib.**flipud**()
    Alias to: numpy.flipud

> - defined by np_flip_ud(a) at numba/np/arrayobj.py:4908-4922

numpy.matlib.**frombuffer**()
    Alias to: numpy.frombuffer

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**full**()
    Alias to: numpy.full

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**full_like**()
    Alias to: numpy.full_like

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matlib.**hamming**()
    Alias to: numpy.hamming

> - defined by window_generator.<locals>.window_overload(M) at
>   numba/np/arraymath.py:4183-4195

numpy.matlib.**hanning**()
    Alias to: numpy.hanning

> - defined by window_generator.<locals>.window_overload(M) at
>   numba/np/arraymath.py:4183-4195

numpy.matlib.**histogram**()
    Alias to: numpy.histogram

- defined by `np_histogram(a, bins=10, range=None)` at numba/np/arraymath.py:3700-3780

numpy.matlib.**hstack**()
    Alias to: `numpy.hstack`

    - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.matlib.**imag**()
    Alias to: `numpy.imag`

    - defined by `np_imag(a)` at numba/np/arraymath.py:3295-3300

numpy.matlib.**interp**()
    Alias to: `numpy.interp`

    - defined by `np_interp(x, xp, fp)` at numba/np/arraymath.py:2444-2505

numpy.matlib.**kaiser**()
    Alias to: `numpy.kaiser`

    - defined by `np_kaiser(M, beta)` at numba/np/arraymath.py:4301-4320

numpy.matlib.**kron**()
    Alias to: `numpy.kron`

    - defined by `kron_impl(a, b)` at numba/np/linalg.py:2722-2776

numpy.matlib.**linspace**()
    Alias to: `numpy.linspace`

    - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.matlib.**mean**()
    Alias to: `numpy.mean`

    - defined by `<class 'abc.Numpy_redirect_mean'>`

numpy.matlib.**median**()
    Alias to: `numpy.median`

    - defined by `np_median(a)` at numba/np/arraymath.py:1254-1266

numpy.matlib.**nancumprod**()
    Alias to: `numpy.nancumprod`

    - defined by `np_nancumprod(a)` at numba/np/arraymath.py:1000-1022

numpy.matlib.**nancumsum**()
    Alias to: `numpy.nancumsum`

    - defined by `np_nancumsum(a)` at numba/np/arraymath.py:1025-1047

numpy.matlib.**nanmax**()
    Alias to: `numpy.nanmax`

    - defined by `np_nanmax(a)` at numba/np/arraymath.py:891-897

numpy.matlib.**nanmean**()
    Alias to: `numpy.nanmean`

    - defined by `np_nanmean(a)` at numba/np/arraymath.py:900-917

numpy.matlib.**nanmedian**()
    Alias to: `numpy.nanmedian`

   • defined by `np_nanmedian(a)` at numba/np/arraymath.py:1438-1460

`numpy.matlib.`**`nanmin`**`()`
    Alias to: `numpy.nanmin`

   • defined by `np_nanmin(a)` at numba/np/arraymath.py:882-888

`numpy.matlib.`**`nanpercentile`**`()`
    Alias to: `numpy.nanpercentile`

   • defined by `np_nanpercentile(a, q)` at numba/np/arraymath.py:1417-1421

`numpy.matlib.`**`nanprod`**`()`
    Alias to: `numpy.nanprod`

   • defined by `np_nanprod(a)` at numba/np/arraymath.py:978-997

`numpy.matlib.`**`nanquantile`**`()`
    Alias to: `numpy.nanquantile`

   • defined by `np_nanquantile(a, q)` at numba/np/arraymath.py:1431-1435

`numpy.matlib.`**`nanstd`**`()`
    Alias to: `numpy.nanstd`

   • defined by `np_nanstd(a)` at numba/np/arraymath.py:945-953

`numpy.matlib.`**`nansum`**`()`
    Alias to: `numpy.nansum`

   • defined by `np_nansum(a)` at numba/np/arraymath.py:956-975

`numpy.matlib.`**`nanvar`**`()`
    Alias to: `numpy.nanvar`

   • defined by `np_nanvar(a)` at numba/np/arraymath.py:920-942

`numpy.matlib.`**`nonzero`**`()`
    Alias to: `numpy.nonzero`

   • defined by `<class 'abc.Numpy_redirect_nonzero'>`

`numpy.matlib.`**`ones_like`**`()`
    Alias to: `numpy.ones_like`

   • defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.matlib.`**`outer`**`()`
    Alias to: `numpy.outer`

   • defined by `outer_impl(a, b, out=None)` at numba/np/linalg.py:2645-2656

`numpy.matlib.`**`partition`**`()`
    Alias to: `numpy.partition`

   • defined by `np_partition(a, kth)` at numba/np/arraymath.py:1519-1541

`numpy.matlib.`**`percentile`**`()`
    Alias to: `numpy.percentile`

   • defined by `np_percentile(a, q)` at numba/np/arraymath.py:1410-1414

`numpy.matlib.`**`prod`**`()`
    Alias to: `numpy.prod`

   • defined by `<class 'abc.Numpy_redirect_prod'>`

numpy.matlib.**ptp**()
    Alias to: numpy.ptp

        • defined by np_ptp(a) at numba/np/arraymath.py:1109-1134

numpy.matlib.**quantile**()
    Alias to: numpy.quantile

        • defined by np_quantile(a, q) at numba/np/arraymath.py:1424-1428

numpy.matlib.**ravel**()
    Alias to: numpy.ravel

        • defined by <class 'abc.Numpy_redirect_ravel'>

numpy.matlib.**real**()
    Alias to: numpy.real

        • defined by np_real(a) at numba/np/arraymath.py:3287-3292

numpy.matlib.**repeat**()
    Alias to: numpy.repeat

        • defined by np_repeat(a, repeats) at numba/np/arrayobj.py:1925-1967

numpy.matlib.**reshape**()
    Alias to: numpy.reshape

        • defined by np_reshape(a, shape) at numba/np/arrayobj.py:1702-1706

numpy.matlib.**roll**()
    Alias to: numpy.roll

        • defined by np_roll(a, shift) at numba/np/arraymath.py:1931-1951

numpy.matlib.**roots**()
    Alias to: numpy.roots

        • defined by roots_impl(p) at numba/np/polynomial.py:14-59

numpy.matlib.**round_**()
    Alias to: numpy.round

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.matlib.**row_stack**()
    Alias to: numpy.row_stack

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
           <locals>.decorate.<locals>.Template'>

numpy.matlib.**searchsorted**()
    Alias to: numpy.searchsorted

        • defined by searchsorted(a, v, side='left') at numba/np/arraymath.py:3533-3565

numpy.matlib.**select**()
    Alias to: numpy.select

        • defined by np_select(condlist, choicelist, default=0) at
        numba/np/arraymath.py:4099-4147

numpy.matlib.**shape**()
    Alias to: numpy.shape

- defined by `np_shape(a)` at numba/np/arrayobj.py:1903-1910

`numpy.matlib.`**`sinc`**`()`
   Alias to: `numpy.sinc`

   - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.matlib.`**`sort`**`()`
   Alias to: `numpy.sort`

   - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.matlib.`**`stack`**`()`
   Alias to: `numpy.stack`

   - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.matlib.`**`std`**`()`
   Alias to: `numpy.std`

   - defined by `<class 'abc.Numpy_redirect_std'>`

`numpy.matlib.`**`sum`**`()`
   Alias to: `numpy.sum`

   - defined by `<class 'abc.Numpy_redirect_sum'>`

`numpy.matlib.`**`take`**`()`
   Alias to: `numpy.take`

   - defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

`numpy.matlib.`**`trace`**`()`
   Alias to: `numpy.trace`

   - defined by `matrix_trace_impl(a, offset=0)` at numba/np/linalg.py:2567-2595

`numpy.matlib.`**`transpose`**`()`
   Alias to: `numpy.transpose`

   - defined by `numpy_transpose(a, axes=None)` at numba/np/arrayobj.py:1530-1542

`numpy.matlib.`**`trapz`**`()`
   Alias to: `numpy.trapz`

   - defined by `np_trapz(y, x=None, dx=1.0)` at numba/np/arraymath.py:1833-1852

`numpy.matlib.`**`tri`**`()`
   Alias to: `numpy.tri`

   - defined by `np_tri(N, M=None, k=0)` at numba/np/arraymath.py:1560-1571

`numpy.matlib.`**`tril`**`()`
   Alias to: `numpy.tril`

   - defined by `my_tril(m, k=0)` at numba/np/arraymath.py:1597-1621

`numpy.matlib.`**`tril_indices`**`()`
   Alias to: `numpy.tril_indices`

   - defined by `np_tril_indices(n, k=0, m=None)` at numba/np/arraymath.py:1624-1635

numpy.matlib.**tril_indices_from**()
    Alias to: numpy.tril_indices_from

       • defined by np_tril_indices_from(arr, k=0) at numba/np/arraymath.py:1638-1649

numpy.matlib.**triu**()
    Alias to: numpy.triu

       • defined by my_triu(m, k=0) at numba/np/arraymath.py:1658-1681

numpy.matlib.**triu_indices**()
    Alias to: numpy.triu_indices

       • defined by np_triu_indices(n, k=0, m=None) at numba/np/arraymath.py:1684-1695

numpy.matlib.**triu_indices_from**()
    Alias to: numpy.triu_indices_from

       • defined by np_triu_indices_from(arr, k=0) at numba/np/arraymath.py:1698-1709

numpy.matlib.**unique**()
    Alias to: numpy.unique

       • defined by np_unique(a) at numba/np/arrayobj.py:1915-1922

numpy.matlib.**vander**()
    Alias to: numpy.vander

       • defined by np_vander(x, N=None, increasing=False) at numba/np/arraymath.py:1891-1928

numpy.matlib.**var**()
    Alias to: numpy.var

       • defined by <class 'abc.Numpy_redirect_var'>

numpy.matlib.**vdot**()
    Alias to: numpy.vdot

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.matlib.**vstack**()
    Alias to: numpy.row_stack

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.matlib.**where**()
    Alias to: numpy.where

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

numpy.matlib.**zeros_like**()
    Alias to: numpy.zeros_like

       • defined by <class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>

Not showing 202 unsupported functions.

supported = 117 / 319 = 36.68%

numpy.matrixlib

This module is not supported.

numpy.matrixlib.defmatrix

numpy.matrixlib.defmatrix.**concatenate**()
    Alias to: numpy.concatenate

> • defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.matrixlib.defmatrix.**matrix_power**()
    Alias to: numpy.linalg.matrix_power

> • defined by matrix_power_impl(a, n) at numba/np/linalg.py:2486-2562

Not showing 5 unsupported functions.

supported = 2 / 7 = 28.57%

numpy.matrixlib.setup

This module is not supported.

numpy.polynomial

This module is not supported.

numpy.polynomial.chebyshev

This module is not supported.

numpy.polynomial.hermite

This module is not supported.

numpy.polynomial.hermite_e

This module is not supported.

numpy.polynomial.laguerre

This module is not supported.

numpy.polynomial.legendre

This module is not supported.

numpy.polynomial.polynomial

This module is not supported.

numpy.polynomial.polyutils

This module is not supported.

numpy.polynomial.setup

This module is not supported.

numpy.random

numpy.random.**beta**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**binomial**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**chisquare**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**choice**()

- defined by choice(a, size=None, replace=True) at numba/cpython/randomimpl.py:1346-
  1427

numpy.random.**exponential**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**f**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**gamma**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**geometric**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**gumbel**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**hypergeometric**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**laplace**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**logistic**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**lognormal**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**logseries**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**multinomial**()

- defined by `multinomial(n, pvals, size=None)` at numba/cpython/randomimpl.py:1433-1508

numpy.random.**negative_binomial**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**normal**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**pareto**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**permutation**()

- defined by `permutation_impl(x)` at numba/cpython/randomimpl.py:1231-1245

numpy.random.**poisson**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**power**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**rand**()

- defined by `rand(*size)` at numba/cpython/randomimpl.py:1314-1326

numpy.random.**randint**()

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.**randn**()

- defined by `randn(*size)` at numba/cpython/randomimpl.py:1328-1340

numpy.random.**random**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**random_sample**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**ranf**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**rayleigh**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**sample**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**seed**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**shuffle**()

> - defined by shuffle_impl(arr) at numba/cpython/randomimpl.py:1227-1229

numpy.random.**standard_cauchy**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**standard_exponential**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**standard_gamma**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**standard_normal**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**standard_t**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**triangular**()

> - defined by <class 'numba.core.typing.templates.Registry.register_global.
>   <locals>.decorate.<locals>.Template'>

numpy.random.**uniform**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**vonmises**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**wald**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**weibull**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.**zipf**()

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

Not showing 9 unsupported functions.

supported = 42 / 51 = 82.35%

numpy.random.bit_generator

This module is not supported.

numpy.random.mtrand

numpy.random.mtrand.**beta**()
    Alias to: numpy.random.beta

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**binomial**()
    Alias to: numpy.random.binomial

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**chisquare**()
    Alias to: numpy.random.chisquare

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**choice**()
    Alias to: numpy.random.choice

- defined by choice(a, size=None, replace=True) at numba/cpython/randomimpl.py:1346-
  1427

numpy.random.mtrand.**exponential**()
    Alias to: numpy.random.exponential

- defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**f**()
 Alias to: numpy.random.f

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**gamma**()
 Alias to: numpy.random.gamma

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**geometric**()
 Alias to: numpy.random.geometric

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**gumbel**()
 Alias to: numpy.random.gumbel

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**hypergeometric**()
 Alias to: numpy.random.hypergeometric

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**laplace**()
 Alias to: numpy.random.laplace

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**logistic**()
 Alias to: numpy.random.logistic

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**lognormal**()
 Alias to: numpy.random.lognormal

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**logseries**()
 Alias to: numpy.random.logseries

  - defined by <class 'numba.core.typing.templates.Registry.register_global.
  <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**multinomial**()
 Alias to: numpy.random.multinomial

  - defined by multinomial(n, pvals, size=None) at numba/cpython/randomimpl.py:1433-1508

numpy.random.mtrand.**negative_binomial**()
 Alias to: numpy.random.negative_binomial

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**normal**()
Alias to: `numpy.random.normal`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**pareto**()
Alias to: `numpy.random.pareto`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**permutation**()
Alias to: `numpy.random.permutation`

- defined by `permutation_impl(x)` at numba/cpython/randomimpl.py:1231-1245

numpy.random.mtrand.**poisson**()
Alias to: `numpy.random.poisson`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**power**()
Alias to: `numpy.random.power`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**rand**()
Alias to: `numpy.random.rand`

- defined by `rand(*size)` at numba/cpython/randomimpl.py:1314-1326

numpy.random.mtrand.**randint**()
Alias to: `numpy.random.randint`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**randn**()
Alias to: `numpy.random.randn`

- defined by `randn(*size)` at numba/cpython/randomimpl.py:1328-1340

numpy.random.mtrand.**random**()
Alias to: `numpy.random.random`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**random_sample**()
Alias to: `numpy.random.random_sample`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**ranf**()
Alias to: `numpy.random.ranf`

- defined by `<class 'numba.core.typing.templates.Registry.register_global.<locals>.decorate.<locals>.Template'>`

numpy.random.mtrand.**rayleigh**()
> Alias to: numpy.random.rayleigh

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**sample**()
> Alias to: numpy.random.sample

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**seed**()
> Alias to: numpy.random.seed

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**shuffle**()
> Alias to: numpy.random.shuffle

> > • defined by shuffle_impl(arr) at numba/cpython/randomimpl.py:1227-1229

numpy.random.mtrand.**standard_cauchy**()
> Alias to: numpy.random.standard_cauchy

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**standard_exponential**()
> Alias to: numpy.random.standard_exponential

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**standard_gamma**()
> Alias to: numpy.random.standard_gamma

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**standard_normal**()
> Alias to: numpy.random.standard_normal

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**standard_t**()
> Alias to: numpy.random.standard_t

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**triangular**()
> Alias to: numpy.random.triangular

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**uniform**()
> Alias to: numpy.random.uniform

> > • defined by <class 'numba.core.typing.templates.Registry.register_global.
> > <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**vonmises**()
    Alias to: numpy.random.vonmises

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**wald**()
    Alias to: numpy.random.wald

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**weibull**()
    Alias to: numpy.random.weibull

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

numpy.random.mtrand.**zipf**()
    Alias to: numpy.random.zipf

        • defined by <class 'numba.core.typing.templates.Registry.register_global.
          <locals>.decorate.<locals>.Template'>

Not showing 8 unsupported functions.

supported = 42 / 50 = 84.00%

numpy.random.setup

This module is not supported.

numpy.setup

This module is not supported.

numpy.testing

This module is not supported.

numpy.testing.print_coercion_tables

This module is not supported.

numpy.testing.setup

This module is not supported.

numpy.testing.utils

This module is not supported.

---

numpy.version

This module is not supported.

7.9.2 Old style listings

Lowering Listing

This lists all lowering definition registered to the CPU target. Each subsection corresponds to a Python function that is supported by numba nopython mode. These functions have one or more lower implementation with different signatures. The compiler chooses the most specific implementation from all overloads.

`'array.argmax'`

Signature	Definition
(Array)	`array_argmax` numba/np/arraymath.py lines 705-773

`'array.argmin'`

Signature	Definition
(Array)	`array_argmin` numba/np/arraymath.py lines 632-702

`'array.argsort'`

Signature	Definition
(Array, StringLiteral)	`array_argsort` numba/np/arrayobj.py lines 5023-5037

`'array.astype'`

Signature	Definition
(Array, DTypeSpec)	`array_astype` numba/np/arrayobj.py lines 4063-4090

`'array.copy'`

Signature	Definition
(Array)	`array_copy` numba/np/arrayobj.py lines 3978-3980

`'array.cumprod'`

Signature	Definition
(Array)	`array_cumprod` numba/np/arraymath.py lines 383-399

`'array.cumsum'`

Signature	Definition
(Array)	`array_cumsum` numba/np/arraymath.py lines 363-380

`'array.flatten'`

Signature	Definition
(Array)	`array_flatten` numba/np/arrayobj.py lines 1766-1774

`'array.item'`

Signature	Definition
(Array)	`array_item` numba/np/arrayobj.py lines 521-533

`'array.itemset'`

Signature	Definition
(Array, any)	`array_itemset` numba/np/arrayobj.py lines 536-550

`'array.max'`

Signature	Definition
(Array)	`array_max` numba/np/arraymath.py lines 549-629

`'array.mean'`

Signature	Definition
(Array)	`array_mean` numba/np/arraymath.py lines 402-417

`'array.min'`

Signature	Definition
(Array)	`array_min` numba/np/arraymath.py lines 466-546

`'array.nonzero'`

Signature	Definition
(Array)	`array_nonzero` numba/np/arraymath.py lines 3112-3168

`'array.prod'`

Signature	Definition
(Array)	`array_prod` numba/np/arraymath.py lines 348-360

`'array.ravel'`

Signature	Definition
(Array)	`array_ravel` numba/np/arrayobj.py lines 1735-1755

`'array.reshape'`

Signature	Definition
*(Array, *any)*	`array_reshape_vararg` numba/np/arrayobj.py lines 1696-1699
(Array, BaseTuple)	`array_reshape` numba/np/arrayobj.py lines 1631-1693

`'array.sort'`

Signature	Definition
(Array)	`array_sort` numba/np/arrayobj.py lines 4999-5009

`'array.std'`

Signature	Definition
(Array)	`array_std` numba/np/arraymath.py lines 438-444

`'array.sum'`

Signature	Definition
(Array)	`array_sum` numba/np/arraymath.py lines 160-173
(Array, DTypeSpec)	`array_sum_dtype` numba/np/arraymath.py lines 289-302
(Array, IntegerLiteral)	`array_sum_axis` numba/np/arraymath.py lines 305-345
(Array, IntegerLiteral, DTypeSpec)	`array_sum_axis_dtype` numba/np/arraymath.py lines 246-286
(Array, int64)	`array_sum_axis` numba/np/arraymath.py lines 305-345
(Array, int64, DTypeSpec)	`array_sum_axis_dtype` numba/np/arraymath.py lines 246-286

`'array.take'`

Signature	Definition
(Array, Array)	`numpy_take_2` numba/np/arrayobj.py lines 3769-3792
(Array, BaseTuple)	`numpy_take_3` numba/np/arrayobj.py lines 3795-3815
(Array, Integer)	`numpy_take_1` numba/np/arrayobj.py lines 3756-3766
(Array, List)	`numpy_take_3` numba/np/arrayobj.py lines 3795-3815

`'array.transpose'`

Signature	Definition
(Array)	`array_transpose` numba/np/arrayobj.py lines 1449-1451
*(Array, *any)*	`array_transpose_vararg` numba/np/arrayobj.py lines 1524-1527
(Array, BaseTuple)	`array_transpose_tuple` numba/np/arrayobj.py lines 1469-1521

`'array.var'`

Signature	Definition
(Array)	`array_var` numba/np/arraymath.py lines 420-435

`'array.view'`

Signature	Definition
(Array, DTypeSpec)	`array_view` numba/np/arrayobj.py lines 1996-2021

`'complex.conjugate'`

Signature	Definition
(Complex)	`complex_conjugate_impl` numba/cpython/numbers.py lines 936-942
(Float)	`real_conjugate_impl` numba/cpython/numbers.py lines 951-952
(Integer)	`real_conjugate_impl` numba/cpython/numbers.py lines 951-952

`'ffi.from_buffer'`

Signature	Definition
(Buffer)	`from_buffer` numba/misc/cffiimpl.py lines 12-22

`'getiter'`

Signature	Definition
(Buffer)	`getiter_array` numba/np/arrayobj.py lines 255-276
(DictItemsIterable-Type)	`impl_iterable_getiter` numba/typed/dictobject.py lines 931-961
(DictKeysIterable-Type)	`impl_iterable_getiter` numba/typed/dictobject.py lines 931-961
(DictType)	`impl_dict_getiter` numba/typed/dictobject.py lines 964-995
(DictValuesIterable-Type)	`impl_iterable_getiter` numba/typed/dictobject.py lines 931-961
(IteratorType)	`iterator_getiter` numba/cpython/iterators.py lines 12-15
(List)	`getiter_list` numba/cpython/listobj.py lines 487-490
(ListType)	`impl_list_getiter` numba/typed/listobject.py lines 1393-1424
(NamedUniTuple)	`getiter_unituple` numba/cpython/tupleobj.py lines 140-156
(Set)	`getiter_set` numba/cpython/setobj.py lines 1202-1205
(UniTuple)	`getiter_unituple` numba/cpython/tupleobj.py lines 140-156
(UnicodeType)	`getiter_unicode` numba/cpython/unicode.py lines 2373-2394
(range_state_int32)	`make_range_impl.<locals>.getiter_range32_impl` numba/cpython/rangeobj.py lines 91-99
(range_state_int64)	`make_range_impl.<locals>.getiter_range32_impl` numba/cpython/rangeobj.py lines 91-99
(range_state_uint64)	`make_range_impl.<locals>.getiter_range32_impl` numba/cpython/rangeobj.py lines 91-99

`'iternext'`

Signature	Definition
(ArrayIterator)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(DictIteratorType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(EnumerateType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(Generator)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(ListIter)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(ListTypeIteratorType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(NumpyFlatType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(NumpyNdEnumerate-Type)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(NumpyNdIndexType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(NumpyNdIterType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(SetIter)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(UniTupleIter)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(UnicodeIteratorType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(ZipType)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(range_iter_int32)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(range_iter_int64)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343
(range_iter_uint64)	`iternext_impl.<locals>.outer.<locals>.wrapper` numba/core/imputils.py lines 329-343

`'list.append'`

Signature	Definition
(List, any)	`list_append` numba/cpython/listobj.py lines 838-848

`'list.clear'`

Signature	Definition
(List)	`list_clear` numba/cpython/listobj.py lines 850-855

`'list.copy'`

Signature	Definition
(List)	`list_copy` numba/cpython/listobj.py lines 857-862

`'list.count'`

Signature	Definition
(List, any)	`list_count` numba/cpython/listobj.py lines 864-874

`'list.extend'`

Signature	Definition
(List, IterableType)	`list_extend` numba/cpython/listobj.py lines 893-906

`'list.index'`

Signature	Definition
(List, any)	`list_index` numba/cpython/listobj.py lines 908-918
(List, any, Integer)	`list_index` numba/cpython/listobj.py lines 920-936
(List, any, Integer, Integer)	`list_index` numba/cpython/listobj.py lines 938-958

`'list.insert'`

Signature	Definition
(List, Integer, any)	`list_insert` numba/cpython/listobj.py lines 960-975

`'list.pop'`

Signature	Definition
(List)	`list_pop` numba/cpython/listobj.py lines 977-989
(List, Integer)	`list_pop` numba/cpython/listobj.py lines 991-1007

`'list.remove'`

Signature	Definition
(List, any)	`list_remove` numba/cpython/listobj.py lines 1009-1020

`'list.reverse'`

Signature	Definition
(List)	`list_reverse` numba/cpython/listobj.py lines 1022-1030

`'not in'`

Signature	Definition
(any, any)	`not_in` numba/cpython/builtins.py lines 371-377

`'np.random.beta'`

Signature	Definition
(Float, Float)	`betavariate_impl` numba/cpython/randomimpl.py lines 599-604
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.binomial'`

Signature	Definition
(Integer, Float)	`binomial_impl` numba/cpython/randomimpl.py lines 812-868
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.chisquare'`

Signature	Definition
(Float)	`chisquare_impl` numba/cpython/randomimpl.py lines 871-878
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.exponential'`

Signature	Definition
()	`exponential_impl` numba/cpython/randomimpl.py lines 655-666
(Float)	`exponential_impl` numba/cpython/randomimpl.py lines 643-653
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.f'`

Signature	Definition
(Float, Float)	`f_impl` numba/cpython/randomimpl.py lines 881-889
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.gamma'`

Signature	Definition
(Float)	`gammavariate_impl` numba/cpython/randomimpl.py lines 509-515
(Float, Float)	`gammavariate_impl` numba/cpython/randomimpl.py lines 509-515
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.geometric'`

Signature	Definition
(Float)	`geometric_impl` numba/cpython/randomimpl.py lines 892-915
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.gumbel'`

Signature	Definition
(Float, Float)	`gumbel_impl` numba/cpython/randomimpl.py lines 918-928
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.hypergeometric'`

Signature	Definition
(Integer, Integer, Integer)	`hypergeometric_impl` numba/cpython/randomimpl.py lines 931-954
(any, any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.laplace'`

Signature	Definition
()	`laplace_impl` numba/cpython/randomimpl.py lines 957-973
(Float)	`laplace_impl` numba/cpython/randomimpl.py lines 957-973
(Float, Float)	`laplace_impl` numba/cpython/randomimpl.py lines 957-973
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.logistic'`

Signature	Definition
()	`logistic_impl` numba/cpython/randomimpl.py lines 976-989
(Float)	`logistic_impl` numba/cpython/randomimpl.py lines 976-989
(Float, Float)	`logistic_impl` numba/cpython/randomimpl.py lines 976-989
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.lognormal'`

Signature	Definition
()	`np_lognormal_impl` numba/cpython/randomimpl.py lines 668-675
(Float)	`np_lognormal_impl` numba/cpython/randomimpl.py lines 668-675
(Float, Float)	`np_lognormal_impl` numba/cpython/randomimpl.py lines 668-675
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.logseries'`

Signature	Definition
(Float)	`logseries_impl` numba/cpython/randomimpl.py lines 991-1019
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.negative_binomial'`

Signature	Definition
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308
(int64, Float)	`negative_binomial_impl` numba/cpython/randomimpl.py lines 1022-1036

`'np.random.normal'`

Signature	Definition
()	`np_gauss_impl` numba/cpython/randomimpl.py lines 241-248
(Float)	`np_gauss_impl` numba/cpython/randomimpl.py lines 241-248
(Float, Float)	`np_gauss_impl` numba/cpython/randomimpl.py lines 241-248
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.pareto'`

Signature	Definition
(Float)	`pareto_impl` numba/cpython/randomimpl.py lines 707-717
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.poisson'`

Signature	Definition
()	`poisson_impl` numba/cpython/randomimpl.py lines 1039-1098
(Float)	`poisson_impl` numba/cpython/randomimpl.py lines 1039-1098
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.power'`

Signature	Definition
(Float)	`power_impl` numba/cpython/randomimpl.py lines 1101-1111
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.randint'`

Signature	Definition
(Integer)	`randint_impl_2` numba/cpython/randomimpl.py lines 416-422
(Integer, Integer)	`randrange_impl_2` numba/cpython/randomimpl.py lines 424-429
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.random'`

Signature	Definition
()	`random_impl` numba/cpython/randomimpl.py lines 224-231
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.random_sample'`

Signature	Definition
()	`random_impl` numba/cpython/randomimpl.py lines 224-231
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.ranf'`

Signature	Definition
()	`random_impl` numba/cpython/randomimpl.py lines 224-231
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.rayleigh'`

Signature	Definition
()	`rayleigh_impl` numba/cpython/randomimpl.py lines 1114-1126
(Float)	`rayleigh_impl` numba/cpython/randomimpl.py lines 1114-1126
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.sample'`

Signature	Definition
()	`random_impl` numba/cpython/randomimpl.py lines 224-231
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.seed'`

Signature	Definition
(uint32)	`seed_impl` numba/cpython/randomimpl.py lines 205-209

`'np.random.standard_cauchy'`

Signature	Definition
()	`cauchy_impl` numba/cpython/randomimpl.py lines 1129-1137
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.standard_exponential'`

Signature	Definition
()	`exponential_impl` numba/cpython/randomimpl.py lines 655-666
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.standard_gamma'`

Signature	Definition
(Float)	`gammavariate_impl` numba/cpython/randomimpl.py lines 509-515
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.standard_normal'`

Signature	Definition
()	`np_gauss_impl` numba/cpython/randomimpl.py lines 241-248
(any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.standard_t'`

Signature	Definition
(Float)	`standard_t_impl` numba/cpython/randomimpl.py lines 1140-1150
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.triangular'`

Signature	Definition
(Float, Float, Float)	`triangular_impl_3` numba/cpython/randomimpl.py lines 475-480
(any, any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.uniform'`

Signature	Definition
(Float, Float)	`uniform_impl` numba/cpython/randomimpl.py lines 436-439
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.vonmises'`

Signature	Definition
(Float, Float)	`vonmisesvariate_impl` numba/cpython/randomimpl.py lines 754-758
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.wald'`

Signature	Definition
(Float, Float)	`wald_impl` numba/cpython/randomimpl.py lines 1153-1172
(any, any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.weibull'`

Signature	Definition
(Float)	`weibull_impl` numba/cpython/randomimpl.py lines 735-746
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'np.random.zipf'`

Signature	Definition
(Float)	`zipf_impl` numba/cpython/randomimpl.py lines 1175-1194
(any, any)	`random_arr` numba/cpython/randomimpl.py lines 1289-1308

`'number.item'`

Signature	Definition
(Boolean)	`number_item_impl` numba/cpython/numbers.py lines 1167-1173
(Number)	`number_item_impl` numba/cpython/numbers.py lines 1167-1173

`'print_item'`

Signature	Definition
(Literal)	`print_item_impl` numba/cpython/printimpl.py lines 16-31
(any)	`print_item_impl` numba/cpython/printimpl.py lines 34-62

`'random.betavariate'`

Signature	Definition
(Float, Float)	`betavariate_impl` numba/cpython/randomimpl.py lines 592-597

`'random.expovariate'`

Signature	Definition
(Float)	`expovariate_impl` numba/cpython/randomimpl.py lines 623-641

`'random.gammavariate'`

Signature	Definition
(Float, Float)	`gammavariate_impl` numba/cpython/randomimpl.py lines 503-507

`'random.gauss'`

Signature	Definition
(Float, Float)	`gauss_impl` numba/cpython/randomimpl.py lines 234-238

`'random.getrandbits'`

Signature	Definition
(Integer)	`getrandbits_impl` numba/cpython/randomimpl.py lines 304-314

`'random.lognormvariate'`

Signature	Definition
(Float, Float)	`lognormvariate_impl` numba/cpython/randomimpl.py lines 677-681

`'random.normalvariate'`

Signature	Definition
(Float, Float)	`gauss_impl` numba/cpython/randomimpl.py lines 234-238

`'random.paretovariate'`

Signature	Definition
(Float)	`paretovariate_impl` numba/cpython/randomimpl.py lines 693-705

`'random.randint'`

Signature	Definition
(Integer, Integer)	`randint_impl_1` numba/cpython/randomimpl.py lines 408-414

`'random.random'`

Signature	Definition
()	`random_impl` numba/cpython/randomimpl.py lines 218-222

`'random.randrange'`

Signature	Definition
(Integer)	`randrange_impl_1` numba/cpython/randomimpl.py lines 386-392
(Integer, Integer)	`randrange_impl_2` numba/cpython/randomimpl.py lines 394-399
(Integer, Integer, Integer)	`randrange_impl_3` numba/cpython/randomimpl.py lines 401-406

`'random.seed'`

Signature	Definition
(uint32)	`seed_impl` numba/cpython/randomimpl.py lines 199-203

`'random.triangular'`

Signature	Definition
(Float, Float)	`triangular_impl_2` numba/cpython/randomimpl.py lines 448-466
(Float, Float, Float)	`triangular_impl_3` numba/cpython/randomimpl.py lines 468-473

`'random.uniform'`

Signature	Definition
(Float, Float)	`uniform_impl` numba/cpython/randomimpl.py lines 431-434

`'random.vonmisesvariate'`

Signature	Definition
(Float, Float)	`vonmisesvariate_impl` numba/cpython/randomimpl.py lines 748-752

`'random.weibullvariate'`

Signature	Definition
(Float, Float)	`weibullvariate_impl` numba/cpython/randomimpl.py lines 719-733

`'set.add'`

Signature	Definition
(Set, any)	`set_add` numba/cpython/setobj.py lines 1219-1225

`'set.clear'`

Signature	Definition
(Set)	`set_clear` numba/cpython/setobj.py lines 1259-1263

`'set.copy'`

Signature	Definition
(Set)	`set_copy` numba/cpython/setobj.py lines 1265-1269

`'set.difference'`

Signature	Definition
(Set, Set)	`set_difference` numba/cpython/setobj.py lines 1335-1343

`'set.difference_update'`

Signature	Definition
(Set, IterableType)	`set_difference_update` numba/cpython/setobj.py lines 1271-1278

`'set.discard'`

Signature	Definition
(Set, any)	set_discard numba/cpython/setobj.py lines 1227-1233

`'set.intersection'`

Signature	Definition
(Set, Set)	set_intersection numba/cpython/setobj.py lines 1345-1358

`'set.intersection_update'`

Signature	Definition
(Set, Set)	set_intersection_update numba/cpython/setobj.py lines 1280-1287

`'set.isdisjoint'`

Signature	Definition
(Set, Set)	set_isdisjoint numba/cpython/setobj.py lines 1394-1399

`'set.issubset'`

Signature	Definition
(Set, Set)	set_issubset numba/cpython/setobj.py lines 1401-1407

`'set.issuperset'`

Signature	Definition
(Set, Set)	set_issuperset numba/cpython/setobj.py lines 1409-1415

`'set.pop'`

Signature	Definition
(Set)	set_pop numba/cpython/setobj.py lines 1235-1243

`'set.remove'`

Signature	Definition
(Set, any)	set_remove numba/cpython/setobj.py lines 1245-1254

`'set.symmetric_difference'`

Signature	Definition
(Set, Set)	`set_symmetric_difference` numba/cpython/setobj.py lines 1360-1374

`'set.symmetric_difference_update'`

Signature	Definition
(Set, Set)	`set_symmetric_difference_update` numba/cpython/setobj.py lines 1289-1296

`'set.union'`

Signature	Definition
(Set, Set)	`set_union` numba/cpython/setobj.py lines 1376-1389

`'set.update'`

Signature	Definition
(Set, IterableType)	`set_update` numba/cpython/setobj.py lines 1298-1318

`'slice.indices'`

Signature	Definition
(SliceType, Integer)	`slice_indices` numba/cpython/slicing.py lines 213-235

`'static_getitem'`

Signature	Definition
(Array, StringLiteral)	`array_record_getitem` numba/np/arrayobj.py lines 2349-2355
(BaseTuple, IntegerLiteral)	`static_getitem_tuple` numba/cpython/tupleobj.py lines 348-373
(BaseTuple, SliceLiteral)	`static_getitem_tuple` numba/cpython/tupleobj.py lines 348-373
(EnumClass, StringLiteral)	`enum_class_getitem` numba/cpython/enumimpl.py lines 72-80
(LiteralList, IntegerLiteral)	`static_getitem_tuple` numba/cpython/tupleobj.py lines 348-373
(LiteralList, SliceLiteral)	`static_getitem_tuple` numba/cpython/tupleobj.py lines 348-373
(LiteralStrKeyDict, StringLiteral)	`static_getitem_tuple` numba/cpython/tupleobj.py lines 348-373
(Record, StringLiteral)	`record_getitem` numba/np/arrayobj.py lines 2419-2425

`'static_setitem'`

Signature	Definition
(Record, StringLiteral, any)	`record_setitem` numba/np/arrayobj.py lines 2428-2438

`'typed_getitem'`

Signature	Definition
(BaseTuple, any)	`getitem_typed` numba/cpython/tupleobj.py lines 208-290

<ufunc 'absolute'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'add'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'arccos'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'arccosh'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'arcsin'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'arcsinh'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'arctan'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'arctan2'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'arctanh'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'bitwise_and'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'bitwise_or'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'bitwise_xor'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'ceil'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'conjugate'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'copysign'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'cos'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'cosh'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'deg2rad'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'degrees'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'divmod'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'equal'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'exp'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'exp2'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'expm1'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'fabs'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'floor'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'floor_divide'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'fmax'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'fmin'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'fmod'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'gcd'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'greater'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'greater_equal'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'hypot'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'invert'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'isfinite'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'isinf'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'isnan'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'isnat'>

Signature	Definition
(NPDatetime)	`_np_isnat_impl` numba/np/npdatetime.py lines 811-814
(NPTimedelta)	`_np_isnat_impl` numba/np/npdatetime.py lines 811-814
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'lcm'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'ldexp'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'left_shift'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'less'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'less_equal'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'log'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'log10'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'log1p'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'log2'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logaddexp'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logaddexp2'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logical_and'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logical_not'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logical_or'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'logical_xor'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'maximum'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'minimum'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'multiply'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'negative'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'nextafter'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'not_equal'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'positive'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'power'>`

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

`<ufunc 'rad2deg'>`

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'radians'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'reciprocal'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'remainder'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'right_shift'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'rint'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'sign'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'signbit'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'sin'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'sinh'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'spacing'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'sqrt'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'square'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'subtract'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'tan'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'tanh'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'true_divide'>

Signature	Definition
(any, any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

<ufunc 'trunc'>

Signature	Definition
(any)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479
(any, Array)	`register_ufunc_kernel.<locals>.do_ufunc` numba/np/npyimpl.py lines 478-479

_operator.add

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_add` numba/cpython/tupleobj.py lines 31-36
(Complex, Complex)	`complex_add_impl` numba/cpython/numbers.py lines 1004-1017
(Float, Float)	`real_add_impl` numba/cpython/numbers.py lines 583-585
(Integer, Integer)	`int_add_impl` numba/cpython/numbers.py lines 36-42
(List, List)	`list_add` numba/cpython/listobj.py lines 676-696
(NPDatetime, NPTimedelta)	`datetime_plus_timedelta` numba/np/npdatetime.py lines 603-612
(NPTimedelta, NPDatetime)	`timedelta_plus_datetime` numba/np/npdatetime.py lines 615-624
(NPTimedelta, NPTimedelta)	`timedelta_add_impl` numba/np/npdatetime.py lines 180-191
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502

operator.and

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Boolean, Boolean)	`int_and_impl` numba/cpython/numbers.py lines 400-406
(Integer, Integer)	`int_and_impl` numba/cpython/numbers.py lines 400-406
(Set, Set)	`set_intersection` numba/cpython/setobj.py lines 1345-1358
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

_operator.contains

Signature	Definition
(Sequence, any)	`in_seq` numba/cpython/listobj.py lines 658-666
(Set, any)	`in_set` numba/cpython/setobj.py lines 1197-1200

`_operator.delitem`

Signature	Definition
(List, Integer)	`delitem_list_index` numba/cpython/listobj.py lines 614-620
(List, SliceType)	`delitem_list` numba/cpython/listobj.py lines 623-653

`_operator.eq`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_eq` numba/cpython/tupleobj.py lines 60-73
(Complex, Complex)	`complex_eq_impl` numba/cpython/numbers.py lines 1110-1119
(EnumMember, Enum-Member)	`enum_eq` numba/cpython/enumimpl.py lines 12-18
(Float, Float)	`real_eq_impl` numba/cpython/numbers.py lines 831-833
(Integer, Integer)	`int_eq_impl` numba/cpython/numbers.py lines 355-357
(IntegerLiteral, IntegerLiteral)	`const_eq_impl` numba/cpython/builtins.py lines 76-84
(List, List)	`list_eq` numba/cpython/listobj.py lines 757-779
(Literal, Literal)	`const_eq_impl` numba/cpython/builtins.py lines 76-84
(LiteralStrKeyDict, LiteralStrKeyDict)	`literalstrkeydict_impl_equals` numba/typed/dictobject.py lines 1169-1175
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 374-400
(Set, Set)	`set_isdisjoint` numba/cpython/setobj.py lines 1417-1422
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_eq_impl` numba/cpython/numbers.py lines 355-357

`_operator.floordiv`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Float, Float)	`real_floordiv_impl` numba/cpython/numbers.py lines 780-796
(Integer, Integer)	`int_floordiv_impl` numba/cpython/numbers.py lines 169-174
(NPTimedelta, Float)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, Integer)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, NPTimedelta)	`timedelta_floor_div_timedelta` numba/np/npdatetime.py lines 301-334
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.ge`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_ge` numba/cpython/tupleobj.py lines 95-98
(Float, Float)	`real_ge_impl` numba/cpython/numbers.py lines 826-828
(IntegerLiteral, IntegerLiteral)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(List, List)	`list_ge` numba/cpython/listobj.py lines 819-825
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_ordering_impl.<locals>.impl` numba/np/npdatetime.py lines 406-425
(Set, Set)	`set_issuperset` numba/cpython/setobj.py lines 1409-1415
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_uge_impl` numba/cpython/numbers.py lines 350-352
(int16, int16)	`int_sge_impl` numba/cpython/numbers.py lines 330-332
(int32, int32)	`int_sge_impl` numba/cpython/numbers.py lines 330-332
(int64, int64)	`int_sge_impl` numba/cpython/numbers.py lines 330-332
(int8, int8)	`int_sge_impl` numba/cpython/numbers.py lines 330-332
(uint16, uint16)	`int_uge_impl` numba/cpython/numbers.py lines 350-352
(uint32, uint32)	`int_uge_impl` numba/cpython/numbers.py lines 350-352
(uint64, uint64)	`int_uge_impl` numba/cpython/numbers.py lines 350-352
(uint8, uint8)	`int_uge_impl` numba/cpython/numbers.py lines 350-352

`_operator.getitem`

Signature	Definition
(Buffer, Array)	`fancy_getitem_array` numba/np/arrayobj.py lines 1064-1081
(Buffer, BaseTuple)	`getitem_array_tuple` numba/np/arrayobj.py lines 444-466
(Buffer, Integer)	`getitem_arraynd_intp` numba/np/arrayobj.py lines 427-441
(Buffer, SliceType)	`getitem_arraynd_intp` numba/np/arrayobj.py lines 427-441
(CPointer, Integer)	`getitem_cpointer` numba/cpython/builtins.py lines 145-150
(List, Integer)	`getitem_list` numba/cpython/listobj.py lines 507-516
(List, SliceType)	`getslice_list` numba/cpython/listobj.py lines 530-551
(NamedUniTuple, int64)	`getitem_unituple` numba/cpython/tupleobj.py lines 293-345
(NamedUniTuple, uint64)	`getitem_unituple` numba/cpython/tupleobj.py lines 293-345
(NumpyFlatType, Integer)	`iternext_numpy_getitem` numba/np/arrayobj.py lines 3187-3200
(UniTuple, int64)	`getitem_unituple` numba/cpython/tupleobj.py lines 293-345
(UniTuple, uint64)	`getitem_unituple` numba/cpython/tupleobj.py lines 293-345

`_operator.gt`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_gt` numba/cpython/tupleobj.py lines 90-93
(Float, Float)	`real_gt_impl` numba/cpython/numbers.py lines 821-823
(IntegerLiteral, IntegerLiteral)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(List, List)	`list_gt` numba/cpython/listobj.py lines 827-833
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_ordering_impl.<locals>.impl` numba/np/npdatetime.py lines 406-425
(Set, Set)	`set_gt` numba/cpython/setobj.py lines 1438-1443
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_ugt_impl` numba/cpython/numbers.py lines 345-347
(int16, int16)	`int_sgt_impl` numba/cpython/numbers.py lines 325-327
(int32, int32)	`int_sgt_impl` numba/cpython/numbers.py lines 325-327
(int64, int64)	`int_sgt_impl` numba/cpython/numbers.py lines 325-327
(int8, int8)	`int_sgt_impl` numba/cpython/numbers.py lines 325-327
(uint16, uint16)	`int_ugt_impl` numba/cpython/numbers.py lines 345-347
(uint32, uint32)	`int_ugt_impl` numba/cpython/numbers.py lines 345-347
(uint64, uint64)	`int_ugt_impl` numba/cpython/numbers.py lines 345-347
(uint8, uint8)	`int_ugt_impl` numba/cpython/numbers.py lines 345-347

`_operator.iadd`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Complex, Complex)	`complex_add_impl` numba/cpython/numbers.py lines 1004-1017
(Float, Float)	`real_add_impl` numba/cpython/numbers.py lines 583-585
(Integer, Integer)	`int_add_impl` numba/cpython/numbers.py lines 36-42
(List, List)	`list_add_inplace` numba/cpython/listobj.py lines 698-703
(NPDatetime, NPTimedelta)	`datetime_plus_timedelta` numba/np/npdatetime.py lines 603-612
(NPTimedelta, NPDatetime)	`timedelta_plus_datetime` numba/np/npdatetime.py lines 615-624
(NPTimedelta, NPTimedelta)	`timedelta_add_impl` numba/np/npdatetime.py lines 180-191
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.iand`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Boolean, Boolean)	`int_and_impl` numba/cpython/numbers.py lines 400-406
(Integer, Integer)	`int_and_impl` numba/cpython/numbers.py lines 400-406
(Set, Set)	`set_inplace` numba/cpython/setobj.py lines 1326-1330
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.ifloordiv`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Float, Float)	`real_floordiv_impl` numba/cpython/numbers.py lines 780-796
(Integer, Integer)	`int_floordiv_impl` numba/cpython/numbers.py lines 169-174
(NPTimedelta, Float)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, Integer)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.ilshift`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Integer, Integer)	`int_shl_impl` numba/cpython/numbers.py lines 379-385
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.imod`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Float, Float)	`real_mod_impl` numba/cpython/numbers.py lines 761-777
(Integer, Integer)	`int_rem_impl` numba/cpython/numbers.py lines 190-195
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.imul`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Complex, Complex)	`complex_mul_impl` numba/cpython/numbers.py lines 1036-1056
(Float, Float)	`real_mul_impl` numba/cpython/numbers.py lines 593-595
(Float, NPTimedelta)	`number_times_timedelta` numba/np/npdatetime.py lines 236-244
(Integer, Integer)	`int_mul_impl` numba/cpython/numbers.py lines 54-60
(Integer, NPTimedelta)	`number_times_timedelta` numba/np/npdatetime.py lines 236-244
(List, Integer)	`list_mul_inplace` numba/cpython/listobj.py lines 726-743
(NPTimedelta, Float)	`timedelta_times_number` numba/np/npdatetime.py lines 225-233
(NPTimedelta, Integer)	`timedelta_times_number` numba/np/npdatetime.py lines 225-233
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.invert`

Signature	Definition
(Array)	`register_unary_operator_kernel.<locals>.lower_unary_operator` numba/np/npyimpl.py lines 494-495
(Boolean)	`int_invert_impl` numba/cpython/numbers.py lines 443-449
(Integer)	`int_invert_impl` numba/cpython/numbers.py lines 443-449

`_operator.ior`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Boolean, Boolean)	`int_or_impl` numba/cpython/numbers.py lines 409-415
(Integer, Integer)	`int_or_impl` numba/cpython/numbers.py lines 409-415
(Set, Set)	`set_inplace` numba/cpython/setobj.py lines 1326-1330
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.ipow`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Complex, Complex)	`complex_power_impl` numba/cpython/numbers.py lines 960-1002
(Float, Float)	`real_power_impl` numba/cpython/numbers.py lines 799-808
(Float, IntegerLiteral)	`static_power_impl` numba/cpython/numbers.py lines 250-312
(Float, int16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Integer, Integer)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Integer, IntegerLiteral)	`static_power_impl` numba/cpython/numbers.py lines 250-312
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.irshift`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Integer, Integer)	`int_shr_impl` numba/cpython/numbers.py lines 388-397
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.is_`

Signature	Definition
(Array, Array)	`array_is` numba/np/arrayobj.py lines 2475-2486
(EnumMember, EnumMember)	`enum_is` numba/cpython/enumimpl.py lines 21-30
(List, List)	`list_is` numba/cpython/listobj.py lines 749-755
(ListType, ListType)	`list_is` numba/typed/listobject.py lines 218-224
(Opaque, Opaque)	`opaque_is` numba/cpython/builtins.py lines 60-73
(Optional, none)	`optional_is_none` numba/core/optional.py lines 18-35
(Set, Set)	`set_is` numba/cpython/setobj.py lines 1445-1451
(any, any)	`generic_is` numba/cpython/builtins.py lines 35-57
(none, Optional)	`optional_is_none` numba/core/optional.py lines 18-35
(none, none)	`always_return_true_impl` numba/core/optional.py lines 10-11

`_operator.is_not`

Signature	Definition
(any, any)	`generic_is_not` numba/cpython/builtins.py lines 26-32

`_operator.isub`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Complex, Complex)	`complex_sub_impl` numba/cpython/numbers.py lines 1020-1033
(Float, Float)	`real_sub_impl` numba/cpython/numbers.py lines 588-590
(Integer, Integer)	`int_sub_impl` numba/cpython/numbers.py lines 45-51
(NPDatetime, NPTimedelta)	`datetime_minus_timedelta` numba/np/npdatetime.py lines 629-638
(NPTimedelta, NPTimedelta)	`timedelta_sub_impl` numba/np/npdatetime.py lines 194-205
(Set, Set)	`set_inplace` numba/cpython/setobj.py lines 1326-1330
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.itruediv`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Complex, Complex)	`complex_div_impl` numba/cpython/numbers.py lines 1061-1090
(Float, Float)	`real_div_impl` numba/cpython/numbers.py lines 598-602
(Integer, Integer)	`int_truediv_impl` numba/cpython/numbers.py lines 177-187
(NPTimedelta, Float)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, Integer)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, NPTimedelta)	`timedelta_over_timedelta` numba/np/npdatetime.py lines 278-293
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.ixor`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Array, any)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510
(Boolean, Boolean)	`int_xor_impl` numba/cpython/numbers.py lines 418-424
(Integer, Integer)	`int_xor_impl` numba/cpython/numbers.py lines 418-424
(Set, Set)	`set_inplace` numba/cpython/setobj.py lines 1326-1330
(any, Array)	`register_binary_operator_kernel.<locals>.lower_inplace_operator` numba/np/npyimpl.py lines 504-510

`_operator.le`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_le` numba/cpython/tupleobj.py lines 85-88
(Float, Float)	`real_le_impl` numba/cpython/numbers.py lines 816-818
(IntegerLiteral, IntegerLiteral)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(List, List)	`list_le` numba/cpython/listobj.py lines 789-802
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_ordering_impl.<locals>.impl` numba/np/npdatetime.py lines 406-425
(Set, Set)	`set_issubset` numba/cpython/setobj.py lines 1401-1407
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_ule_impl` numba/cpython/numbers.py lines 340-342
(int16, int16)	`int_sle_impl` numba/cpython/numbers.py lines 320-322
(int32, int32)	`int_sle_impl` numba/cpython/numbers.py lines 320-322
(int64, int64)	`int_sle_impl` numba/cpython/numbers.py lines 320-322
(int8, int8)	`int_sle_impl` numba/cpython/numbers.py lines 320-322
(uint16, uint16)	`int_ule_impl` numba/cpython/numbers.py lines 340-342
(uint32, uint32)	`int_ule_impl` numba/cpython/numbers.py lines 340-342
(uint64, uint64)	`int_ule_impl` numba/cpython/numbers.py lines 340-342
(uint8, uint8)	`int_ule_impl` numba/cpython/numbers.py lines 340-342

`_operator.lshift`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Integer, Integer)	`int_shl_impl` numba/cpython/numbers.py lines 379-385
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.lt`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_lt` numba/cpython/tupleobj.py lines 80-83
(Float, Float)	`real_lt_impl` numba/cpython/numbers.py lines 811-813
(IntegerLiteral, IntegerLiteral)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(List, List)	`list_lt` numba/cpython/listobj.py lines 804-817
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_ordering_impl.<locals>.impl` numba/np/npdatetime.py lines 406-425
(Set, Set)	`set_lt` numba/cpython/setobj.py lines 1431-1436
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_ult_impl` numba/cpython/numbers.py lines 335-337
(int16, int16)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(int32, int32)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(int64, int64)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(int8, int8)	`int_slt_impl` numba/cpython/numbers.py lines 315-317
(uint16, uint16)	`int_ult_impl` numba/cpython/numbers.py lines 335-337
(uint32, uint32)	`int_ult_impl` numba/cpython/numbers.py lines 335-337
(uint64, uint64)	`int_ult_impl` numba/cpython/numbers.py lines 335-337
(uint8, uint8)	`int_ult_impl` numba/cpython/numbers.py lines 335-337

`_operator.matmul`

Signature	Definition
(Array, Array)	`dot_2` numba/np/linalg.py lines 523-542

`_operator.mod`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Float, Float)	`real_mod_impl` numba/cpython/numbers.py lines 761-777
(Integer, Integer)	`int_rem_impl` numba/cpython/numbers.py lines 190-195
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

_operator.mul

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Complex, Complex)	`complex_mul_impl` numba/cpython/numbers.py lines 1036-1056
(Float, Float)	`real_mul_impl` numba/cpython/numbers.py lines 593-595
(Float, NPTimedelta)	`number_times_timedelta` numba/np/npdatetime.py lines 236-244
(Integer, Integer)	`int_mul_impl` numba/cpython/numbers.py lines 54-60
(Integer, NPTimedelta)	`number_times_timedelta` numba/np/npdatetime.py lines 236-244
(List, Integer)	`list_mul` numba/cpython/listobj.py lines 706-724
(NPTimedelta, Float)	`timedelta_times_number` numba/np/npdatetime.py lines 225-233
(NPTimedelta, Integer)	`timedelta_times_number` numba/np/npdatetime.py lines 225-233
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

_operator.ne

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(BaseTuple, BaseTuple)	`tuple_ne` numba/cpython/tupleobj.py lines 75-78
(Complex, Complex)	`complex_ne_impl` numba/cpython/numbers.py lines 1122-1131
(EnumMember, EnumMember)	`enum_ne` numba/cpython/enumimpl.py lines 33-39
(Float, Float)	`real_ne_impl` numba/cpython/numbers.py lines 836-838
(Integer, Integer)	`int_ne_impl` numba/cpython/numbers.py lines 360-362
(List, List)	`list_ne` numba/cpython/listobj.py lines 781-787
(NPDatetime, NPDatetime)	`_create_datetime_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 663-689
(NPTimedelta, NPTimedelta)	`_create_timedelta_comparison_impl.<locals>.impl` numba/np/npdatetime.py lines 374-400
(Set, Set)	`set_ne` numba/cpython/setobj.py lines 1424-1429
(StringLiteral, StringLiteral)	`const_ne_impl` numba/cpython/builtins.py lines 87-94
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(bool, bool)	`int_ne_impl` numba/cpython/numbers.py lines 360-362

`_operator.neg`

Signature	Definition
(Array)	`register_unary_operator_kernel.<locals>.lower_unary_operator` numba/np/npyimpl.py lines 494-495
(Complex)	`complex_negate_impl` numba/cpython/numbers.py lines 1093-1102
(Float)	`real_negate_impl` numba/cpython/numbers.py lines 848-851
(Integer)	`int_negate_impl` numba/cpython/numbers.py lines 427-433
(NPTi-medelta)	`timedelta_neg_impl` numba/np/npdatetime.py lines 139-142
(bool)	`bool_negate_impl` numba/cpython/numbers.py lines 494-499

`_operator.not_`

Signature	Definition
(Complex)	`number_not_impl` numba/cpython/numbers.py lines 1179-1184
(Float)	`number_not_impl` numba/cpython/numbers.py lines 1179-1184
(Integer)	`number_not_impl` numba/cpython/numbers.py lines 1179-1184
(bool)	`number_not_impl` numba/cpython/numbers.py lines 1179-1184

`_operator.or_`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Boolean, Boolean)	`int_or_impl` numba/cpython/numbers.py lines 409-415
(Integer, Integer)	`int_or_impl` numba/cpython/numbers.py lines 409-415
(Set, Set)	`set_union` numba/cpython/setobj.py lines 1376-1389
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.pos`

Signature	Definition
(Array)	`array_positive_impl` numba/np/npyimpl.py lines 525-537
(Array)	`register_unary_operator_kernel.<locals>.lower_unary_operator` numba/np/npyimpl.py lines 494-495
(Complex)	`complex_positive_impl` numba/cpython/numbers.py lines 1105-1107
(Float)	`real_positive_impl` numba/cpython/numbers.py lines 854-858
(Integer)	`int_positive_impl` numba/cpython/numbers.py lines 436-440
(NPTi-medelta)	`timedelta_pos_impl` numba/np/npdatetime.py lines 133-136
(bool)	`bool_unary_positive_impl` numba/cpython/numbers.py lines 502-506

`_operator.pow`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Complex, Complex)	`complex_power_impl` numba/cpython/numbers.py lines 960-1002
(Float, Float)	`real_power_impl` numba/cpython/numbers.py lines 799-808
(Float, IntegerLiteral)	`static_power_impl` numba/cpython/numbers.py lines 250-312
(Float, int16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Integer, Integer)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Integer, IntegerLiteral)	`static_power_impl` numba/cpython/numbers.py lines 250-312
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.rshift`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Integer, Integer)	`int_shr_impl` numba/cpython/numbers.py lines 388-397
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.setitem`

Signature	Definition
(Buffer, any, any)	`setitem_array` numba/np/arrayobj.py lines 469-507
(CPointer, Integer, any)	`setitem_cpointer` numba/cpython/builtins.py lines 153-157
(List, Integer, any)	`setitem_list` numba/cpython/listobj.py lines 518-527
(List, SliceType, any)	`setitem_list` numba/cpython/listobj.py lines 553-610
(NumpyFlatType, Integer, any)	`iternext_numpy_getitem_any` numba/np/arrayobj.py lines 3203-3217

`_operator.sub`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Complex, Complex)	`complex_sub_impl` numba/cpython/numbers.py lines 1020-1033
(Float, Float)	`real_sub_impl` numba/cpython/numbers.py lines 588-590
(Integer, Integer)	`int_sub_impl` numba/cpython/numbers.py lines 45-51
(NPDatetime, NPDatetime)	`datetime_minus_datetime` numba/np/npdatetime.py lines 643-657
(NPDatetime, NPTimedelta)	`datetime_minus_timedelta` numba/np/npdatetime.py lines 629-638
(NPTimedelta, NPTimedelta)	`timedelta_sub_impl` numba/np/npdatetime.py lines 194-205
(Set, Set)	`set_difference` numba/cpython/setobj.py lines 1335-1343
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.truediv`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Complex, Complex)	`complex_div_impl` numba/cpython/numbers.py lines 1061-1090
(Float, Float)	`real_div_impl` numba/cpython/numbers.py lines 598-602
(Integer, Integer)	`int_truediv_impl` numba/cpython/numbers.py lines 177-187
(NPTimedelta, Float)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, Integer)	`timedelta_over_number` numba/np/npdatetime.py lines 247-275
(NPTimedelta, NPTimedelta)	`timedelta_over_timedelta` numba/np/npdatetime.py lines 278-293
(any, Array)	`register_binary_operator_kernel.<locals>.` `lower_binary_operator` numba/np/npyimpl.py lines 501-502

`_operator.xor`

Signature	Definition
(Array, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Array, any)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502
(Boolean, Boolean)	`int_xor_impl` numba/cpython/numbers.py lines 418-424
(Integer, Integer)	`int_xor_impl` numba/cpython/numbers.py lines 418-424
(Set, Set)	`set_symmetric_difference` numba/cpython/setobj.py lines 1360-1374
(any, Array)	`register_binary_operator_kernel.<locals>.lower_binary_operator` numba/np/npyimpl.py lines 501-502

`builtins.abs`

Signature	Definition
(Complex)	`complex_abs_impl` numba/cpython/numbers.py lines 1134-1142
(Float)	`real_abs_impl` numba/cpython/numbers.py lines 841-845
(NPTimedelta)	`timedelta_abs_impl` numba/np/npdatetime.py lines 145-155
(int16)	`int_abs_impl` numba/cpython/numbers.py lines 365-371
(int32)	`int_abs_impl` numba/cpython/numbers.py lines 365-371
(int64)	`int_abs_impl` numba/cpython/numbers.py lines 365-371
(int8)	`int_abs_impl` numba/cpython/numbers.py lines 365-371
(uint16)	`uint_abs_impl` numba/cpython/numbers.py lines 374-376
(uint32)	`uint_abs_impl` numba/cpython/numbers.py lines 374-376
(uint64)	`uint_abs_impl` numba/cpython/numbers.py lines 374-376
(uint8)	`uint_abs_impl` numba/cpython/numbers.py lines 374-376

`builtins.bool`

Signature	Definition
(Complex)	`complex_as_bool` numba/cpython/numbers.py lines 1202-1211
(Float)	`float_as_bool` numba/cpython/numbers.py lines 1197-1200
(Integer)	`int_as_bool` numba/cpython/numbers.py lines 1192-1195
(Sequence)	`sequence_bool` numba/cpython/listobj.py lines 668-673
(Sized)	`sized_bool` numba/cpython/builtins.py lines 390-396
(bool)	`bool_as_bool` numba/cpython/numbers.py lines 1187-1190

`builtins.complex`

Signature	Definition
*(*any)*	`complex_impl` numba/cpython/builtins.py lines 270-295

`builtins.dict`

Signature	Definition
()	`impl_dict` numba/typed/dictimpl.py lines 7-20

`builtins.divmod`

Signature	Definition
(Float, Float)	`real_divmod_impl` numba/cpython/numbers.py lines 735-758
(Integer, Integer)	`int_divmod_impl` numba/cpython/numbers.py lines 160-166

builtins.enumerate

Signature	Definition
(IterableType)	`make_enumerate_object` numba/cpython/iterators.py lines 20-44
(IterableType, Integer)	`make_enumerate_object` numba/cpython/iterators.py lines 20-44

builtins.float

Signature	Definition
(any)	`int_impl` numba/cpython/builtins.py lines 261-267

builtins.int

Signature	Definition
(any)	`int_impl` numba/cpython/builtins.py lines 261-267

builtins.iter

Signature	Definition
(IterableType)	`iter_impl` numba/cpython/builtins.py lines 348-353

builtins.len

Signature	Definition
(Buffer)	`array_len` numba/np/arrayobj.py lines 510-518
(ConstSized)	`constsized_len` numba/cpython/builtins.py lines 382-387
(List)	`list_len` numba/cpython/listobj.py lines 482-485
(NumpyFlatType)	`iternext_numpy_getitem_flat` numba/np/arrayobj.py lines 3220-3228
(Set)	`set_len` numba/cpython/setobj.py lines 1192-1195
(range_state_int32)	`make_range_impl.<locals>.range_len` numba/cpython/rangeobj.py lines 81-89
(range_state_int64)	`make_range_impl.<locals>.range_len` numba/cpython/rangeobj.py lines 81-89
(range_state_uint64)	`make_range_impl.<locals>.range_len` numba/cpython/rangeobj.py lines 81-89

builtins.list

Signature	Definition
()	`list_constructor` numba/cpython/listobj.py lines 472-477
(IterableType)	`list_constructor` numba/cpython/listobj.py lines 462-470

`builtins.max`

Signature	Definition
*(*any)*	`max_vararg` numba/cpython/builtins.py lines 192-194
(BaseTuple)	`max_iterable` numba/cpython/builtins.py lines 186-190

`builtins.min`

Signature	Definition
*(*any)*	`min_vararg` numba/cpython/builtins.py lines 202-204
(BaseTuple)	`min_iterable` numba/cpython/builtins.py lines 196-200

`builtins.next`

Signature	Definition
(IteratorType)	`next_impl` numba/cpython/builtins.py lines 356-366

`builtins.pow`

Signature	Definition
(Complex, Complex)	`complex_power_impl` numba/cpython/numbers.py lines 960-1002
(Float, Float)	`real_power_impl` numba/cpython/numbers.py lines 799-808
(Float, int16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, int8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint16)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint32)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint64)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Float, uint8)	`int_power_impl` numba/cpython/numbers.py lines 207-247
(Integer, Integer)	`int_power_impl` numba/cpython/numbers.py lines 207-247

`builtins.print`

Signature	Definition
*(*any)*	`print_varargs_impl` numba/cpython/printimpl.py lines 65-83

builtins.range

Signature	Definition
(int32)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int32, int32)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int32, int32, int32)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(int64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int64, int64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int64, int64, int64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(uint64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(uint64, uint64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(uint64, uint64, uint64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79

builtins.round

Signature	Definition
(Float)	`round_impl_unary` numba/cpython/builtins.py lines 211-221
(Float, Integer)	`round_impl_binary` numba/cpython/builtins.py lines 223-255

builtins.set

Signature	Definition
()	`set_empty_constructor` numba/cpython/setobj.py lines 1167-1171
(IterableType)	`set_constructor` numba/cpython/setobj.py lines 1173-1186

builtins.slice

Signature	Definition
*(*any)*	`slice_constructor_impl` numba/cpython/slicing.py lines 151-191

builtins.tuple

Signature	Definition
()	`lower_empty_tuple` numba/cpython/builtins.py lines 398-402
(BaseTuple)	`lower_tuple` numba/cpython/builtins.py lines 404-407

`builtins.type`

Signature	Definition
(any)	`type_impl` numba/cpython/builtins.py lines 340-345

`builtins.zip`

Signature	Definition
*(*any)*	`make_zip_object` numba/cpython/iterators.py lines 75-87

`cmath.acos`

Signature	Definition
(Complex)	`acos_impl` numba/cpython/cmathimpl.py lines 379-403

`cmath.acosh`

Signature	Definition
(Complex)	`acosh_impl` numba/cpython/cmathimpl.py lines 405-429

`cmath.asin`

Signature	Definition
(Complex)	`asin_impl` numba/cpython/cmathimpl.py lines 455-463

`cmath.asinh`

Signature	Definition
(Complex)	`asinh_impl` numba/cpython/cmathimpl.py lines 431-453

`cmath.atan`

Signature	Definition
(Complex)	`atan_impl` numba/cpython/cmathimpl.py lines 465-477

`cmath.atanh`

Signature	Definition
(Complex)	`atanh_impl` numba/cpython/cmathimpl.py lines 479-531

cmath.cos

Signature	Definition
(Complex)	`cos_impl` numba/cpython/cmathimpl.py lines 267-274

cmath.cosh

Signature	Definition
(Complex)	`cosh_impl` numba/cpython/cmathimpl.py lines 276-302

cmath.exp

Signature	Definition
(Complex)	`intrinsic_complex_unary.<locals>.wrapper` numba/cpython/cmathimpl.py lines 94-108

cmath.isfinite

Signature	Definition
(Complex)	`isfinite_float_impl` numba/cpython/cmathimpl.py lines 50-56

cmath.isinf

Signature	Definition
(Complex)	`isinf_float_impl` numba/cpython/cmathimpl.py lines 41-47

cmath.isnan

Signature	Definition
(Complex)	`isnan_float_impl` numba/cpython/cmathimpl.py lines 33-39

cmath.log

Signature	Definition
(Complex)	`intrinsic_complex_unary.<locals>.wrapper` numba/cpython/cmathimpl.py lines 94-108
(Complex, Complex)	`log_base_impl` numba/cpython/cmathimpl.py lines 166-175

cmath.log10

Signature	Definition
(Complex)	`log10_impl` numba/cpython/cmathimpl.py lines 178-190

cmath.phase

Signature	Definition
(Complex)	`intrinsic_complex_unary.<locals>.wrapper` numba/cpython/cmathimpl.py lines 94-108

cmath.polar

Signature	Definition
(Complex)	`intrinsic_complex_unary.<locals>.wrapper` numba/cpython/cmathimpl.py lines 94-108

cmath.rect

Signature	Definition
(Float, Float)	`rect_impl` numba/cpython/cmathimpl.py lines 59-91

cmath.sin

Signature	Definition
(Complex)	`sin_impl` numba/cpython/cmathimpl.py lines 305-313

cmath.sinh

Signature	Definition
(Complex)	`sinh_impl` numba/cpython/cmathimpl.py lines 315-338

cmath.sqrt

Signature	Definition
(Complex)	`sqrt_impl` numba/cpython/cmathimpl.py lines 206-264

cmath.tan

Signature	Definition
(Complex)	`tan_impl` numba/cpython/cmathimpl.py lines 341-349

cmath.tanh

Signature	Definition
(Complex)	`tanh_impl` numba/cpython/cmathimpl.py lines 351-376

math.acos

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.acosh

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.asin

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.asinh

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.atan

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.atan2

Signature	Definition
(Float, Float)	`atan2_float_impl` numba/cpython/mathimpl.py lines 313-326
(int64, int64)	`atan2_s64_impl` numba/cpython/mathimpl.py lines 297-303
(uint64, uint64)	`atan2_u64_impl` numba/cpython/mathimpl.py lines 305-311

math.atanh

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.ceil

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.copysign

Signature	Definition
(Float, Float)	`copysign_float_impl` numba/cpython/mathimpl.py lines 250-257

math.cos

Signature	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.cosh

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.degrees

Signature	Definition
(Float)	`degrees_float_impl` numba/cpython/mathimpl.py lines 394-399
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.erf

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.erfc

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.exp

Signature	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.expm1

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.fabs

Signature	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.floor

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.frexp

Signature	Definition
(Float)	`frexp_impl` numba/cpython/mathimpl.py lines 263-277

math.gamma

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.gcd

Signature	Definition
(Integer, Integer)	`gcd_impl` numba/cpython/mathimpl.py lines 427-454

math.hypot

Signature	Definition
(Float, Float)	`hypot_float_impl` numba/cpython/mathimpl.py lines 352-378
(int64, int64)	`hypot_s64_impl` numba/cpython/mathimpl.py lines 332-339
(uint64, uint64)	`hypot_u64_impl` numba/cpython/mathimpl.py lines 342-349

math.isfinite

Signature	Definition
(Float)	`isfinite_float_impl` numba/cpython/mathimpl.py lines 237-241
(Integer)	`isfinite_int_impl` numba/cpython/mathimpl.py lines 244-247

math.isinf

Signature	Definition
(Float)	`isinf_float_impl` numba/cpython/mathimpl.py lines 225-229
(Integer)	`isinf_int_impl` numba/cpython/mathimpl.py lines 231-234

math.isnan

Signature	Definition
(Float)	`isnan_float_impl` numba/cpython/mathimpl.py lines 213-217
(Integer)	`isnan_int_impl` numba/cpython/mathimpl.py lines 219-222

math.ldexp

Signature	Definition
(Float, int32)	`ldexp_impl` numba/cpython/mathimpl.py lines 280-291

math.lgamma

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.log

Signa-ture	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.log10

Signa-ture	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.log1p

Signa-ture	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.log2

Signa-ture	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.pow

Signature	Definition
(Float, Float)	`pow_impl` numba/cpython/mathimpl.py lines 405-409
(Float, Integer)	`pow_impl` numba/cpython/mathimpl.py lines 405-409

`math.radians`

Signature	Definition
(Float)	`radians_float_impl` numba/cpython/mathimpl.py lines 383-388
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

`math.sin`

Signature	Definition
(Float)	`unary_math_intr.<locals>.float_impl` numba/cpython/mathimpl.py lines 131-134
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

`math.sinh`

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

`math.sqrt`

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

`math.tan`

Signature	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.tanh

Signa-ture	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

math.trunc

Signa-ture	Definition
(Float)	`unary_math_extern.<locals>.float_impl` numba/cpython/mathimpl.py lines 149-165
(Integer)	`_unary_int_input_wrapper_impl.<locals>.implementer` numba/cpython/mathimpl.py lines 113-119

numba.core.types.abstract.TypeRef

Signature	Definition
*(*any)*	`redirect_type_ctor` numba/cpython/builtins.py lines 553-582

numba.core.types.functions.NamedTupleClass

Signature	Definition
*(*any)*	`namedtuple_constructor` numba/cpython/tupleobj.py lines 17-29

numba.core.types.functions.NumberClass

Signature	Definition
(any)	`number_constructor` numba/cpython/builtins.py lines 298-311

numba.core.typing.builtins.IndexValue

Signature	Definition
(int64, Type)	`impl_index_value` numba/cpython/builtins.py lines 493-501
(uint64, Type)	`impl_index_value` numba/cpython/builtins.py lines 493-501

numba.cpython.builtins.get_type_max_value

Signature	Definition
(DType)	`lower_get_type_max_value` numba/cpython/builtins.py lines 467-486
(NumberClass)	`lower_get_type_max_value` numba/cpython/builtins.py lines 467-486

numba.cpython.builtins.get_type_min_value

Signature	Definition
(DType)	`lower_get_type_min_value` numba/cpython/builtins.py lines 446-465
(NumberClass)	`lower_get_type_min_value` numba/cpython/builtins.py lines 446-465

numba.misc.special.pndindex

Signature	Definition
(*<class 'numba.core.types.scalars.Integer'>)	`make_array_ndindex` numba/np/arrayobj.py lines 3265-3277
(BaseTuple)	`make_array_ndindex_tuple` numba/np/arrayobj.py lines 3280-3300

numba.misc.special.prange

Signature	Definition
(int32)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int32, int32)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int32, int32, int32)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(int64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int64, int64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int64, int64, int64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(uint64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(uint64, uint64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(uint64, uint64, uint64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79

numba.np.arrayobj.reshape_unchecked

Signature	Definition
(Array, BaseTuple, BaseTuple)	`impl_shape_unchecked` numba/np/arrayobj.py lines 5078-5097

`numba.np.numpy_support.carray`

Signature	Definition
(any, any)	`np_cfarray` numba/np/arrayobj.py lines 4133-4186
(any, any, DTypeSpec)	`np_cfarray` numba/np/arrayobj.py lines 4133-4186

`numba.np.numpy_support.farray`

Signature	Definition
(any, any)	`np_cfarray` numba/np/arrayobj.py lines 4133-4186
(any, any, DTypeSpec)	`np_cfarray` numba/np/arrayobj.py lines 4133-4186

`numba.parfors.parfor.internal_prange`

Signature	Definition
(int32)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int32, int32)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int32, int32, int32)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(int64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(int64, int64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(int64, int64, int64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79
(uint64)	`make_range_impl.<locals>.range1_impl` numba/cpython/rangeobj.py lines 30-45
(uint64, uint64)	`make_range_impl.<locals>.range2_impl` numba/cpython/rangeobj.py lines 47-62
(uint64, uint64, uint64)	`make_range_impl.<locals>.range3_impl` numba/cpython/rangeobj.py lines 64-79

`numba.stencils.stencil.stencil`

Signature	Definition
()	`stencil_dummy_lower` numba/stencils/stencil.py lines 800-803

`numpy.amax`

Signature	Definition
(Array)	`array_max` numba/np/arraymath.py lines 549-629

numpy.amin

Signature	Definition
(Array)	`array_min` numba/np/arraymath.py lines 466-546

numpy.angle

Signature	Definition
(Array)	`array_angle_kwarg` numba/np/arraymath.py lines 3093-3109
(Array, Boolean)	`array_angle_kwarg` numba/np/arraymath.py lines 3093-3109
(Number)	`scalar_angle_kwarg` numba/np/arraymath.py lines 3074-3090
(Number, Boolean)	`scalar_angle_kwarg` numba/np/arraymath.py lines 3074-3090

numpy.argmax

Signature	Definition
(Array)	`array_argmax` numba/np/arraymath.py lines 705-773

numpy.argmin

Signature	Definition
(Array)	`array_argmin` numba/np/arraymath.py lines 632-702

numpy.argsort

Signature	Definition
(Array, StringLiteral)	`array_argsort` numba/np/arrayobj.py lines 5023-5037

numpy.array

Signature	Definition
(any)	`np_array` numba/np/arrayobj.py lines 4328-4344
(any, DTypeSpec)	`np_array` numba/np/arrayobj.py lines 4328-4344

numpy.ascontiguousarray

Signature	Definition
(Array)	`array_ascontiguousarray` numba/np/arrayobj.py lines 4058-4060

numpy.asfortranarray

Signature	Definition
(Array)	`array_asfortranarray` numba/np/arrayobj.py lines 4053-4055

numpy.atleast_1d

Signature	Definition
(*<class 'numba.core.types.npytypes.Array'>)	`np_atleast_1d` numba/np/arrayobj.py lines 4505-4509

numpy.atleast_2d

Signature	Definition
(*<class 'numba.core.types.npytypes.Array'>)	`np_atleast_2d` numba/np/arrayobj.py lines 4512-4516

numpy.atleast_3d

Signature	Definition
(*<class 'numba.core.types.npytypes.Array'>)	`np_atleast_3d` numba/np/arrayobj.py lines 4519-4523

numpy.column_stack

Signature	Definition
(BaseTuple)	`np_column_stack` numba/np/arrayobj.py lines 4746-4771

numpy.concatenate

Signature	Definition
(BaseTuple)	`np_concatenate` numba/np/arrayobj.py lines 4726-4733
(BaseTuple, Integer)	`np_concatenate_axis` numba/np/arrayobj.py lines 4736-4743

numpy.copy

Signature	Definition
(Array)	`numpy_copy` numba/np/arrayobj.py lines 3983-3985

numpy.cumprod

Signature	Definition
(Array)	`array_cumprod` numba/np/arraymath.py lines 383-399

numpy.cumsum

Signature	Definition
(Array)	`array_cumsum` numba/np/arraymath.py lines 363-380

numpy.diag

Signature	Definition
(Array)	`numpy_diag` numba/np/arrayobj.py lines 3707-3711
(Array, Integer)	`numpy_diag_kwarg` numba/np/arrayobj.py lines 3714-3753

numpy.dot

Signature	Definition
(Array, Array)	`dot_2` numba/np/linalg.py lines 523-542
(Array, Array, Array)	`dot_3` numba/np/linalg.py lines 727-742

numpy.dstack

Signature	Definition
(BaseTuple)	`np_dstack` numba/np/arrayobj.py lines 4838-4870

numpy.empty

Signature	Definition
(any)	`numpy_empty_nd` numba/np/arrayobj.py lines 3500-3505
(any, any)	`numpy_empty_nd` numba/np/arrayobj.py lines 3500-3505

numpy.empty_like

Signature	Definition
(any)	`numpy_empty_like_nd` numba/np/arrayobj.py lines 3508-3513
(any, DTypeSpec)	`numpy_empty_like_nd` numba/np/arrayobj.py lines 3508-3513

`numpy.expand_dims`

Signature	Definition
(Array, Integer)	`np_expand_dims` numba/np/arrayobj.py lines 4452-4459

`numpy.frombuffer`

Signature	Definition
(Buffer)	`np_frombuffer` numba/np/arrayobj.py lines 4093-4130
(Buffer, DTypeSpec)	`np_frombuffer` numba/np/arrayobj.py lines 4093-4130

`numpy.full`

Signature	Definition
(any, any)	`numpy_full_nd` numba/np/arrayobj.py lines 3534-3544
(any, any, DTypeSpec)	`numpy_full_dtype_nd` numba/np/arrayobj.py lines 3547-3557

`numpy.full_like`

Signature	Definition
(any, any)	`numpy_full_like_nd` numba/np/arrayobj.py lines 3560-3570
(any, any, DTypeSpec)	`numpy_full_like_nd_type_spec` numba/np/arrayobj.py lines 3573-3583

`numpy.hstack`

Signature	Definition
(BaseTuple)	`np_hstack` numba/np/arrayobj.py lines 4797-4814

`numpy.identity`

Signature	Definition
(Integer)	`numpy_identity` numba/np/arrayobj.py lines 3640-3650
(Integer, DTypeSpec)	`numpy_identity_type_spec` numba/np/arrayobj.py lines 3653-3663

`numpy.linspace`

Signature	Definition
(Number, Number)	`numpy_linspace_2` numba/np/arrayobj.py lines 3911-3918
(Number, Number, Integer)	`numpy_linspace_3` numba/np/arrayobj.py lines 3921-3938

numpy.mean

Signature	Definition
(Array)	`array_mean` numba/np/arraymath.py lines 402-417

numpy.ndenumerate

Signature	Definition
(Array)	`make_array_ndenumerate` numba/np/arrayobj.py lines 3231-3246

numpy.ndindex

Signature	Definition
(*<class 'numba.core.types.scalars.Integer'>)	`make_array_ndindex` numba/np/arrayobj.py lines 3265-3277
(BaseTuple)	`make_array_ndindex_tuple` numba/np/arrayobj.py lines 3280-3300

numpy.nditer

Signature	Definition
(any)	`make_array_nditer` numba/np/arrayobj.py lines 3315-3332

numpy.nonzero

Signature	Definition
(Array)	`array_nonzero` numba/np/arraymath.py lines 3112-3168

numpy.ones

Signature	Definition
(any)	`numpy_ones_nd` numba/np/arrayobj.py lines 3586-3598
(any, DTypeSpec)	`numpy_ones_dtype_nd` numba/np/arrayobj.py lines 3601-3611

numpy.ones_like

Signature	Definition
(any)	`numpy_ones_like_nd` numba/np/arrayobj.py lines 3614-3624
(any, DTypeSpec)	`numpy_ones_like_dtype_nd` numba/np/arrayobj.py lines 3627-3637

numpy.prod

Signature	Definition
(Array)	`array_prod` numba/np/arraymath.py lines 348-360

numpy.ravel

Signature	Definition
(Array)	`np_ravel` numba/np/arrayobj.py lines 1758-1763

numpy.round_

Signature	Definition
(Array, Integer, Array)	`array_round` numba/np/arraymath.py lines 3036-3046
(Complex)	`scalar_round_unary_complex` numba/np/arraymath.py lines 2984-2991
(Complex, Integer)	`scalar_round_binary_complex` numba/np/arraymath.py lines 3026-3033
(Float)	`scalar_round_unary_float` numba/np/arraymath.py lines 2972-2975
(Float, Integer)	`scalar_round_binary_float` numba/np/arraymath.py lines 2994-3023
(Integer)	`scalar_round_unary_integer` numba/np/arraymath.py lines 2978-2981
(Integer, Integer)	`scalar_round_binary_float` numba/np/arraymath.py lines 2994-3023

numpy.sinc

Signature	Definition
(Array)	`array_sinc` numba/np/arraymath.py lines 3049-3057
(Number)	`scalar_sinc` numba/np/arraymath.py lines 3060-3071

numpy.sort

Signature	Definition
(Array)	`np_sort` numba/np/arrayobj.py lines 5012-5020

numpy.stack

Signature	Definition
(BaseTuple)	`np_stack` numba/np/arrayobj.py lines 4785-4788
(BaseTuple, Integer)	`np_stack_axis` numba/np/arrayobj.py lines 4791-4794

`numpy.std`

Signature	Definition
(Array)	`array_std` numba/np/arraymath.py lines 438-444

`numpy.sum`

Signature	Definition
(Array)	`array_sum` numba/np/arraymath.py lines 160-173
(Array, DTypeSpec)	`array_sum_dtype` numba/np/arraymath.py lines 289-302
(Array, IntegerLiteral)	`array_sum_axis` numba/np/arraymath.py lines 305-345
(Array, IntegerLiteral, DTypeSpec)	`array_sum_axis_dtype` numba/np/arraymath.py lines 246-286
(Array, int64)	`array_sum_axis` numba/np/arraymath.py lines 305-345
(Array, int64, DTypeSpec)	`array_sum_axis_dtype` numba/np/arraymath.py lines 246-286

`numpy.take`

Signature	Definition
(Array, Array)	`numpy_take_2` numba/np/arrayobj.py lines 3769-3792
(Array, BaseTuple)	`numpy_take_3` numba/np/arrayobj.py lines 3795-3815
(Array, Integer)	`numpy_take_1` numba/np/arrayobj.py lines 3756-3766
(Array, List)	`numpy_take_3` numba/np/arrayobj.py lines 3795-3815

`numpy.var`

Signature	Definition
(Array)	`array_var` numba/np/arraymath.py lines 420-435

`numpy.vdot`

Signature	Definition
(Array, Array)	`vdot` numba/np/linalg.py lines 548-556

`numpy.vstack`

Signature	Definition
(BaseTuple)	`np_vstack` numba/np/arrayobj.py lines 4817-4835

`numpy.where`

Signature	Definition
(Array)	`array_nonzero` [numba/np/arraymath.py lines 3112-3168](#)
(any, any, any)	`any_where` [numba/np/arraymath.py lines 3254-3284](#)

`numpy.zeros`

Signature	Definition
(any)	`numpy_zeros_nd` [numba/np/arrayobj.py lines 3516-3522](#)
(any, any)	`numpy_zeros_nd` [numba/np/arrayobj.py lines 3516-3522](#)

`numpy.zeros_like`

Signature	Definition
(any)	`numpy_zeros_like_nd` [numba/np/arrayobj.py lines 3525-3531](#)
(any, DTypeSpec)	`numpy_zeros_like_nd` [numba/np/arrayobj.py lines 3525-3531](#)

7.10 Notes on stencils

Numba provides the *@stencil decorator* to represent stencil computations. This document explains how this feature is implemented in the several different modes available in Numba. Currently, calls to the stencil from non-jitted code is supported as well as calls from jitted code, either with or without the *parallel=True* option.

7.10.1 The stencil decorator

The stencil decorator itself just returns a `StencilFunc` object. This object encapsulates the original stencil kernel function as specified in the program and the options passed to the stencil decorator. Also of note is that after the first compilation of the stencil, the computed neighborhood of the stencil is stored in the `StencilFunc` object in the `neighborhood` attribute.

7.10.2 Handling the three modes

As mentioned above, Numba supports the calling of stencils from inside or outside a `@jit` compiled function, with or without the *parallel=True* option.

Outside jit context

`StencilFunc` overrides the `__call__` method so that calls to `StencilFunc` objects execute the stencil:

```python
def __call__(self, *args, **kwargs):
    result = kwargs.get('out')

    new_stencil_func = self._stencil_wrapper(result, None, *args)
```

(continues on next page)

```python
    if result is None:
        return new_stencil_func.entry_point(*args)
    else:
        return new_stencil_func.entry_point(*args, result)
```

First, the presence of the optional *out* parameter is checked. If it is present then the output array is stored in `result`. Then, the call to `_stencil_wrapper` generates the stencil function given the result and argument types and finally the generated stencil function is executed and its result returned.

Jit without `parallel=True`

When constructed, a `StencilFunc` inserts itself into the typing context's set of user functions and provides the `_type_me` callback. In this way, the standard Numba compiler is able to determine the output type and signature of a `StencilFunc`. Each `StencilFunc` maintains a cache of previously seen combinations of input argument types and keyword types. If previously seen, the `StencilFunc` returns the computed signature. If not previously computed, the `StencilFunc` computes the return type of the stencil by running the Numba compiler frontend on the stencil kernel and then performing type inference on the *Numba IR* (IR) to get the scalar return type of the kernel. From that, a Numpy array type is constructed whose element type matches that scalar return type.

After computing the signature of the stencil for a previously unseen combination of input and keyword types, the `StencilFunc` then *creates the stencil function* itself. `StencilFunc` then installs the new stencil function's definition in the target context so that jitted code is able to call it.

Thus, in this mode, the generated stencil function is a stand-alone function called like a normal function from within jitted code.

Jit with `parallel=True`

When calling a `StencilFunc` from a jitted context with `parallel=True`, a separate stencil function as generated by *Creating the stencil function* is not used. Instead, *parfors* (*Stage 6b: Perform Automatic Parallelization*) are created within the current function that implements the stencil. This code again starts with the stencil kernel and does a similar kernel size computation but then rather than standard Python looping syntax, corresponding *parfors* are created so that the execution of the stencil will take place in parallel.

The stencil to *parfor* translations can also be selectively disabled by setting `parallel={'stencil': False}`, among other sub-options described in *Stage 6b: Perform Automatic Parallelization*.

7.10.3 Creating the stencil function

Conceptually, a stencil function is created from the user-specified stencil kernel by adding looping code around the kernel, transforming the relative kernel indices into absolute array indices based on the loop indices, and replacing the kernel's `return` statement with a statement to assign the computed value into the output array.

To accomplish this transformation, first, a copy of the stencil kernel IR is created so that subsequent modifications of the IR for different stencil signatures will not effect each other.

Then, an approach similar to how GUFunc's are created for *parfors* is employed. In a text buffer, a Python function is created with a unique name. The input array parameter is added to the function definition and if the `out` argument type is present then an `out` parameter is added to the stencil function definition. If the `out` argument is not present then first an output array is created with `numpy.zeros` having the same shape as the input array.

The kernel is then analyzed to compute the stencil size and the shape of the boundary (or the `neighborhood` stencil decorator argument is used for this purpose if present). Then, one `for` loop for each dimension of the input array

---

is added to the stencil function definition. The range of each loop is controlled by the stencil kernel size previously computed so that the boundary of the output image is not modified but instead left as is. The body of the innermost `for` loop is a single `sentinel` statement that is easily recognized in the IR. A call to `exec` with the text buffer is used to force the stencil function into existence and an `eval` is used to get access to the corresponding function on which `run_frontend` is used to get the stencil function IR.

Various renaming and relabeling is performed on the stencil function IR and the kernel IR so that the two can be combined without conflict. The relative indices in the kernel IR (i.e., `getitem` calls) are replaced with expressions where the corresponding loop index variables are added to the relative indices. The `return` statement in the kernel IR is replaced with a `setitem` for the corresponding element in the output array. The stencil function IR is then scanned for the sentinel and the sentinel replaced with the modified kernel IR.

Next, `compile_ir` is used to compile the combined stencil function IR. The resulting compile result is cached in the `StencilFunc` so that other calls to the same stencil do not need to undertake this process again.

7.10.4 Exceptions raised

Various checks are performed during stencil compilation to make sure that user-specified options do not conflict with each other or with other runtime parameters. For example, if the user has manually specified a `neighborhood` to the stencil decorator, the length of that neighborhood must match the dimensionality of the input array. If this is not the case, a `ValueError` is raised.

If the neighborhood has not been specified then it must be inferred and a requirement to infer the kernel is that all indices are constant integers. If they are not, a `ValueError` is raised indicating that kernel indices may not be non-constant.

Finally, the stencil implementation detects the output array type by running Numba type inference on the stencil kernel. If the return type of this kernel does not match the type of the value passed to the `cval` stencil decorator option then a `ValueError` is raised.

7.11 Customizing the Compiler

> **Warning:** The custom pipeline feature is for expert use only. Modifying the compiler behavior can invalidate internal assumptions in the numba source code.

For library developers looking for a way to extend or modify the compiler behavior, you can do so by defining a custom compiler by inheriting from `numba.compiler.CompilerBase`. The default Numba compiler is defined as `numba.compiler.Compiler`, implementing the `.define_pipelines()` method, which adds the *nopython-mode*, *object-mode* and *interpreted-mode* pipelines. For convenience these three pipelines are defined in `numba.compiler.DefaultPassBuilder` by the methods:

- `.define_nopython_pipeline()`
- `.define_objectmode_pipeline()`
- `.define_interpreted_pipeline()`

respectively.

To use a custom subclass of `CompilerBase`, supply it as the `pipeline_class` keyword argument to the `@jit` and `@generated_jit` decorators. By doing so, the effect of the custom pipeline is limited to the function being decorated.

7.11.1 Implementing a compiler pass

Numba makes it possible to implement a new compiler pass and does so through the use of an API similar to that of LLVM. The following demonstrates the basic process involved.

Compiler pass classes

All passes must inherit from `numba.compiler_machinery.CompilerPass`, commonly used subclasses are:

- `numba.compiler_machinery.FunctionPass` for describing a pass that operates on a function-at-once level and may mutate the IR state.

- `numba.compiler_machinery.AnalysisPass` for describing a pass that performs analysis only.

- `numba.compiler_machinery.LoweringPass` for describing a pass that performs lowering only.

In this example a new compiler pass will be implemented that will rewrite all `ir.Const(x)` nodes, where `x` is a subclass of `numbers.Number`, such that the value of `x` is incremented by one. There is no use for this pass other than to serve as a pedagogical vehicle!

The `numba.compiler_machinery.FunctionPass` is appropriate for the suggested pass behavior and so is the base class of the new pass. Further, a `run_pass` method is defined to do the work (this method is abstract, all compiler passes must implement it).

First the new class:

```python
from numba import njit
from numba.core import ir
from numba.core.compiler import CompilerBase, DefaultPassBuilder
from numba.core.compiler_machinery import FunctionPass, register_pass
from numba.core.untyped_passes import IRProcessing
from numbers import Number


# Register this pass with the compiler framework, declare that it will not
# mutate the control flow graph and that it is not an analysis_only pass (it
# potentially mutates the IR).
@register_pass(mutates_CFG=False, analysis_only=False)
class ConstsAddOne(FunctionPass):
    _name = "consts_add_one" # the common name for the pass

    def __init__(self):
        FunctionPass.__init__(self)

    # implement method to do the work, "state" is the internal compiler
    # state from the CompilerBase instance.
    def run_pass(self, state):
        func_ir = state.func_ir # get the FunctionIR object
        mutated = False # used to record whether this pass mutates the IR
        # walk the blocks
        for blk in func_ir.blocks.values():
            # find the assignment nodes in the block and walk them
            for assgn in blk.find_insts(ir.Assign):
                # if an assignment value is a ir.Consts
                if isinstance(assgn.value, ir.Const):
                    const_val = assgn.value
                    # if the value of the ir.Const is a Number
                    if isinstance(const_val.value, Number):
                        # then add one!
```

(continues on next page)

```
                           const_val.value += 1
                           mutated |= True
        return mutated # return True if the IR was mutated, False if not.
```

Note also that the class must be registered with Numba's compiler machinery using `@register_pass`. This in part is to allow the declaration of whether the pass mutates the control flow graph and whether it is an analysis only pass.

Next, define a new compiler based on the existing `numba.compiler.CompilerBase`. The compiler pipeline is defined through the use of an existing pipeline and the new pass declared above is added to be run after the `IRProcessing` pass.

```
class MyCompiler(CompilerBase): # custom compiler extends from CompilerBase

    def define_pipelines(self):
        # define a new set of pipelines (just one in this case) and for ease
        # base it on an existing pipeline from the DefaultPassBuilder,
        # namely the "nopython" pipeline
        pm = DefaultPassBuilder.define_nopython_pipeline(self.state)
        # Add the new pass to run after IRProcessing
        pm.add_pass_after(ConstsAddOne, IRProcessing)
        # finalize
        pm.finalize()
        # return as an iterable, any number of pipelines may be defined!
        return [pm]
```

Finally update the `@njit` decorator at the call site to make use of the newly defined compilation pipeline.

```
@njit(pipeline_class=MyCompiler) # JIT compile using the custom compiler
def foo(x):
    a = 10
    b = 20.2
    c = x + a + b
    return c

print(foo(100)) # 100 + 10 + 20.2 (+ 1 + 1), extra + 1 + 1 from the rewrite!
```

7.11.2 Debugging compiler passes

Observing IR Changes

It is often useful to be able to see the changes a pass makes to the IR. Numba conveniently permits this through the use of the environment variable *NUMBA_DEBUG_PRINT_AFTER*. In the case of the above pass, running the example code with `NUMBA_DEBUG_PRINT_AFTER="ir_processing,consts_add_one"` gives:

```
----------------------------nopython: ir_processing----------------------------
label 0:
    x = arg(0, name=x)                      ['x']
    $const0.1 = const(int, 10)              ['$const0.1']
    a = $const0.1                           ['$const0.1', 'a']
    del $const0.1                           []
    $const0.2 = const(float, 20.2)          ['$const0.2']
    b = $const0.2                           ['$const0.2', 'b']
    del $const0.2                           []
    $0.5 = x + a                            ['$0.5', 'a', 'x']
```

```
    del x                                     []
    del a                                     []
    $0.7 = $0.5 + b                           ['$0.5', '$0.7', 'b']
    del b                                     []
    del $0.5                                  []
    c = $0.7                                  ['$0.7', 'c']
    del $0.7                                  []
    $0.9 = cast(value=c)                      ['$0.9', 'c']
    del c                                     []
    return $0.9                               ['$0.9']
--------------------------nopython: consts_add_one--------------------------
label 0:
    x = arg(0, name=x)                        ['x']
    $const0.1 = const(int, 11)                ['$const0.1']
    a = $const0.1                             ['$const0.1', 'a']
    del $const0.1                             []
    $const0.2 = const(float, 21.2)            ['$const0.2']
    b = $const0.2                             ['$const0.2', 'b']
    del $const0.2                             []
    $0.5 = x + a                              ['$0.5', 'a', 'x']
    del x                                     []
    del a                                     []
    $0.7 = $0.5 + b                           ['$0.5', '$0.7', 'b']
    del b                                     []
    del $0.5                                  []
    c = $0.7                                  ['$0.7', 'c']
    del $0.7                                  []
    $0.9 = cast(value=c)                      ['$0.9', 'c']
    del c                                     []
    return $0.9                               ['$0.9']
```

Note the change in the values in the `const` nodes.

Pass execution times

Numba has built-in support for timing all compiler passes, the execution times are stored in the metadata associated with a compilation result. This demonstrates one way of accessing this information based on the previously defined function, `foo`:

```python
compile_result = foo.overloads[foo.signatures[0]]
nopython_times = compile_result.metadata['pipeline_times']['nopython']
for k in nopython_times.keys():
    if ConstsAddOne._name in k:
        print(nopython_times[k])
```

the output of which is, for example:

```
pass_timings(init=1.914000677061267e-06, run=4.308700044930447e-05, finalize=1.
↪7400006981915794e-06)
```

this displaying the pass initialization, run and finalization times in seconds.

7.12 Notes on Inlining

There are occasions where it is useful to be able to inline a function at its call site, at the Numba IR level of representation. The decorators such as *numba.jit()*, `numba.extending.overload()` and `register_jitable()` support the keyword argument `inline`, to facilitate this behaviour.

When attempting to inline at this level, it is important to understand what purpose this serves and what effect this will have. In contrast to the inlining performed by LLVM, which is aimed at improving performance, the main reason to inline at the Numba IR level is to allow type inference to cross function boundaries.

As an example, consider the following snippet:

```python
from numba import njit


@njit
def bar(a):
    a.append(10)


@njit
def foo():
    z = []
    bar(z)


foo()
```

This will fail to compile and run, because the type of `z` can not be inferred as it will only be refined within `bar`. If we now add `inline=True` to the decorator for `bar` the snippet will compile and run. This is because inlining the call to `a.append(10)` will mean that `z` will be refined to hold integers and so type inference will succeed.

So, to recap, inlining at the Numba IR level is unlikely to have a performance benefit. Whereas inlining at the LLVM level stands a better chance.

The `inline` keyword argument can be one of three values:

- The string `'never'`, this is the default and results in the function not being inlined under any circumstances.

- The string `'always'`, this results in the function being inlined at all call sites.

- A python function that takes three arguments. The first argument is always the `ir.Expr` node that is the `call` requesting the inline, this is present to allow the function to make call contextually aware decisions. The second and third arguments are:

  - In the case of an untyped inline, i.e. that which occurs when using the *numba.jit()* family of decorators, both arguments are `numba.ir.FunctionIR` instances. The second argument corresponding to the IR of the caller, the third argument corresponding to the IR of the callee.

  - In the case of a typed inline, i.e. that which occurs when using `numba.extending.overload()`, both arguments are instances of a `namedtuple` with fields (corresponding to their standard use in the compiler internals):

    * `func_ir` - the function's Numba IR.

    * `typemap` - the function's type map.

    * `calltypes` - the call types of any calls in the function.

    * `signature` - the function's signature.

  The second argument holds the information from the caller, the third holds the information from the callee.

In all cases the function should return True to inline and return False to not inline, this essentially permitting custom inlining rules (typical use might be cost models).

- Recursive functions with `inline='always'` will result in a non-terminating compilation. If you wish to avoid this, supply a function to limit the recursion depth (see below).

Note: No guarantee is made about the order in which functions are assessed for inlining or about the order in which they are inlined.

7.12.1 Example using `numba.jit()`

An example of using all three options to `inline` in the `numba.njit()` decorator:

```python
from numba import njit
import numba
from numba.core import ir


@njit(inline='never')
def never_inline():
    return 100


@njit(inline='always')
def always_inline():
    return 200


def sentinel_cost_model(expr, caller_info, callee_info):
    # this cost model will return True (i.e. do inlining) if either:
    # a) the callee IR contains an `ir.Const(37)`
    # b) the caller IR contains an `ir.Const(13)` logically prior to the call
    #     site

    # check the callee
    for blk in callee_info.blocks.values():
        for stmt in blk.body:
            if isinstance(stmt, ir.Assign):
                if isinstance(stmt.value, ir.Const):
                    if stmt.value.value == 37:
                        return True

    # check the caller
    before_expr = True
    for blk in caller_info.blocks.values():
        for stmt in blk.body:
            if isinstance(stmt, ir.Assign):
                if isinstance(stmt.value, ir.Expr):
                    if stmt.value == expr:
                        before_expr = False
                if isinstance(stmt.value, ir.Const):
                    if stmt.value.value == 13:
                        return True & before_expr
    return False
```

(continues on next page)

```
@njit(inline=sentinel_cost_model)
def maybe_inline1():
    # Will not inline based on the callee IR with the declared cost model
    # The following is ir.Const(300).
    return 300


@njit(inline=sentinel_cost_model)
def maybe_inline2():
    # Will inline based on the callee IR with the declared cost model
    # The following is ir.Const(37).
    return 37



@njit
def foo():
    a = never_inline()  # will never inline
    b = always_inline()  # will always inline

    # will not inline as the function does not contain a magic constant known to
    # the cost model, and the IR up to the call site does not contain a magic
    # constant either
    d = maybe_inline1()

    # declare this magic constant to trigger inlining of maybe_inline1 in a
    # subsequent call
    magic_const = 13

    # will inline due to above constant declaration
    e = maybe_inline1()

    # will inline as the maybe_inline2 function contains a magic constant known
    # to the cost model
    c = maybe_inline2()

    return a + b + c + d + e + magic_const


foo()
```

which produces the following when executed (with a print of the IR after the legalization pass, enabled via the environment variable NUMBA_DEBUG_PRINT_AFTER="ir_legalization"):

```
label 0:
    $0.1 = global(never_inline: CPUDispatcher(<function never_inline at␣
→0x7f890ccf9048>)) ['$0.1']
    $0.2 = call $0.1(func=$0.1, args=[], kws=(), vararg=None) ['$0.1', '$0.2']
    del $0.1                                  []
    a = $0.2                                  ['$0.2', 'a']
    del $0.2                                  []
    $0.3 = global(always_inline: CPUDispatcher(<function always_inline at␣
→0x7f890ccf9598>)) ['$0.3']
    del $0.3                                  []
    $const0.1.0 = const(int, 200)             ['$const0.1.0']
    $0.2.1 = $const0.1.0                       ['$0.2.1', '$const0.1.0']
```

```
del $const0.1.0                         []
$0.4 = $0.2.1                           ['$0.2.1', '$0.4']
del $0.2.1                              []
b = $0.4                                ['$0.4', 'b']
del $0.4                                []
$0.5 = global(maybe_inline1: CPUDispatcher(<function maybe_inline1 at␣
→0x7f890ccf9ae8>)) ['$0.5']
$0.6 = call $0.5(func=$0.5, args=[], kws=(), vararg=None) ['$0.5', '$0.6']
del $0.5                                []
d = $0.6                                ['$0.6', 'd']
del $0.6                                []
$const0.7 = const(int, 13)              ['$const0.7']
magic_const = $const0.7                 ['$const0.7', 'magic_const']
del $const0.7                           []
$0.8 = global(maybe_inline1: CPUDispatcher(<function maybe_inline1 at␣
→0x7f890ccf9ae8>)) ['$0.8']
del $0.8                                []
$const0.1.2 = const(int, 300)           ['$const0.1.2']
$0.2.3 = $const0.1.2                     ['$0.2.3', '$const0.1.2']
del $const0.1.2                          []
$0.9 = $0.2.3                            ['$0.2.3', '$0.9']
del $0.2.3                               []
e = $0.9                                ['$0.9', 'e']
del $0.9                                []
$0.10 = global(maybe_inline2: CPUDispatcher(<function maybe_inline2 at␣
→0x7f890ccf9b70>)) ['$0.10']
del $0.10                               []
$const0.1.4 = const(int, 37)            ['$const0.1.4']
$0.2.5 = $const0.1.4                     ['$0.2.5', '$const0.1.4']
del $const0.1.4                          []
$0.11 = $0.2.5                           ['$0.11', '$0.2.5']
del $0.2.5                               []
c = $0.11                               ['$0.11', 'c']
del $0.11                               []
$0.14 = a + b                           ['$0.14', 'a', 'b']
del b                                   []
del a                                   []
$0.16 = $0.14 + c                       ['$0.14', '$0.16', 'c']
del c                                   []
del $0.14                               []
$0.18 = $0.16 + d                       ['$0.16', '$0.18', 'd']
del d                                   []
del $0.16                               []
$0.20 = $0.18 + e                       ['$0.18', '$0.20', 'e']
del e                                   []
del $0.18                               []
$0.22 = $0.20 + magic_const             ['$0.20', '$0.22', 'magic_const']
del magic_const                         []
del $0.20                               []
$0.23 = cast(value=$0.22)               ['$0.22', '$0.23']
del $0.22                               []
return $0.23                            ['$0.23']
```

Things to note in the above:

1. The call to the function `never_inline` remains as a call.

2. The `always_inline` function has been inlined, note its `const(int, 200)` in the caller body.

---

3. There is a call to `maybe_inline1` before the `const(int, 13)` declaration, the cost model prevented this from being inlined.

4. After the `const(int, 13)` the subsequent call to `maybe_inline1` has been inlined as shown by the `const(int, 300)` in the caller body.

5. The function `maybe_inline2` has been inlined as demonstrated by `const(int, 37)` in the caller body.

6. That dead code elimination has not been performed and as a result there are superfluous statements present in the IR.

7.12.2 Example using `numba.extending.overload()`

An example of using inlining with the `numba.extending.overload()` decorator. It is most interesting to note that if a function is supplied as the argument to `inline` a lot more information is available via the supplied function arguments for use in decision making. Also that different `@overload`s can have different inlining behaviours, with multiple ways to achieve this:

```python
import numba
from numba.extending import overload
from numba import njit, types


def bar(x):
    """A function stub to overload"""
    pass


@overload(bar, inline='always')
def ol_bar_tuple(x):
    # An overload that will always inline, there is a type guard so that this
    # only applies to UniTuples.
    if isinstance(x, types.UniTuple):
        def impl(x):
            return x[0]
        return impl


def cost_model(expr, caller, callee):
    # Only inline if the type of the argument is an Integer
    return isinstance(caller.typemap[expr.args[0].name], types.Integer)


@overload(bar, inline=cost_model)
def ol_bar_scalar(x):
    # An overload that will inline based on a cost model, it only applies to
    # scalar values in the numerical domain as per the type guard on Number
    if isinstance(x, types.Number):
        def impl(x):
            return x + 1
        return impl


@njit
def foo():

    # This will resolve via `ol_bar_tuple` as the argument is a types.UniTuple
```

(continues on next page)

```
    # instance. It will always be inlined as specified in the decorator for this
    # overload.
    a = bar((1, 2, 3))

    # This will resolve via `ol_bar_scalar` as the argument is a types.Number
    # instance, hence the cost_model will be used to determine whether to
    # inline.
    # The function will be inlined as the value 100 is an IntegerLiteral which
    # is an instance of a types.Integer as required by the cost_model function.
    b = bar(100)

    # This will also resolve via `ol_bar_scalar` as the argument is a
    # types.Number instance, again the cost_model will be used to determine
    # whether to inline.
    # The function will not be inlined as the complex value is not an instance
    # of a types.Integer as required by the cost_model function.
    c = bar(300j)

    return a + b + c


foo()
```

which produces the following when executed (with a print of the IR after the legalization pass, enabled via the environment variable NUMBA_DEBUG_PRINT_AFTER=`"ir_legalization"`):

```
label 0:
    $const0.2 = const(tuple, (1, 2, 3))      ['$const0.2']
    x.0 = $const0.2                          ['$const0.2', 'x.0']
    del $const0.2                            []
    $const0.2.2 = const(int, 0)              ['$const0.2.2']
    $0.3.3 = getitem(value=x.0, index=$const0.2.2) ['$0.3.3', '$const0.2.2', 'x.0']
    del x.0                                  []
    del $const0.2.2                          []
    $0.4.4 = $0.3.3                          ['$0.3.3', '$0.4.4']
    del $0.3.3                               []
    $0.3 = $0.4.4                            ['$0.3', '$0.4.4']
    del $0.4.4                               []
    a = $0.3                                 ['$0.3', 'a']
    del $0.3                                 []
    $const0.5 = const(int, 100)              ['$const0.5']
    x.5 = $const0.5                          ['$const0.5', 'x.5']
    del $const0.5                            []
    $const0.2.7 = const(int, 1)              ['$const0.2.7']
    $0.3.8 = x.5 + $const0.2.7               ['$0.3.8', '$const0.2.7', 'x.5']
    del x.5                                  []
    del $const0.2.7                          []
    $0.4.9 = $0.3.8                          ['$0.3.8', '$0.4.9']
    del $0.3.8                               []
    $0.6 = $0.4.9                            ['$0.4.9', '$0.6']
    del $0.4.9                               []
    b = $0.6                                 ['$0.6', 'b']
    del $0.6                                 []
    $0.7 = global(bar: <function bar at 0x7f6c3710d268>) ['$0.7']
    $const0.8 = const(complex, 300j)         ['$const0.8']
    $0.9 = call $0.7($const0.8, func=$0.7, args=[Var($const0.8, inline_overload_
→example.py (56))], kws=(), vararg=None) ['$0.7', '$0.9', '$const0.8']
```

```
    del $const0.8                          []
    del $0.7                               []
    c = $0.9                               ['$0.9', 'c']
    del $0.9                               []
    $0.12 = a + b                          ['$0.12', 'a', 'b']
    del b                                  []
    del a                                  []
    $0.14 = $0.12 + c                      ['$0.12', '$0.14', 'c']
    del c                                  []
    del $0.12                              []
    $0.15 = cast(value=$0.14)              ['$0.14', '$0.15']
    del $0.14                              []
    return $0.15                           ['$0.15']
```

Things to note in the above:

1. The first highlighted section is the always inlined overload for the `UniTuple` argument type.

2. The second highlighted section is the overload for the `Number` argument type that has been inlined as the cost model function decided to do so as the argument was an `Integer` type instance.

3. The third highlighted section is the overload for the `Number` argument type that has not inlined as the cost model function decided to reject it as the argument was an `Complex` type instance.

4. That dead code elimination has not been performed and as a result there are superfluous statements present in the IR.

7.12.3 Using a function to limit the inlining depth of a recursive function

When using recursive inlines, you can terminate the compilation by using a cost model.

```python
from numba import njit
import numpy as np

class CostModel(object):
    def __init__(self, max_inlines):
        self._count = 0
        self._max_inlines = max_inlines

    def __call__(self, expr, caller, callee):
        ret = self._count < self._max_inlines
        self._count += 1
        return ret

@njit(inline=CostModel(3))
def factorial(n):
    if n <= 0:
        return 1
    return n * factorial(n - 1)

factorial(5)
```

7.13 Environment Object

The Environment object (Env) is used to maintain references to python objects that are needed to support compiled functions for both object-mode and nopython-mode.

In nopython-mode, the Env is used for:

- Storing pyobjects for reconstruction from native values, such as:
    - for printing native values of NumPy arrays;
    - for returning or yielding native values back to the interpreter.

In object-mode, the Env is used for:

- storing constant values referenced in the code.
- storing a reference to the function's global dictionary to load global values.

7.13.1 The Implementation

The Env is implemented in two parts. In `_dynfunc.c`, the Env is defined as `EnvironmentObject` as a Python C-extension type. In `lowering.py`, the *EnvironmentObject`* (exported as `_dynfunc.Environment`) is extended to support necessary operations needed at lowering.

Serialization

The Env supports being pickled. Compilation cache files and ahead-of-time compiled modules serialize all the used Envs for recreation at the runtime.

Usage

At the start of the lowering for a function or a generator, an Env is created. Throughout the compilation, the Env is mutated to attach additional information. The compiled code references an Env via a global variable in the emitted LLVM IR. The global variable is zero-initialized with "common" linkage, which is the default linkage for C global values. The use of this linkage allows multiple definitions of the global variable to be merged into a single definition when the modules are linked together. The name of the global variable is computed from the name of the function (see `FunctionDescriptor.env_name` and `.get_env_name()` of the target context).

The Env is initialized when the compiled-function is loaded. The JIT engine finds the address of the associated global variable for the Env and stores the address of the Env into it. For cached functions, the same process applies. For ahead-of-time compiled functions, the module initializer in the generated library is responsible for initializing the global variables of all the Envs in the module.

7.14 Notes on Hashing

Numba supports the built-in `hash()` and does so by simply calling the `__hash__()` member function on the supplied argument. This makes it trivial to add hash support for new types as all that is required is the application of the extension API `overload_method()` decorator to overload a function for computing the hash value for the new type registered to the type's `__hash__()` method. For example:

```
from numba.extending import overload_method

@overload_method(myType, '__hash__')
def myType_hash_overload(obj):
    # implementation details
```

7.14.1 The Implementation

The implementation of the Numba hashing functions strictly follows that of Python 3. The only exception to this is that for hashing Unicode and bytes (for content longer than sys.hash_info.cutoff) the only supported algorithm is siphash24 (default in CPython 3). As a result Numba will match Python 3 hash values for all supported types under the default conditions described.

Unicode hash cache differences

Both Numba and CPython Unicode string internal representations have a hash member for the purposes of caching the string's hash value. This member is always checked ahead of computing a hash value the with view of simply providing a value from cache as it is considerably cheaper to do so. The Numba Unicode string hash caching implementation behaves in a similar way to that of CPython's. The only notable behavioral change (and its only impact is a minor potential change in performance) is that Numba always computes and caches the hash for Unicode strings created in nopython mode at the time they are boxed for reuse in Python, this is too eager in some cases in comparison to CPython which may delay hashing a new Unicode string depending on creation method. It should also be noted that Numba copies in the hash member of the CPython internal representation for Unicode strings when unboxing them to its own representation so as to not recompute the hash of a string that already has a hash value associated with it.

The accommodation of `PYTHONHASHSEED`

The PYTHONHASHSEED environment variable can be used to seed the CPython hashing algorithms for e.g. the purposes of reproduciblity. The Numba hashing implementation directly reads the CPython hashing algorithms' internal state and as a result the influence of PYTHONHASHSEED is replicated in Numba's hashing implementations.

7.15 Notes on Caching

Numba supports caching of compiled functions into the filesystem for future use of the same functions.

7.15.1 The Implementation

Caching is done by saving the compiled *object code*, the ELF object of the executable code. By using the *object code*, cached functions have minimal overhead because no compilation is needed. The cached data is saved under the cache directory (see *NUMBA_CACHE_DIR*). The index of the cache is stored in a .nbi file, with one index per function, and it lists all the overloaded signatures compiled for the function. The *object code* is stored in files with an .nbc extension, one file per overload. The data in both files is serialized with pickle.

Note: On Python <=3.7, Numba extends pickle using the pure-Python pickler. To use the faster C Pickler, install pickle5 from pip. pickle5 backports Python 3.8 pickler features.

Requirements for Cacheability

Developers should note the requirements of a function to permit it to be cached to ensure that the features they are working on are compatible with caching.

Requirements for cacheable function:

- The LLVM module must be *self-contained*, meaning that it cannot rely on other compiled units without linking to them.

- The only allowed external symbols are from the *NRT* or other common symbols from system libraries (i.e. libc and libm).

Debugging note:

- Look for the usage of `inttoptr` in the LLVM IR or `target_context.add_dynamic_add()` in the lowering code in Python. They indicate potential usage of runtime address. Not all uses are problematic and some are necessary. Only the conversion of constant integers into pointers will affect caching.

- Misuse of dynamic address or dynamic symbols will likely result in a segfault.

- Linking order matters because unused symbols are dropped after linking. Linking should start from the leaf nodes of the dependency graph.

Features Compatible with Caching

The following features are explicitly verified to work with caching.

- ufuncs and gufuncs for the `cpu` and `parallel` target

- parallel accelerator features (i.e. `parallel=True`)

Caching Limitations

This is a list of known limitation of the cache:

- Cache invalidation fails to recognize changes in symbols defined in a different file.

- Global variables are treated as constants. The cache will remember the value in the global variable used at compilation. On cache load, the cached function will not rebind to the new value of the global variable.

Cache Sharing

It is safe to share and reuse the contents in the cache directory on a different machine. The cache remembers the CPU model and the available CPU features during compilation. If the CPU model and the CPU features do not match exactly, the cache contents will not be considered. (Also see *NUMBA_CPU_NAME*)

If the cache directory is shared on a network filesystem, concurrent read/write of the cache is safe only if file replacement operation is atomic for the filesystem. Numba always writes to a unique temporary file first, it then replaces the target cache file path with the temporary file. Numba is tolerant against lost cache files and lost cache entries.

Cache Clearing

The cache is invalidated when the corresponding source file is modified. However, it is necessary sometimes to clear the cache directory manually. For instance, changes in the compiler will not be recognized because the source files are not modified.

To clear the cache, the cache directory can be simply removed.

Removing the cache directory when a Numba application is running may cause an `OSError` exception to be raised at the compilation site.

Related Environment Variables

See *env-vars for caching*.

7.16 Notes on Numba's threading implementation

The execution of the work presented by the Numba `parallel` targets is undertaken by the Numba threading layer. Practically, the "threading layer" is a Numba built-in library that can perform the required concurrent execution. At the time of writing there are three threading layers available, each implemented via a different lower level native threading library. More information on the threading layers and appropriate selection of a threading layer for a given application/system can be found in the *threading layer documentation*.

The pertinent information to note for the following sections is that the function in the threading library that performs the parallel execution is the `parallel_for` function. The job of this function is to both orchestrate and execute the parallel tasks.

The relevant source files referenced in this document are

- `numba/np/ufunc/tbbpool.cpp`

- `numba/np/ufunc/omppool.cpp`

- `numba/np/ufunc/workqueue.c`

  These files contain the TBB, OpenMP, and workqueue threadpool implementations, respectively. Each includes the functions `set_num_threads()`, `get_num_threads()`, and `get_thread_id()`, as well as the relevant logic for thread masking in their respective schedulers. Note that the basic thread local variable logic is duplicated in each of these files, and not shared between them.

- `numba/np/ufunc/parallel.py`

  This file contains the Python and JIT compatible wrappers for `set_num_threads()`, `get_num_threads()`, and `get_thread_id()`, as well as the code that loads the above libraries into Python and launches the threadpool.

- `numba/parfors/parfor_lowering.py`

  This file contains the main logic for generating code for the parallel backend. The thread mask is accessed in this file in the code that generates scheduler code, and passed to the relevant backend scheduler function (see below).

7.16.1 Thread masking

As part of its design, Numba never launches new threads beyond the threads that are launched initially with `numba.np.ufunc.parallel._launch_threads()` when the first parallel execution is run. This is due to the way threads were already implemented in Numba prior to thread masking being implemented. This restriction was kept to keep the design simple, although it could be removed in the future. Consequently, it's possible to programmatically set the number of threads, but only to less than or equal to the total number that have already been launched. This is done by "masking" out unused threads, causing them to do no work. For example, on a 16 core machine, if the user were to call `set_num_threads(4)`, Numba would always have 16 threads present, but 12 of them would sit idle for parallel computations. A further call to `set_num_threads(16)` would cause those same threads to do work in later computations.

Thread masking was added to make it possible for a user to programmatically alter the number of threads performing work in the threading layer. Thread masking proved challenging to implement as it required the development of a programming model that is suitable for users, easy to reason about, and could be implemented safely, with consistent behavior across the various threading layers.

Programming model

The programming model chosen is similar to that found in OpenMP. The reasons for this choice were that it is familiar to a lot of users, restricted in scope and also simple. The number of threads in use is specified by calling `set_num_threads` and the number of threads in use can be queried by calling `get_num_threads`.These two functions are synonymous with their OpenMP counterparts (with the above restriction that the mask must be less than or equal to the number of launched threads). The execution semantics are also similar to OpenMP in that once a parallel region is launched, altering the thread mask has no impact on the currently executing region, but will have an impact on parallel regions executed subsequently.

The Implementation

So as to place no further restrictions on user code other than those that already existed in the threading layer libraries, careful consideration of the design of thread masking was required. The "thread mask" cannot be stored in a global value as concurrent use of the threading layer may result in classic forms of race conditions on the value itself. Numerous designs were discussed involving various types of mutex on such a global value, all of which were eventually broken through thought experiment alone. It eventually transpired that, following some OpenMP implementations, the "thread mask" is best implemented as a `thread local`. This means each thread that executes a Numba parallel function will have a thread local storage (TLS) slot that contains the value of the thread mask to use when scheduling threads in the `parallel_for` function.

The above notion of TLS use for a thread mask is relatively easy to implement, `get_num_threads` and `set_num_threads` simply need to address the TLS slot in a given threading layer. This also means that the execution schedule for a parallel region can be derived from a run time call to `get_num_threads`. This is achieved via a well known and relatively easy to implement pattern of a `C` library function registration and wrapping it in the internal Numba implementation.

In addition to satisfying the original upfront thread masking requirements, a few more complicated scenarios needed consideration as follows.

Nested parallelism

In all threading layers a "main thread" will invoke the `parallel_for` function and then in the parallel region, depending on the threading layer, some number of additional threads will assist in doing the actual work. If the work contains a call to another parallel function (i.e. nested parallelism) it is necessary for the thread making the call to know what the "thread mask" of the main thread is so that it can propagate it into the `parallel_for` call it makes when executing the nested parallel function. The implementation of this behavior is threading layer specific but the general principle is for the "main thread" to always "send" the value of the thread mask from its TLS slot to all threads in the threading layer that are active in the parallel region. These active threads then update their TLS slots with this value prior to performing any work. The net result of this implementation detail is that:

- thread masks correctly propagate into nested functions

- it's still possible for each thread in a parallel region to safely have a different mask with which to call nested functions, if it's not set explicitly then the inherited mask from the "main thread" is used

- threading layers which have dynamic scheduling with threads potentially joining and leaving the active pool during a `parallel_for` execution are successfully accommodated

- any "main thread" thread mask is entirely decoupled from the in-flux nature of the thread masks of the threads in the active thread pool

Python threads independently invoking parallel functions

The threading layer launch sequence is heavily guarded to ensure that the launch is both thread and process safe and run once per process. In a system with numerous Python `threading` module threads all using Numba, the first thread through the launch sequence will get its thread mask set appropriately, but no further threads can run the launch sequence. This means that other threads will need their initial thread mask set some other way. This is achieved when `get_num_threads` is called and no thread mask is present, in this case the thread mask will be set to the default. In the implementation, "no thread mask is present" is represented by the value -1 and the "default thread mask" (unset) is represented by the value 0. The implementation also immediately calls `set_num_threads(NUMBA_NUM_THREADS)` after doing this, so if either -1 or 0 is encountered as a result from `get_num_threads()` it indicates a bug in the above processes.

OS `fork()` calls

The use of TLS was also in part driven by the Linux (the most popular platform for Numba use by far) having a `fork(2, 3P)` call that will do TLS propagation into child processes, see `clone(2)`'s CLONE_SETTLS.

Thread ID

A private `get_thread_id()` function was added to each threading backend, which returns a unique ID for each thread. This can be accessed from Python by `numba.np.ufunc.parallel._get_thread_id()` (it can also be used inside a JIT compiled function). The thread ID function is useful for testing that the thread masking behavior is correct, but it should not be used outside of the tests. For example, one can call `set_num_threads(4)` and then collect all unique `_get_thread_id()`s in a parallel region to verify that only 4 threads are run.

Caveats

Some caveats to be aware of when testing thread masking:

- The TBB backend may choose to schedule fewer than the given mask number of threads. Thus a test such as the one described above may return fewer than 4 unique threads.

- The workqueue backend is not threadsafe, so attempts to do multithreading nested parallelism with it may result in deadlocks or other undefined behavior. The workqueue backend will raise a SIGABRT signal if it detects nested parallelism.

- Certain backends may reuse the main thread for computation, but this behavior shouldn't be relied upon (for instance, if propagating exceptions).

Use in Code Generation

The general pattern for using `get_num_threads` in code generation is

```
import llvmlite.llvmpy.core as lc

get_num_threads = builder.module.get_or_insert_function(
    lc.Type.function(lc.Type.int(types.intp.bitwidth), []),
    name="get_num_threads")
```

```
num_threads = builder.call(get_num_threads, [])

with cgutils.if_unlikely(builder, builder.icmp_signed('<=', num_threads,
                                        num_threads.type(0))):
    cgutils.printf(builder, "num_threads: %d\n", num_threads)
    context.call_conv.return_user_exc(builder, RuntimeError,
                                        ("Invalid number of threads. "
                                         "This likely indicates a bug in Numba.
↪",))

# Pass num_threads through to the appropriate backend function here
```

See the code in `numba/parfors/parfor_lowering.py`.

The guard against `num_threads` being <= 0 is not strictly necessary, but it can protect against accidentally incorrect behavior in case the thread masking logic contains a bug.

The `num_threads` variable should be passed through to the appropriate backend function, such as `do_scheduling` or `parallel_for`. If it's used in some way other than passing it through to the backend function, the above considerations should be taken into account to ensure the use of the `num_threads` variable is safe. It would probably be better to keep such logic in the threading backends, rather than trying to do it in code generation.

7.17 Notes on Literal Types

Note: This document describes an advanced feature designed to overcome some limitations of the compilation mechanism relating to types.

Some features need to specialize based on the literal value during compliation to produce type stable code necessary for successful compilation in Numba. This can be achieved by propagating the literal value through the type system. Numba recognizes inline literal values as *numba.types.Literal*. For example:

```
def foo(x):
    a = 123
    return bar(x, a)
```

Numba will infer the type of `a` as `Literal[int](123)`. The definition of `bar()` can subsequently specialize its implementation knowing that the second argument is an `int` with the value `123`.

7.17.1 `Literal` Type

Classes and methods related to the `Literal` type.

class numba.types.**Literal**(*value*)

    Base class for Literal types. Literal types contain the original Python value in the type.

    A literal type should always be constructed from the *literal(val)* function.

numba.types.**literal**(*value*)

    Returns a Literal instance or raise LiteralTypingError

numba.types.**unliteral**(*lit_type*)

    Get base type from Literal type.

numba.types.**maybe_literal**(*value*)
> Get a Literal type for the value or None.

7.17.2 Specifying for Literal Typing

To specify a value as a `Literal` type in code scheduled for JIT compilation, use the following function:

numba.**literally**(*obj*)
> Forces Numba to interpret *obj* as an Literal value.
>
> *obj* must be either a literal or an argument of the caller function, where the argument must be bound to a literal. The literal requirement propagates up the call stack.
>
> This function is intercepted by the compiler to alter the compilation behavior to wrap the corresponding function parameters as `Literal`. It has **no effect** outside of nopython-mode (interpreter, and objectmode).
>
> The current implementation detects literal arguments in two ways:
>
> 1. Scans for uses of `literally` via a compiler pass.
> 2. `literally` is overloaded to raise `numba.errors.ForceLiteralArg` to signal the dispatcher to treat the corresponding parameter differently. This mode is to support indirect use (via a function call).
>
> The execution semantic of this function is equivalent to an identity function.
>
> See numba/tests/test_literal_dispatch.py for examples.

Code Example

Listing 2: from `test_literally_usage` of `numba/tests/doc_examples/test_literally_usage.py`

```
1  import numba
2
3  def power(x, n):
4      raise NotImplementedError
5
6  @numba.extending.overload(power)
7  def ov_power(x, n):
8      if isinstance(n, numba.types.Literal):
9          # only if `n` is a literal
10         if n.literal_value == 2:
11             # special case: square
12             print("square")
13             return lambda x, n: x * x
14         elif n.literal_value == 3:
15             # special case: cubic
16             print("cubic")
17             return lambda x, n: x * x * x
18
19     print("generic")
20     return lambda x, n: x ** n
21
22  @numba.njit
23  def test_power(x, n):
24      return power(x, numba.literally(n))
25
26  # should print "square" and "9"
```

(continues on next page)

```
27          print(test_power(3, 2))
28
29          # should print "cubic" and "27"
30          print(test_power(3, 3))
31
32          # should print "generic" and "81"
33          print(test_power(3, 4))
34
```

Internal Details

Internally, the compiler raises a `ForceLiteralArgs` exception to signal the dispatcher to wrap specified arguments using the `Literal` type.

class numba.errors.**ForceLiteralArg**(*arg_indices*, *fold_arguments=None*, *loc=None*)
    A Pseudo-exception to signal the dispatcher to type an argument literally

        Attributes

            requested_args [frozenset[int]] requested positions of the arguments.

    __init__(*arg_indices*, *fold_arguments=None*, *loc=None*)

        Parameters

            arg_indices [Sequence[int]] requested positions of the arguments.

            fold_arguments: callable A function (`tuple`, `dict`) `-> tuple` that binds and flattens the `args` and `kwargs`.

            loc [numba.ir.Loc or None]

    __or__(*other*)
        Same as self.combine(other)

    combine(*other*)
        Returns a new instance by or'ing the requested_args.

7.17.3 Inside Extensions

`@overload` extensions can use `literally` inside the implementation body like in normal jit-code.

Explicit handling of literal requirements is possible through use of the following:

class numba.extending.**SentryLiteralArgs**

        Parameters

            literal_args [Sequence[str]] A sequence of names for literal arguments

    Examples

    The following line:

```
>>> SentryLiteralArgs(literal_args).for_pysig(pysig).bind(*args, **kwargs)
```

    is equivalent to:

```
>>> sentry_literal_args(pysig, literal_args, args, kwargs)
```

for_function(*func*)
  Bind the sentry to the signature of *func*.

> **Parameters**
>
>> **func** [Function] A python function.
>
> **Returns**
>
>> **obj** [BoundLiteralArgs]

for_pysig(*pysig*)
  Bind the sentry to the given signature *pysig*.

> **Parameters**
>
>> **pysig** [inspect.Signature]
>
> **Returns**
>
>> **obj** [BoundLiteralArgs]

class numba.extending.**BoundLiteralArgs**
  This class is usually created by SentryLiteralArgs.

bind(**args*, ***kwargs*)
  Bind to argument types.

numba.extending.**sentry_literal_args**(*pysig*, *literal_args*, *args*, *kwargs*)
  Ensures that the given argument types (in *args* and *kwargs*) are literally typed for a function with the python signature *pysig* and the list of literal argument names in *literal_args*.

  Alternatively, this is the same as:

```
SentryLiteralArgs(literal_args).for_pysig(pysig).bind(*args, **kwargs)
```

7.18 Notes on Debugging

This section describes techniques that can be useful in debugging the compilation and execution of generated code.

See also:

Debugging JIT compiled code with GDB

7.18.1 Memcheck

Memcheck is a memory error detector implemented using Valgrind. It is useful for detecting memory errors in compiled code, particularly out-of-bounds accesses and use-after-free errors. Buggy or miscompiled native code can generate these kinds of errors. The Memcheck documentation explains its usage; here, we discuss only the specifics of using it with Numba.

The Python interpreter and some of the libraries used by Numba can generate false positives with Memcheck - see this section of the manual for more information on why false positives occur. The false positives can make it difficult to determine when an actual error has occurred, so it is helpful to suppress known false positives. This can be done by supplying a suppressions file, which instructs Memcheck to ignore errors that match the suppressions defined in it.

The CPython source distribution includes a suppressions file, in the file `Misc/valgrind-python.supp`. Using this file prevents a lot of spurious errors generated by Python's memory allocation implementation. Additionally, the Numba repository includes a suppressions file in `contrib/valgrind-numba.supp`.

Note: It is important to use the suppressions files from the versions of the Python interpreter and Numba that you are using - these files evolve over time, so non-current versions can fail to suppress some errors, or erroneously suppress actual errors.

To run the Python interpreter under Memcheck with both suppressions files, it is invoked with the following command:

```
valgrind --tool=memcheck \
         --suppressions=${CPYTHON_SRC_DIR}/Misc/valgrind-python.supp \
         --suppressions=${NUMBA_SRC_DIR}/contrib/valgrind-numba.supp \
         python ${PYTHON_ARGS}
```

where `${CPYTHON_SRC_DIR}` is set to the location of the CPython source distribution, `${NUMBA_SRC_DIR}` is the location of the Numba source dir, and `${PYTHON_ARGS}` are the arguments to the Python interpreter.

If there are errors, then messages describing them will be printed to standard error. An example of an error is:

```
==77113==     at 0x24169A: PyLong_FromLong (longobject.c:251)
==77113==     by 0x241881: striter_next (bytesobject.c:3084)
==77113==     by 0x2D3C95: _PyEval_EvalFrameDefault (ceval.c:2809)
==77113==     by 0x21B499: _PyEval_EvalCodeWithName (ceval.c:3930)
==77113==     by 0x26B436: _PyFunction_FastCallKeywords (call.c:433)
==77113==     by 0x2D3605: call_function (ceval.c:4616)
==77113==     by 0x2D3605: _PyEval_EvalFrameDefault (ceval.c:3124)
==77113==     by 0x21B977: _PyEval_EvalCodeWithName (ceval.c:3930)
==77113==     by 0x21C2A4: _PyFunction_FastCallDict (call.c:376)
==77113==     by 0x2D5129: do_call_core (ceval.c:4645)
==77113==     by 0x2D5129: _PyEval_EvalFrameDefault (ceval.c:3191)
==77113==     by 0x21B499: _PyEval_EvalCodeWithName (ceval.c:3930)
==77113==     by 0x26B436: _PyFunction_FastCallKeywords (call.c:433)
==77113==     by 0x2D46DA: call_function (ceval.c:4616)
==77113==     by 0x2D46DA: _PyEval_EvalFrameDefault (ceval.c:3139)
==77113==
==77113== Use of uninitialised value of size 8
```

The traceback provided only outlines the C call stack, which can make it difficult to determine what the Python interpreter was doing at the time of the error. One can learn more about the state of the stack by looking at the backtrace in the GNU Debugger (GDB). Launch `valgrind` with an additional argument, `--vgdb-error=0` and attach to the process using GDB as instructed by the output. Once an error is encountered, GDB will stop at the error and the stack can be inspected.

GDB does provide support for backtracing through the Python stack, but this requires symbols which may not be easily available in your Python distribution. In this case, it is still possible to determine some information about what was happening in Python, but this depends on examining the backtrace closely. For example, in a backtrace corresponding to the above error, we see items such as:

We can see some of the arguments, in particular the names of the compiled functions, e.g:

```
_ZN5numba4cuda5tests6cudapy13test_constmem19cuconstRecAlign
↪$247E5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE5ArrayIdLi1E1C7mutable7alignedE5
```

We can run this through `c++filt` to see a more human-readable representation:

```
numba::cuda::tests::cudapy::test_constmem::cuconstRecAlign$247(
  Array<double, 1, C, mutable, aligned>,
  Array<double, 1, C, mutable, aligned>,
  Array<double, 1, C, mutable, aligned>,
  Array<double, 1, C, mutable, aligned>,
  Array<double, 1, C, mutable, aligned>)
```

which is the fully qualified name of a jitted function and the types with which it was called.

7.19 Numba Project Roadmap

Note: This page was last revised in *December 2018*.

This roadmap is for informational purposes only. Priorities and resources change, so we may choose to reorder or abandon things on this list. Additionally, the further out items are, the less concrete they will be. If you have an interest in working on one of these items, please open an issue where we can discuss the design and approach first.

7.19.1 Short Term: 2019H1

- Container improvements:
    - Numba dictionary support
    - Refactor lists to follow new container best practices. See the discussion in issue 3546.
- Deprecate Python 2.7 support
- Improve caching:
    - Full support for functions compiled with ParallelAccelerator
    - Safe caching of generated functions (eval of strings)
    - Expire cache when any function in call chain (even in other files) changes
    - Process for distributing pre-populated cache
- Continue to improve usability and debugging:
    - Trap more unsupported features earlier in pipeline (especially things that parfors can't handle)
    - Error messages
    - Diagnostic tools for debugging and understanding performance
    - Better on-boarding for new users and contributors (revise docs, more examples)
- Begin refactoring existing features that cause common bug reports:
    - Enhance description of interfaces provided by Numba functions to give more type information
    - Convert older Numba function implementations to use public extension mechanisms
    - More unit testing and modularization of ParallelAccelerator passes

7.19.2 Medium Term: 2019H2

- Unify dispatch of regular functions, ufuncs, and gufuncs

- Declare Numba 1.0 with stable interfaces

- Continue to improve usability and debugging (see above)

- Continue refactoring Numba internals to solve common bug reports (see above)

- JIT class review and improvement

- Improve compilation speed

- Improve memory management of Numba-allocated memory

- Better support for writing code transformation passes

- Make caching and parallel execution features opt-out instead of opt-in

  - add heuristic to determine if parfor passes will be beneficial

7.19.3 Long Term: 2020 and beyond

- Unify GPU backends (share more code and interfaces)

- Improve ahead of time compilation (for low powered devices)

- Improve cross language connections (C++, JVM?, Julia?, R?)

  - Call Numba from other languages,

  - Call from Numba into other languages

- Better support for "hybrid" CPU/GPU/TPU/etc programming

- Partial / deferred compilation of functions

- Foster integration of Numba into core PyData packages:

  - scipy/scikit-learn/scikit-image/pandas

- More support for efforts to put Numba into other applications (databases, etc) for compiling user-defined functions

- More support for usage of Numba as a "compiler toolkit" to create custom compilers (like HPAT, automatic differentiation of functions, etc)

- Investigate AST-based Numba frontend in addition to existing bytecode-based frontend

Numba Enhancement Proposals

Numba Enhancement Proposals (not really abbreviated "NEPs", since "NEP" is already taken by the Numpy project) describe proposed changes to Numba. They are modeled on Python Enhancement Proposals (PEPs) and Numpy Enhancement Proposals, and are typically written up when important changes (behavioural changes, feature additions...) to Numba are proposed.

This page provides an overview of all proposals, making only a distinction between the ones that have been implemented and those that have not been implemented.

8.1 Implemented proposals

8.1.1 NBEP 1: Changes in integer typing

Author Antoine Pitrou

Date July 2015

Status Final

Current semantics

Type inference of integers in Numba currently has some subtleties and some corner cases. The simple case is when some variable has an obvious Numba type (for example because it is the result of a constructor call to a Numpy scalar type such as `np.int64`). That case suffers no ambiguity.

The less simple case is when a variable doesn't bear such explicit information. This can happen because it is inferred from a built-in Python `int` value, or from an arithmetic operation between two integers, or other cases yet. Then Numba has a number of rules to infer the resulting Numba type, especially its signedness and bitwidth.

Currently, the generic case could be summarized as: *start small, grow bigger as required.* Concretely:

1. Each constant or pseudo-constant is inferred using the *smallest signed integer type* that can correctly represent it (or, possibly, `uint64` for positive integers between `2**63` and `2**64 - 1`).

2. The result of an operation is typed so as to ensure safe representation in the face of overflow and other magnitude increases (for example, `int32 + int32` would be typed `int64`).

3. As an exception, a Python `int` used as function argument is always typed `intp`, a pointer-size integer. This is to avoid the proliferation of compiled specializations, as otherwise various integer bitwidths in input arguments may produce multiple signatures.

Note: The second rule above (the "respect magnitude increases" rule) reproduces Numpy's behaviour with arithmetic on scalar values. Numba, however, has different implementation and performance constraints than Numpy scalars.

It is worth nothing, by the way, that Numpy arrays do not implement said rule (i.e. `array(int32) + array(int32)` is typed `array(int32)`, not `array(int64)`). Probably because this makes performance more controllable.

This has several non-obvious side-effects:

1. It is difficult to predict the precise type of a value inside a function, after several operations. The basic operands in an expression tree may for example be `int8` but the end result may be `int64`. Whether this is desirable or not is an open question; it is good for correctness, but potentially bad for performance.

2. In trying to follow the correctness over predictability rule, some values can actually leave the integer realm. For example, `int64 + uint64` is typed `float64` in order to avoid magnitude losses (but incidentally will lose precision on large integer values. . .), again following Numpy's semantics for scalars. This is usually not intended by the user.

3. More complicated scenarios can produce unexpected errors at the type unification stage. An example is at Github issue 1299, the gist of which is reproduced here:

```python
@jit(nopython=True)
def f():
    variable = 0
    for i in range(1):
        variable = variable + 1
    return np.arange(variable)
```

At the time of this writing, this fails compiling, on a 64-bit system, with the error:

```
numba.errors.TypingError: Failed at nopython (nopython frontend)
Can't unify types of variable '$48.4': $48.4 := {array(int32, 1d, C), array(int64,
↪ 1d, C)}
```

People expert with Numba's type unification system can understand why. But the user is caught in mystery.

Proposal: predictable width-conserving typing

We propose to turn the current typing philosophy on its head. Instead of "*start small and grow as required*", we propose "*start big and keep the width unchanged*".

Concretely:

1. The typing of Python `int` values used as function arguments doesn't change, as it works satisfyingly and doesn't surprise the user.

2. The typing of integer *constants* (and pseudo-constants) changes to match the typing of integer arguments. That is, every non-explicitly typed integer constant is typed `intp`, the pointer-sized integer; except for the rare cases where `int64` (on 32-bit systems) or `uint64` is required.

3. Operations on integers promote bitwidth to `intp`, if smaller, otherwise they don't promote. For example, on a 32-bit machine, `int8 + int8` is typed `int32`, as is `int32 + int32`. However, `int64 + int64` is typed `int64`.

4. Furthermore, mixed operations between signed and unsigned fall back to signed, while following the same bitwidth rule. For example, on a 32-bit machine, `int8 + uint16` is typed `int32`, as is `uint32 + int32`.

Proposal impact

Semantics

With this proposal, the semantics become clearer. Regardless of whether the arguments and constants of a function were explicitly typed or not, the results of various expressions at any point in the function have easily predictable types.

When using built-in Python `int`, the user gets acceptable magnitude (32 or 64 bits depending on the system's bitness), and the type remains the same across all computations.

When explicitly using smaller bitwidths, intermediate results don't suffer from magnitude loss, since their bitwidth is promoted to `intp`.

There is also less potential for annoyances with the type unification system as demonstrated above. The user would have to force several different types to be faced with such an error.

One potential cause for concern is the discrepancy with Numpy's scalar semantics; but at the same time this brings Numba scalar semantics closer to array semantics (both Numba's and Numpy's), which seems a desirable outcome as well.

It is worth pointing out that some sources of integer numbers, such as the `range()` built-in, always yield 32-bit integers or larger. This proposal could be an opportunity to standardize them on `intp`.

Performance

Except in trivial cases, it seems unlikely that the current "best fit" behaviour for integer constants really brings a performance benefit. After all, most integers in Numba code would either be stored in arrays (with well-known types, chosen by the user) or be used as indices, where a `int8` is highly unlikely to fare better than a `intp` (actually, it may be worse, if LLVM isn't able to optimize away the required sign-extension).

As a side note, the default use of `intp` rather than `int64` ensures that 32-bit systems won't suffer from poor arithmetic performance.

Implementation

Optimistically, this proposal may simplify some Numba internals a bit. Or, at least, it doesn't threaten to make them significantly more complicated.

Limitations

This proposal doesn't really solve the combination of signed and unsigned integers. It is geared mostly at solving the bitwidth issues, which are a somewhat common cause of pain for users. Unsigned integers are in practice very uncommon in Numba-compiled code, except when explicitly asked for, and therefore much less of a pain point.

On the bitwidth front, 32-bit systems could still show discrepancies based on the values of constants: if a constant is too large to fit in 32 bits, it is typed `int64`, which propagates through other computations. This would be a reminiscence of the current behaviour, but rarer and much more controlled still.

Long-term horizon

While we believe this proposal makes Numba's behaviour more regular and more predictable, it also pulls it further from general compatibility with pure Python semantics, where users can assume arbitrary-precision integers without any truncation issues.

8.1.2 NBEP 7: CUDA External Memory Management Plugins

Author Graham Markall, NVIDIA

Contributors Thomson Comer, Peter Entschev, Leo Fang, John Kirkham, Keith Kraus

Date March 2020

Status Final

Background and goals

The *CUDA Array Interface* enables sharing of data between different Python libraries that access CUDA devices. However, each library manages its own memory distinctly from the others. For example:

- Numba internally manages memory for the creation of device and mapped host arrays.
- The RAPIDS libraries (cuDF, cuML, etc.) use the Rapids Memory Manager for allocating device memory.
- CuPy includes a memory pool implementation for both device and pinned memory.

The goal of this NBEP is to describe a plugin interface that enables Numba's internal memory management to be replaced with an external memory manager by the user. When the plugin interface is in use, Numba no longer directly allocates or frees any memory when creating arrays, but instead requests allocations and frees through the external manager.

Requirements

Provide an *External Memory Manager (EMM)* interface in Numba.

- When the EMM is in use, Numba will make all memory allocation using the EMM. It will never directly call functions such as `CuMemAlloc`, `cuMemFree`, etc.
- When not using an *External Memory Manager (EMM)*, Numba's present behaviour is unchanged (at the time of writing, the current version is the 0.48 release).

If an EMM is to be used, it will entirely replace Numba's internal memory management for the duration of program execution. An interface for setting the memory manager will be provided.

Device vs. Host memory

An EMM will always take responsibility for the management of device memory. However, not all CUDA memory management libraries also support managing host memory, so a facility for Numba to continue the management of host memory whilst ceding control of device memory to the EMM will be provided.

Deallocation strategies

Numba's internal memory management uses a *deallocation strategy* designed to increase efficiency by deferring deallocations until a significant quantity are pending. It also provides a mechanism for preventing deallocations entirely during critical sections, using the *defer_cleanup()* context manager.

- When the EMM is not in use, the deallocation strategy and operation of `defer_cleanup` remain unchanged.

- When the EMM is in use, the deallocation strategy is implemented by the EMM, and Numba's internal deallocation mechanism is not used. For example:

    - A similar strategy to Numba's could be implemented by the EMM, or

    - Deallocated memory might immediately be returned to a memory pool.

- The `defer_cleanup` context manager may behave differently with an EMM - an EMM should be accompanied by documentation of the behaviour of the `defer_cleanup` context manager when it is in use.

    - For example, a pool allocator could always immediately return memory to a pool even when the context manager is in use, but could choose not to free empty pools until `defer_cleanup` is not in use.

Management of other objects

In addition to memory, Numba manages the allocation and deallocation of *events*, *streams*, and modules (a module is a compiled object, which is generated from `@cuda.jit`-ted functions). The management of streams, events, and modules should be unchanged by the presence or absence of an EMM.

Asynchronous allocation / deallocation

An asynchronous memory manager might provide the facility for an allocation or free to take a CUDA stream and execute asynchronously. For freeing, this is unlikely to cause issues since it operates at a layer beneath Python, but for allocations this could be problematic if the user tries to then launch a kernel on the default stream from this asynchronous memory allocation.

The interface described in this proposal will not be required to support asynchronous allocation and deallocation, and as such these use cases will not be considered further. However, nothing in this proposal should preclude the straightforward addition of asynchronous operations in future versions of the interface.

Non-requirements

In order to minimise complexity and constrain this proposal to a reasonable scope, the following will not be supported:

- Using different memory manager implementations for different contexts. All contexts will use the same memory manager implementation - either the Numba internal implementation or an external implementation.

- Changing the memory manager once execution has begun. It is not practical to change the memory manager and retain all allocations. Cleaning up the entire state and then changing to a different memory allocator (rather than starting a new process) appears to be a rather niche use case.

- Any changes to the `__cuda_array_interface__` to further define its semantics, e.g. for acquiring / releasing memory as discussed in Numba Issue #4886 - these are independent, and can be addressed as part of separate proposals.

- Managed memory / UVM is not supported. At present Numba does not support UVM - see Numba Issue #4362 for discussion of support.

Interface for Plugin developers

New classes and functions will be added to `numba.cuda.cudadrv.driver`:

- `BaseCUDAMemoryManager` and `HostOnlyCUDAMemoryManager`: base classes for EMM plugin implementations.
- `set_memory_manager`: a method for registering an external memory manager with Numba.

These will be exposed through the public API, in the `numba.cuda` module. Additionally, some classes that are already part of the *driver* module will be exposed as part of the public API:

- `MemoryPointer`: used to encapsulate information about a pointer to device memory.
- `MappedMemory`: used to hold information about host memory that is mapped into the device address space (a subclass of `MemoryPointer`).
- `PinnedMemory`: used to hold information about host memory that is pinned (a subclass of `mviewbuf.MemAlloc`, a class internal to Numba).

As an alternative to calling the `set_memory_manager` function, an environment variable can be used to set the memory manager. The value of the environment variable should be the name of the module containing the memory manager in its global scope, named `_numba_memory_manager`:

When this variable is set, Numba will automatically use the memory manager from the specified module. Calls to `set_memory_manager` will issue a warning, but otherwise be ignored.

Plugin Base Classes

An EMM plugin is implemented by inheriting from the `BaseCUDAMemoryManager` class, which is defined as:

```python
class BaseCUDAMemoryManager(object, metaclass=ABCMeta):
    @abstractmethod
    def memalloc(self, size):
        """
        Allocate on-device memory in the current context. Arguments:

        - `size`: Size of allocation in bytes

        Returns: a `MemoryPointer` to the allocated memory.
        """

    @abstractmethod
    def memhostalloc(self, size, mapped, portable, wc):
        """
        Allocate pinned host memory. Arguments:

        - `size`: Size of the allocation in bytes
        - `mapped`: Whether the allocated memory should be mapped into the CUDA
                    address space.
        - `portable`: Whether the memory will be considered pinned by all
                      contexts, and not just the calling context.
        - `wc`: Whether to allocate the memory as write-combined.

        Returns a `MappedMemory` or `PinnedMemory` instance that owns the
        allocated memory, depending on whether the region was mapped into
        device memory.
        """
```

(continues on next page)

```python
    @abstractmethod
    def mempin(self, owner, pointer, size, mapped):
        """
        Pin a region of host memory that is already allocated. Arguments:

        - `owner`: An object owning the memory - e.g. a `DeviceNDArray`.
        - `pointer`: The pointer to the beginning of the region to pin.
        - `size`: The size of the region to pin.
        - `mapped`: Whether the region should also be mapped into device memory.

        Returns a `MappedMemory` or `PinnedMemory` instance that refers to the
        allocated memory, depending on whether the region was mapped into device
        memory.
        """

    @abstractmethod
    def initialize(self):
        """
        Perform any initialization required for the EMM plugin to be ready to
        use.
        """

    @abstractmethod
    def get_memory_info(self):
        """
        Returns (free, total) memory in bytes in the context
        """

    @abstractmethod
    def get_ipc_handle(self, memory):
        """
        Return an `IpcHandle` from a GPU allocation. Arguments:

        - `memory`: A `MemoryPointer` for which the IPC handle should be created.
        """

    @abstractmethod
    def reset(self):
        """
        Clear up all memory allocated in this context.
        """

    @abstractmethod
    def defer_cleanup(self):
        """
        Returns a context manager that ensures the implementation of deferred
        cleanup whilst it is active.
        """

    @property
    @abstractmethod
    def interface_version(self):
        """
        Returns an integer specifying the version of the EMM Plugin interface
        supported by the plugin implementation. Should always return 1 for
        implementations described in this proposal.
        """
```

All of the methods of an EMM plugin are called from within Numba - they never need to be invoked directly by a Numba user.

The `initialize` method is called by Numba prior to any memory allocations being requested. This gives the EMM an opportunity to initialize any data structures, etc., that it needs for its normal operations. The method may be called multiple times during the lifetime of the program - subsequent calls should not invalidate or reset the state of the EMM.

The `memalloc`, `memhostalloc`, and `mempin` methods are called when Numba requires an allocation of device or host memory, or pinning of host memory. Device memory should always be allocated in the current context.

`get_ipc_handle` is called when an IPC handle for an array is required. Note that there is no method for closing an IPC handle - this is because the `IpcHandle` object constructed by `get_ipc_handle` contains a `close()` method as part of its definition in Numba, which closes the handle by calling `cuIpcCloseMemHandle`. It is expected that this is sufficient for general use cases, so no facility for customising the closing of IPC handles is provided by the EMM Plugin interface.

`get_memory_info` may be called at any time after `initialize`.

`reset` is called as part of resetting a context. Numba does not normally call reset spontaneously, but it may be called at the behest of the user. Calls to `reset` may even occur before `initialize` is called, so the plugin should be robust against this occurrence.

`defer_cleanup` is called when the `numba.cuda.defer_cleanup` context manager is used from user code.

`interface_version` is called by Numba when the memory manager is set, to ensure that the version of the interface implemented by the plugin is compatible with the version of Numba in use.

Representing pointers

Device Memory

The `MemoryPointer` class is used to represent a pointer to memory. Whilst there are various details of its implementation, the only aspect relevant to EMM plugin development is its initialization. The `__init__` method has the following interface:

```python
class MemoryPointer:
    def __init__(self, context, pointer, size, owner=None, finalizer=None):
```

- `context`: The context in which the pointer was allocated.

- `pointer`: A `ctypes` pointer (e.g. `ctypes.c_uint64`) holding the address of the memory.

- `size`: The size of the allocation in bytes.

- `owner`: The owner is sometimes set by the internals of the class, or used for Numba's internal memory management, but need not be provided by the writer of an EMM plugin - the default of `None` should always suffice.

- `finalizer`: A method that is called when the last reference to the `MemoryPointer` object is released. Usually this will make a call to the external memory management library to inform it that the memory is no longer required, and that it could potentially be freed (though the EMM is not required to free it immediately).

Host Memory

Memory mapped into the CUDA address space (which is created when the `memhostalloc` or `mempin` methods are called with `mapped=True`) is managed using the `MappedMemory` class:

```
class MappedMemory(AutoFreePointer):
    def __init__(self, context, pointer, size, owner, finalizer=None):
```

- `context`: The context in which the pointer was allocated.

- `pointer`: A `ctypes` pointer (e.g. `ctypes.c_void_p`) holding the address of the allocated memory.

- `size`: The size of the allocated memory in bytes.

- `owner`: A Python object that owns the memory, e.g. a `DeviceNDArray` instance.

- `finalizer`: A method that is called when the last reference to the `MappedMemory` object is released. For example, this method could call `cuMemFreeHost` on the pointer to deallocate the memory immediately.

Note that the inheritance from `AutoFreePointer` is an implementation detail and need not concern the developer of an EMM plugin - `MemoryPointer` is higher in the MRO of `MappedMemory`.

Memory that is only in the host address space and has been pinned is represented with the `PinnedMemory` class:

```
class PinnedMemory(mviewbuf.MemAlloc):
    def __init__(self, context, pointer, size, owner, finalizer=None):
```

- `context`: The context in which the pointer was allocated.

- `pointer`: A `ctypes` pointer (e.g. `ctypes.c_void_p`) holding the address of the pinned memory.

- `size`: The size of the pinned region in bytes.

- `owner`: A Python object that owns the memory, e.g. a `DeviceNDArray` instance.

- `finalizer`: A method that is called when the last reference to the `PinnedMemory` object is released. This method could e.g. call `cuMemHostUnregister` on the pointer to unpin the memory immediately.

Providing device memory management only

Some external memory managers will support management of on-device memory but not host memory. To make it easy to implement an EMM plugin using one of these managers, Numba will provide a memory manager class with implementations of the `memhostalloc` and `mempin` methods. An abridged definition of this class follows:

```
class HostOnlyCUDAMemoryManager(BaseCUDAMemoryManager):
    # Unimplemented methods:
    #
    # - memalloc
    # - get_memory_info

    def memhostalloc(self, size, mapped, portable, wc):
        # Implemented.

    def mempin(self, owner, pointer, size, mapped):
        # Implemented.

    def initialize(self):
        # Implemented.
        #
        # Must be called by any subclass when its initialize() method is
        # called.

    def reset(self):
        # Implemented.
```

```
        #
        # Must be called by any subclass when its reset() method is
        # called.

    def defer_cleanup(self):
        # Implemented.
        #
        # Must be called by any subclass when its defer_cleanup() method is
        # called.
```

A class can subclass the `HostOnlyCUDAMemoryManager` and then it only needs to add implementations of methods for on-device memory. Any subclass must observe the following rules:

- If the subclass implements `__init__`, then it must also call `HostOnlyCUDAMemoryManager.__init__`, as this is used to initialize some of its data structures (`self.allocations` and `self.deallocations`).

- The subclass must implement `memalloc` and `get_memory_info`.

- The `initialize` and `reset` methods perform initialisation of structures used by the `HostOnlyCUDAMemoryManager`.

  - If the subclass has nothing to do on initialisation (possibly) or reset (unlikely) then it need not implement these methods.

  - However, if it does implement these methods then it must also call the methods from `HostOnlyCUDAMemoryManager` in its own implementations.

- Similarly if `defer_cleanup` is implemented, it should enter the context provided by `HostOnlyCUDAManager.defer_cleanup()` prior to `yield`ing (or in the `__enter__` method) and release it prior to exiting (or in the `__exit__` method).

Import order

The order in which Numba and the library implementing an EMM Plugin should not matter. For example, if `rmm` were to implement and register an EMM Plugin, then:

```python
from numba import cuda
import rmm
```

and

```python
import rmm
from numba import cuda
```

are equivalent - this is because Numba does not initialize CUDA or allocate any memory until the first call to a CUDA function - neither instantiating and registering an EMM plugin, nor importing `numba.cuda` causes a call to a CUDA function.

Numba as a Dependency

Adding the implementation of an EMM Plugin to a library naturally makes Numba a dependency of the library where it may not have been previously. In order to make the dependency optional, if this is desired, one might conditionally instantiate and register the EMM Plugin like:

```python
try:
    import numba
    from mylib.numba_utils import MyNumbaMemoryManager
    numba.cuda.cudadrv.driver.set_memory_manager(MyNumbaMemoryManager)
except:
    print("Numba not importable – not registering EMM Plugin")
```

so that `mylib.numba_utils`, which contains the implementation of the EMM Plugin, is only imported if Numba is already present. If Numba is not available, then `mylib.numba_utils` (which necessarily imports `numba`), will never be imported.

It is recommended that any library with an EMM Plugin includes at least some environments with Numba for testing with the EMM Plugin in use, as well as some environments without Numba, to avoid introducing an accidental Numba dependency.

Example implementation - A RAPIDS Memory Manager (RMM) Plugin

An implementation of an EMM plugin within the Rapids Memory Manager (RMM) is sketched out in this section. This is intended to show an overview of the implementation in order to support the descriptions above and to illustrate how the plugin interface can be used - different choices may be made for a production-ready implementation.

The plugin implementation consists of additions to python/rmm/rmm.py:

```python
# New imports:
from contextlib import context_manager
# RMM already has Numba as a dependency, so these imports need not be guarded
# by a check for the presence of numba.
from numba.cuda import (HostOnlyCUDAMemoryManager, MemoryPointer, IpcHandle,
                        set_memory_manager)


# New class implementing the EMM Plugin:
class RMMNumbaManager(HostOnlyCUDAMemoryManager):
    def memalloc(self, size):
        # Allocates device memory using RMM functions. The finalizer for the
        # allocated memory calls back to RMM to free the memory.
        addr = librmm.rmm_alloc(bytesize, 0)
        ctx = cuda.current_context()
        ptr = ctypes.c_uint64(int(addr))
        finalizer = _make_finalizer(addr, stream)
        return MemoryPointer(ctx, ptr, size, finalizer=finalizer)

    def get_ipc_handle(self, memory):
        """
        Get an IPC handle for the memory with offset modified by the RMM memory
        pool.
        """
        # This implementation provides a functional implementation and illustrates
        # what get_ipc_handle needs to do, but it is not a very "clean"
        # implementation, and it relies on borrowing bits of Numba internals to
        # initialise ipchandle.
        #
        # A more polished implementation might make use of additional functions in
        # the RMM C++ layer for initialising IPC handles, and not use any Numba
        # internals.
        ipchandle = (ctypes.c_byte * 64)()  # IPC handle is 64 bytes
```

<div align="right">(continues on next page)</div>

```python
            cuda.cudadrv.memory.driver_funcs.cuIpcGetMemHandle(
                ctypes.byref(ipchandle),
                memory.owner.handle,
            )
            source_info = cuda.current_context().device.get_device_identity()
            ptr = memory.device_ctypes_pointer.value
            offset = librmm.rmm_getallocationoffset(ptr, 0)
            return IpcHandle(memory, ipchandle, memory.size, source_info,
                             offset=offset)

    def get_memory_info(self):
        # Returns a tuple of (free, total) using RMM functionality.
        return get_info() # Function defined in rmm.py

    def initialize(self):
        # Nothing required to initialize RMM here, but this method is added
        # to illustrate that the super() method should also be called.
        super().initialize()

    @contextmanager
    def defer_cleanup(self):
        # Does nothing to defer cleanup - a full implementation may choose to
        # implement a different policy.
        with super().defer_cleanup():
            yield

    @property
    def interface_version(self):
        # As required by the specification
        return 1

# The existing _make_finalizer function is used by RMMNumbaManager:
def _make_finalizer(handle, stream):
    """
    Factory to make the finalizer function.
    We need to bind *handle* and *stream* into the actual finalizer, which
    takes no args.
    """

    def finalizer():
        """
        Invoked when the MemoryPointer is freed
        """
        librmm.rmm_free(handle, stream)

    return finalizer

# Utility function register `RMMNumbaManager` as an EMM:
def use_rmm_for_numba():
    set_memory_manager(RMMNumbaManager)

# To support `NUMBA_CUDA_MEMORY_MANAGER=rmm`:
_numba_memory_manager = RMMNumbaManager
```

Example usage

A simple example that configures Numba to use RMM for memory management and creates a device array is as follows:

```python
# example.py
import rmm
import numpy as np

from numba import cuda

rmm.use_rmm_for_numba()

a = np.zeros(10)
d_a = cuda.to_device(a)
del(d_a)
print(rmm.csv_log())
```

Running this should result in output similar to the following:

Note that there is some scope for improvement in RMM for detecting the line number at which the allocation / free occurred, but this is outside the scope of the example in this proposal.

Setting the memory manager through the environment

Rather than calling `rmm.use_rmm_for_numba()` in the example above, the memory manager could also be set to use RMM globally with an environment variable, so the Python interpreter is invoked to run the example as:

Numba internal changes

This section is intended primarily for Numba developers - those with an interest in the external interface for implementing EMM plugins may choose to skip over this section.

Current model / implementation

At present, memory management is implemented in the `Context` class. It maintains lists of allocations and deallocations:

- `allocations` is a `numba.core.utils.UniqueDict`, created at context creation time.
- `deallocations` is an instance of the `_PendingDeallocs` class, and is created when `Context. prepare_for_use()` is called.

These are used to track allocations and deallocations of:

- Device memory
- Pinned memory
- Mapped memory
- Streams
- Events
- Modules

The `_PendingDeallocs` class implements the deferred deallocation strategy - cleanup functions (such as `cuMemFree`) for the items above are added to its list of pending deallocations by the finalizers of objects representing allocations. These finalizers are run when the objects owning them are garbage-collected by the Python interpreter. When the addition of a new cleanup function to the deallocation list causes the number or size of pending deallocations to exceed a configured ratio, the `_PendingDeallocs` object runs deallocators for all items it knows about and then clears its internal pending list.

See *Deallocation Behavior* for more details of this implementation.

Proposed changes

This section outlines the major changes that will be made to support the EMM plugin interface - there will be various small changes to other parts of Numba that will be required in order to adapt to these changes; an exhaustive list of these is not provided.

Context changes

The `numba.cuda.cudadrv.driver.Context` class will no longer directly allocate and free memory. Instead, the context will hold a reference to a memory manager instance, and its memory allocation methods will call into the memory manager, e.g.:

```python
def memalloc(self, size):
    return self.memory_manager.memalloc(size)

def memhostalloc(self, size, mapped=False, portable=False, wc=False):
    return self.memory_manager.memhostalloc(size, mapped, portable, wc)

def mempin(self, owner, pointer, size, mapped=False):
    if mapped and not self.device.CAN_MAP_HOST_MEMORY:
        raise CudaDriverError("%s cannot map host memory" % self.device)
    return self.memory_manager.mempin(owner, pointer, size, mapped)

def prepare_for_use(self):
    self.memory_manager.initialize()

def get_memory_info(self):
    self.memory_manager.get_memory_info()

def get_ipc_handle(self, memory):
    return self.memory_manager.get_ipc_handle(memory)

def reset(self):
    # ... Already-extant reset logic, plus:
    self._memory_manager.reset()
```

The `memory_manager` member is initialised when the context is created.

The `memunpin` method (not shown above but currently exists in the `Context` class) has never been implemented - it presently raises a `NotImplementedError`. This method arguably un-needed - pinned memory is immediately unpinned by its finalizer, and unpinning before a finalizer runs would invalidate the state of `PinnedMemory` objects for which references are still held. It is proposed that this is removed when making the other changes to the `Context` class.

The `Context` class will still instantiate `self.allocations` and `self.deallocations` as before - these will still be used by the context to manage the allocations and deallocations of events, streams, and modules, which are not handled by the EMM plugin.

New components of the `driver` module

- `BaseCUDAMemoryManager`: An abstract class, as defined in the plugin interface above.

- `HostOnlyCUDAMemoryManager`: A subclass of `BaseCUDAMemoryManager`, with the logic from `Context.memhostalloc` and `Context.mempin` moved into it. This class will also create its own `allocations` and `deallocations` members, similarly to how the `Context` class creates them. These are used to manage the allocations and deallocations of pinned and mapped host memory.

- `NumbaCUDAMemoryManager`: A subclass of `HostOnlyCUDAMemoryManager`, which also contains an implementation of `memalloc` based on that presently existing in the `Context` class. This is the default memory manager, and its use preserves the behaviour of Numba prior to the addition of the EMM plugin interface - that is, all memory allocation and deallocation for Numba arrays is handled within Numba.

  - This class shares the `allocations` and `deallocations` members with its parent class `HostOnlyCUDAMemoryManager`, and it uses these for the management of device memory that it allocates.

- The `set_memory_manager` function, which sets a global pointing to the memory manager class. This global initially holds `NumbaCUDAMemoryManager` (the default).

Staged IPC

Staged IPC should not take ownership of the memory that it allocates. When the default internal memory manager is in use, the memory allocated for the staging array is already owned. When an EMM plugin is in use, it is not legitimate to take ownership of the memory.

This change can be made by applying the following small patch, which has been tested to have no effect on the CUDA test suite:

```diff
diff --git a/numba/cuda/cudadrv/driver.py b/numba/cuda/cudadrv/driver.py
index 7832955..f2c1352 100644
--- a/numba/cuda/cudadrv/driver.py
+++ b/numba/cuda/cudadrv/driver.py
@@ -922,7 +922,11 @@ class _StagedIpcImpl(object):
         with cuda.gpus[srcdev.id]:
             impl.close()

-        return newmem.own()
+        return newmem
```

Testing

Alongside the addition of appropriate tests for new functionality, there will be some refactoring of existing tests required, but these changes are not substantial. Tests of the deallocation strategy (e.g. `TestDeallocation`, `TestDeferCleanup`) will need to be modified to ensure that they are examining the correct set of deallocations. When an EMM plugin is in use, they will need to be skipped.

Prototyping / experimental implementation

Some prototype / experimental implementations have been produced to guide the designs presented in this document. The current implementations can be found in:

- Numba branch: https://github.com/gmarkall/numba/tree/grm-numba-nbep-7.

- RMM branch: https://github.com/gmarkall/rmm/tree/grm-numba-nbep-7.

- CuPy implementation: https://github.com/gmarkall/nbep-7/blob/master/nbep7/cupy_mempool.py - uses an unmodified CuPy.

  - See CuPy memory management docs.

Current implementation status

RMM Plugin

For a minimal example, a simple allocation and free using RMM works as expected. For the example code (similar to the RMM example above):

```python
import rmm
import numpy as np

from numba import cuda

rmm.use_rmm_for_numba()

a = np.zeros(10)
d_a = cuda.to_device(a)
del(d_a)
print(rmm.csv_log())
```

We see the following output:

This output is similar to the expected output from the example usage presented above (though note that the pointer addresses and timestamps vary compared to the example), and provides some validation of the example use case.

CuPy Plugin

```python
from nbep7.cupy_mempool import use_cupy_mm_for_numba
import numpy as np

from numba import cuda

use_cupy_mm_for_numba()

a = np.zeros(10)
d_a = cuda.to_device(a)
del(d_a)
```

The prototype CuPy plugin has somewhat primitive logging, so we see the output:

Numba CUDA Unit tests

As well as providing correct execution of a simple example, all relevant Numba CUDA unit tests also pass with the prototype branch, for both the internal memory manager and the RMM EMM Plugin.

RMM

The unit test suite can be run with the RMM EMM Plugin with:

A summary of the unit test suite output is:

When running with the built-in Numba memory management, the output is:

i.e. the changes for using an external memory manager do not break the built-in Numba memory management. There are an additional 6 skipped tests, from:

- `TestDeallocation`: skipped as it specifically tests Numba's internal deallocation strategy.

- `TestDeferCleanup`: skipped as it specifically tests Numba's implementation of deferred cleanup.

- `TestCudaArrayInterface.test_ownership`: skipped as Numba does not own memory when an EMM Plugin is used, but ownership is assumed by this test case.

CuPy

The test suite can be run with the CuPy plugin using:

This plugin implementation is presently more primitive than the RMM implementation, and results in some errors with the unit test suite:

The 8 errors are due to a lack of implementation of `get_ipc_handle` in the CuPy EMM Plugin implementation. It is expected that this implementation will be re-visited and completed so that CuPy can be used stably as an allocator for Numba in the future.

8.2 Other proposals

8.2.1 NBEP 2: Extension points

Author Antoine Pitrou

Date July 2015

Status Draft

Implementing new types or functions in Numba requires hooking into various mechanisms along the compilation chain (and potentially outside of it). This document aims, first, at examining the current ways of doing so and, second, at making proposals to make extending easier.

If some of the proposals are implemented, we should first strive to use and exercise them internally, before exposing the APIs to the public.

Note: This document doesn't cover CUDA or any other non-CPU backend.

High-level API

There is currently no high-level API, making some use cases more complicated than they should be.

Proposed changes

Dedicated module

We propose the addition of a `numba.extending` module exposing the main APIs useful for extending Numba.

Implementing a function

We propose the addition of a `@overload` decorator allowing the implementation of a given function for use in *nopython mode*. The overloading function has the same formal signature as the implemented function, and receives the actual argument types. It should return a Python function implementing the overloaded function for the given types.

The following example implements *numpy.where()* with this approach.

```python
import numpy as np

from numba.core import types
from numba.extending import overload

@overload(np.where)
def where(cond, x, y):
    """
    Implement np.where().
    """
    # Choose implementation based on argument types.
    if isinstance(cond, types.Array):
        # Array where() => return an array of the same shape
        if all(ty.layout == 'C' for ty in (cond, x, y)):
            def where_impl(cond, x, y):
                """
                Fast implementation for C-contiguous arrays
                """
                shape = cond.shape
                if x.shape != shape or y.shape != shape:
                    raise ValueError("all inputs should have the same shape")
                res = np.empty_like(x)
                cf = cond.flat
                xf = x.flat
                yf = y.flat
                rf = res.flat
                for i in range(cond.size):
                    rf[i] = xf[i] if cf[i] else yf[i]
                return res
        else:
            def where_impl(cond, x, y):
                """
                Generic implementation for other arrays
                """
                shape = cond.shape
                if x.shape != shape or y.shape != shape:
                    raise ValueError("all inputs should have the same shape")
                res = np.empty_like(x)
                for idx, c in np.ndenumerate(cond):
                    res[idx] = x[idx] if c else y[idx]
                return res
```

```python
    else:
        def where_impl(cond, x, y):
            """
            Scalar where() => return a 0-dim array
            """
            scal = x if cond else y
            return np.full_like(scal, scal)

    return where_impl
```

It is also possible to implement functions already known to Numba, to support additional types. The following example implements the built-in function `len()` for tuples with this approach:

```python
@overload(len)
def tuple_len(x):
    if isinstance(x, types.BaseTuple):
        # The tuple length is known at compile-time, so simply reify it
        # as a constant.
        n = len(x)
        def len_impl(x):
            return n
        return len_impl
```

Implementing an attribute

We propose the addition of a `@overload_attribute` decorator allowing the implementation of an attribute getter for use in *nopython mode*.

The following example implements the `.nbytes` attribute on Numpy arrays:

```python
@overload_attribute(types.Array, 'nbytes')
def array_nbytes(arr):
    def get(arr):
        return arr.size * arr.itemsize
    return get
```

Note: The overload_attribute() signature allows for expansion to also define setters and deleters, by letting the decorated function return a `getter, setter, deleter` tuple instead of a single `getter`.

Implementing a method

We propose the addition of a `@overload_method` decorator allowing the implementation of an instance method for use in *nopython mode*.

The following example implements the `.take()` method on Numpy arrays:

```python
@overload_method(types.Array, 'take')
def array_take(arr, indices):
    if isinstance(indices, types.Array):
        def take_impl(arr, indices):
```

```
            n = indices.shape[0]
            res = np.empty(n, arr.dtype)
            for i in range(n):
                res[i] = arr[indices[i]]
            return res
        return take_impl
```

Exposing a structure member

We propose the addition of a `make_attribute_wrapper()` function exposing an internal field as a visible read-only attribute, for those types backed by a `StructModel` data model.

For example, assuming `PdIndexType` is the Numba type of pandas indices, here is how to expose the underlying Numpy array as a `._data` attribute:

```
@register_model(PdIndexType)
class PdIndexModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ('values', fe_type.as_array),
            ]
        models.StructModel.__init__(self, dmm, fe_type, members)

make_attribute_wrapper(PdIndexType, 'values', '_data')
```

Typing

Numba types

Numba's standard types are declared in `numba.types`. To declare a new type, one subclasses the base `Type` class or one of its existing abstract subclasses, and implements the required functionality.

Proposed changes

No change required.

Type inference on values

Values of a new type need to be type-inferred if they can appear as function arguments or constants. The core machinery is in `numba.typing.typeof`.

In the common case where some Python class or classes map exclusively to the new type, one can extend a generic function to dispatch on said classes, e.g.:

```
from numba.typing.typeof import typeof_impl

@typeof_impl(MyClass)
def _typeof_myclass(val, c):
    if "some condition":
        return MyType(...)
```

The `typeof_impl` specialization must return a Numba type instance, or None if the value failed typing.

(when one controls the class being type-inferred, an alternative to `typeof_impl` is to define a `_numba_type_` property on the class)

In the rarer case where the new type can denote various Python classes that are impossible to enumerate, one must insert a manual check in the fallback implementation of the `typeof_impl` generic function.

Proposed changes

Allow people to define a generic hook without monkeypatching the fallback implementation.

Fast path for type inference on function arguments

Optionally, one may want to allow a new type to participate in the fast type resolution (written in C code) to minimize function call overhead when a JIT-compiled function is called with the new type. One must then insert the required checks and implementation in the `_typeof.c` file, presumably inside the `compute_fingerprint()` function.

Proposed changes

None. Adding generic hooks to C code embedded in a C Python extension is too delicate a change.

Type inference on operations

Values resulting from various operations (function calls, operators, etc.) are typed using a set of helpers called "templates". One can define a new template by subclass one of the existing base classes and implement the desired inference mechanism. The template is explicitly registered with the type inference machinery using a decorator.

The `ConcreteTemplate` base class allows one to define inference as a set of supported signatures for a given operation. The following example types the modulo operator:

```python
@builtin
class BinOpMod(ConcreteTemplate):
    key = "%"
    cases = [signature(op, op, op)
                for op in sorted(types.signed_domain)]
    cases += [signature(op, op, op)
                 for op in sorted(types.unsigned_domain)]
    cases += [signature(op, op, op) for op in sorted(types.real_domain)]
```

(note that type *instances* are used in the signatures, severely limiting the amount of genericity that can be expressed)

The `AbstractTemplate` base class allows to define inference programmatically, giving it full flexibility. Here is a simplistic example of how tuple indexing (i.e. the `__getitem__` operator) can be expressed:

```python
@builtin
class GetItemUniTuple(AbstractTemplate):
    key = "getitem"

    def generic(self, args, kws):
        tup, idx = args
        if isinstance(tup, types.UniTuple) and isinstance(idx, types.Integer):
            return signature(tup.dtype, tup, idx)
```

The `AttributeTemplate` base class allows to type the attributes and methods of a given type. Here is an example, typing the `.real` and `.imag` attributes of complex numbers:

```
@builtin_attr
class ComplexAttribute(AttributeTemplate):
    key = types.Complex

    def resolve_real(self, ty):
        return ty.underlying_float

    def resolve_imag(self, ty):
        return ty.underlying_float
```

Note: `AttributeTemplate` only works for getting attributes. Setting an attribute's value is hardcoded in `numba.typeinfer`.

The `CallableTemplate` base class offers an easier way to parse flexible function signatures, by letting one define a callable that has the same definition as the function being typed. For example, here is how one could hypothetically type Python's `sorted` function if Numba supported lists:

```
@builtin
class Sorted(CallableTemplate):
    key = sorted

    def generic(self):
        def typer(iterable, key=None, reverse=None):
            if reverse is not None and not isinstance(reverse, types.Boolean):
                return
            if key is not None and not isinstance(key, types.Callable):
                return
            if not isinstance(iterable, types.Iterable):
                return
            return types.List(iterable.iterator_type.yield_type)

        return typer
```

(note you can return just the function's return type instead of the full signature)

Proposed changes

Naming of the various decorators is quite vague and confusing. We propose renaming `@builtin` to `@infer`, `@builtin_attr` to `@infer_getattr` and `builtin_global` to `infer_global`.

The two-step declaration for global values is a bit verbose, we propose simplifying it by allowing the use of `infer_global` as a decorator:

```
@infer_global(len)
class Len(AbstractTemplate):
    key = len

    def generic(self, args, kws):
        assert not kws
        (val,) = args
        if isinstance(val, (types.Buffer, types.BaseTuple)):
            return signature(types.intp, val)
```

The class-based API can feel clumsy, we can add a functional API for some of the template kinds:

```
@type_callable(sorted)
def type_sorted(context):
    def typer(iterable, key=None, reverse=None):
        # [same function as above]

    return typer
```

Code generation

Concrete representation of values of a Numba type

Any concrete Numba type must be able to be represented in LLVM form (for variable storage, argument passing, etc.). One defines that representation by implementing a datamodel class and registering it with a decorator. Datamodel classes for standard types are defined in `numba.datamodel.models`.

Proposed changes

No change required.

Conversion between types

Implicit conversion between Numba types is currently implemented as a monolithic sequence of choices and type checks in the `BaseContext.cast()` method. To add a new implicit conversion, one appends a type-specific check in that method.

Boolean evaluation is a special case of implicit conversion (the destination type being `types.Boolean`).

Note: Explicit conversion is seen as a regular operation, e.g. a constructor call.

Proposed changes

Add a generic function for implicit conversion, with multiple dispatch based on the source and destination types. Here is an example showing how to write a float-to-integer conversion:

```
@lower_cast(types.Float, types.Integer)
def float_to_integer(context, builder, fromty, toty, val):
    lty = context.get_value_type(toty)
    if toty.signed:
        return builder.fptosi(val, lty)
    else:
        return builder.fptoui(val, lty)
```

Implementation of an operation

Other operations are implemented and registered using a set of generic functions and decorators. For example, here is how lookup for a the `.ndim` attribute on Numpy arrays is implemented:

```
@builtin_attr
@impl_attribute(types.Kind(types.Array), "ndim", types.intp)
def array_ndim(context, builder, typ, value):
    return context.get_constant(types.intp, typ.ndim)
```

And here is how calling `len()` on a tuple value is implemented:

```
@builtin
@implement(types.len_type, types.Kind(types.BaseTuple))
def tuple_len(context, builder, sig, args):
    tupty, = sig.args
    retty = sig.return_type
    return context.get_constant(retty, len(tupty.types))
```

Proposed changes

Review and streamline the API. Drop the requirement to write `types.Kind(...)` explicitly. Remove the separate `@implement` decorator and rename `@builtin` to `@lower_builtin`, `@builtin_attr` to `@lower_getattr`, etc.

Add decorators to implement `setattr()` operations, named `@lower_setattr` and `@lower_setattr_generic`.

Conversion from / to Python objects

Some types need to be converted from or to Python objects, if they can be passed as function arguments or returned from a function. The corresponding boxing and unboxing operations are implemented using a generic function. The implementations for standard Numba types are in `numba.targets.boxing`. For example, here is the boxing implementation for a boolean value:

```
@box(types.Boolean)
def box_bool(c, typ, val):
    longval = c.builder.zext(val, c.pyapi.long)
    return c.pyapi.bool_from_long(longval)
```

Proposed changes

Change the implementation signature from `(c, typ, val)` to `(typ, val, c)`, to match the one chosen for the `typeof_impl` generic function.

8.2.2 NBEP 3: JIT Classes

Author Siu Kwan Lam

Date Dec 2015

Status Draft

Introduction

Numba does not yet support user-defined classes. Classes provide useful abstraction and promote modularity when used right. In the simplest sense, a class specifies the set of data and operations as attributes and methods, respectively. A class instance is an instantiation of that class. This proposal will focus on supporting this simple usecase of classes–with just attributes and methods. Other features, such as class methods, static methods, and inheritance are deferred to another proposal, but we believe these features can be easily implemented given the foundation described here.

Proposal: jit-classes

A JIT-classes is more restricted than a Python class. We will focus on the following operations on a class and its instance:

- Instantiation: create an instance of a class using the class object as the constructor: `cls(*args, **kwargs)`

- Destruction: remove resources allocated during instantiation and release all references to other objects.

- Attribute access: loading and storing attributes using `instance.attr` syntax.

- Method access: loading methods using `instance.method` syntax.

With these operations, a class object (not the instance) does not need to be materialize. Using the class object as a constructor is fully resolved (a runtime implementation is picked) during the typing phase in the compiler. This means **a class object will not be first class**. On the other hand, implementating a first-class class object will require an "interface" type, or the type of class.

The instantiation of a class will allocate resources for storing the data attributes. This is described in the "Storage model" section. Methods are never stored in the instance. They are information attached to the class. Since a class object only exists in the type domain, the methods will also be fully resolved at the typing phase. Again, numba do not have first-class function value and each function type maps uniquely to each function implementation (this needs to be changed to support function value as argument).

A class instance can contain other NRT reference-counted object as attributes. To properly clean up an instance, a destructor is called when the reference count of the instance is dropped to zero. This is described in the "Reference count and descructor" section.

Storage model

For compatibility with C, attributes are stored in a simple plain-old-data structure. Each attribute are stored in a user-defined order in a padded (for proper alignment), contiguous memory region. An instance that contains three fields of int32, float32, complex64 will be compatible with the following C structure:

```
struct {
    int32    field0;
    float32  field1;
    complex64 field2;
};
```

This will also be comptabile with an aligned numpy structure dtype.

Methods

Methods are regular function that can be bounded to an instance. They can be compiled as regular function by numba. The operation `getattr(instance, name)` (getting an attribute `name` from `instance`) binds the instance to the requested method at runtime.

The special `__init__` method is also handled like regular functions.

`__del__` is not supported at this time.

Reference count and destructor

An instance of jit-class is reference-counted by NRT. Since it may contain other NRT tracked object, it must call a destructor when its reference count dropped to zero. The destructor will decrement the reference count of all attributes by one.

At this time, there is no support for user defined `__del__` method.

Proper cleanup for cyclic reference is not handled at this time. Cycles will cause memory leak.

Type inference

So far we have not described the type of the attributes or the methods. Type information is necessary to materailize the instance (e.g. allocate the storage). The simplest way is to let user provide the type of each attributes as well as the ordering; for instance:

```
dct = OrderedDict()
dct['x'] = int32
dct['y'] = float32
```

Allowing user to supply an ordered dictionary will provide the name, ordering and types of the attributes. However, this statically typed semantic is not as flexible as the Python semantic which behaves like a generic class.

Inferring the type of attributes is difficult. In a previous attempt to implement JIT classes, the `__init__` method is specialized to capture the type stored into the attributes. Since the method can contain arbitrary logic, the problem can become a dependent typing problem if types are assigned conditionally depending on the value. (Very few languages implement dependent typing and those that does are mostly theorem provers.)

Example: typing function using an OrderedDict

```
spec = OrderedDict()
spec['x'] = numba.int32
spec['y'] = numba.float32

@jitclass(spec)
class Vec(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def add(self, dx, dy):
        self.x += dx
        self.y += dy
```

Example: typing function using a list of 2-tuples

```
spec = [('x', numba.int32),
        ('y', numba.float32)]

@jitclass(spec)
class Vec(object):
    ...
```

Creating multiple jitclasses from a single class object

The *jitclass(spec)* decorator creates a new jitclass type even when applied to the same class object and the same type specification.

```
class Vec(object):
  ...

Vec1 = jitclass(spec)(Vec)
Vec2 = jitclass(spec)(Vec)
# Vec1 and Vec2 are two different jitclass types
```

Usage from the Interpreter

When constructing a new instance of a jitclass, a "box" is created that wraps the underlying jitclass instance from numba. Attributes and methods are accessible from the interpreter. The actual implementation will be in numba compiled code. Any Python object is converted to its native representation for consumption in numba. Similarly, the returned value is converted to its Python representation. As a result, there may be overhead in manipulating jitclass instances in the interpreter. This overhead is minimal and should be easily amortized by more efficient computation in the compiled methods.

Support for property, staticmethod and classmethod

The use of `property` is accepted for getter and setter only. Deleter is not supported.

The use of `staticmethod` is not supported.

The use of `classmethod` is not supported.

Inheritance

Class inhertance is not considered in this proposal. The only accepted base class for a jitclass is *object*.

Supported targets

Only the CPU target (including the parallel target) is supported. GPUs (e.g. CUDA and HSA) targets are supported via an immutable version of the jitclass instance, which will be described in a separate NBEP.

Other properties

Given:

```
spec = [('x', numba.int32),
        ('y', numba.float32)]

@jitclass(spec)
class Vec(object):
    ...
```

- `isinstance(Vec(1, 2), Vec)` is True.
- `type(Vec(1, 2))` may not be `Vec`.

Future enhancements

This proposal has only described the basic semantic and functionality of a jitclass. Additional features will be described in future enhancement proposals.

8.2.3 NBEP 4: Defining C callbacks

> **Author** Antoine Pitrou
>
> **Date** April 2016
>
> **Status** Draft

Interfacing with some native libraries (for example written in C or C++) can necessitate writing native callbacks to provide business logic to the library. Some Python-facing libraries may also provide the alternative of passing a ctypes-wrapped native callback instead of a Python callback for better performance. A simple example is the `scipy.integrate` package where the user passes the function to be integrated as a callback.

Users of those libraries may want to benefit from the performance advantage of running purely native code, while writing their code in Python. This proposal outlines a scheme to provide such a functionality in Numba.

Basic usage

We propose adding a new decorator, `@cfunc`, importable from the main package. This decorator allows defining a callback as in the following example:

```
from numba import cfunc
from numba.types import float64

# A callback with the C signature `double(double)`

@cfunc(float64(float64), nopython=True)
def integrand(x):
    return 1 / x
```

The `@cfunc` decorator returns a "C function" object holding the resources necessary to run the given compiled function (for example its LLVM module). This object has several attributes and methods:

- the `ctypes` attribute is a ctypes function object representing the native function.

- the `address` attribute is the address of the native function code, as an integer (note this can also be computed from the `ctypes` attribute).

- the `native_name` attribute is the symbol under which the function can be looked up inside the current process.

- the `inspect_llvm()` method returns the IR for the LLVM module in which the function is compiled. It is expected that the `native_name` attribute corresponds to the function's name in the LLVM IR.

The general signature of the decorator is `cfunc(signature, **options)`.

The `signature` must specify the argument types and return type of the function using Numba types. In contrary to `@jit`, the return type cannot be omitted.

The `options` are keyword-only parameters specifying compilation options. We are expecting that the standard `@jit` options (`nopython`, `forceobj`, `cache`) can be made to work with `@cfunc`.

Calling from Numba-compiled functions

While the intended use is to pass a callback's address to foreign C code expecting a function pointer, it should be made possible to call the C callback from a Numba-compiled function.

Passing array data

Native platform ABIs as used by C or C++ don't have the notion of a shaped array as in Numpy. One common solution is to pass a raw data pointer and one or several size arguments (depending on dimensionality). Numba must provide a way to rebuild an array view of this data inside the callback.

```python
from numba import cfunc, carray
from numba.types import float64, CPointer, void, intp

# A callback with the C signature `void(double *, double *, size_t)`

@cfunc(void(CPointer(float64), CPointer(float64), intp))
def invert(in_ptr, out_ptr, n):
    in_ = carray(in_ptr, (n,))
    out = carray(out_ptr, (n,))
    for i in range(n):
        out[i] = 1 / in_[i]
```

The `carray` function takes (`pointer, shape, dtype`) arguments (`dtype` being optional) and returns a C-layout array view over the data *pointer*, with the given *shape* and *dtype*. *pointer* must be a ctypes pointer object (not a Python integer). The array's dimensionality corresponds to the *shape* tuple's length. If *dtype* is not given, the array's dtype corresponds to the *pointer*'s pointee type.

The `farray` function is similar except that it returns a F-layout array view.

Error handling

There is no standard mechanism in C for error reporting. Unfortunately, Numba currently doesn't handle `try.. except` blocks, which makes it more difficult for the user to implement the required error reporting scheme. The current stance of this proposal is to let users guard against invalid arguments where necessary, and do whatever is required to inform the caller of the error.

Based on user feedback, we can later add support for some error reporting schemes, such as returning an integer error code depending on whether an exception was raised, or setting `errno`.

Deferred topics

Ahead-of-Time compilation

This proposal doesn't make any provision for AOT compilation of C callbacks. It would probably necessitate a separate API (a new method on the `numba.pycc.CC` object), and the implementation would require exposing a subset of the C function object's functionality from the compiled C extension module.

Opaque data pointers

Some libraries allow passing an opaque data pointer (`void *`) to a user-provided callback, to provide any required context for execution of the callback. Taking advantage of this functionality would require adding specific support in Numba, for example the ability to do generic conversion from `types.voidptr` and to take the address of a Python-facing `jitclass` instance.

8.2.4 NBEP 5: Type Inference

Author Siu Kwan Lam

Date Sept 2016

Status Draft

This document describes the current type inference implementation in numba.

Introduction

Numba uses type information to ensure that every variable in the user code can be correctly lowered (translated into a low-level representation). The type of a variable describes the set of valid operations and available attributes. Resolving this information during compilation avoids the overhead of type checking and dispatching at runtime. However, Python is dynamically typed and the user does not declare variable types. Since type information is absent, we use type inference to reconstruct the missing information.

Numba Type Semantic

Type inference operates on *Numba IR*, a mostly static-single-assignment (SSA) encoding of the Python bytecode. Conceptually, all intermediate values in the Python code are explicitly assigned to a variable in the IR. Numba enforces that each IR variable to have one type only. A user variable (from the Python source code) can be mapped to multiple variables in the IR. They are *versions* of a variable. Each time a user variable is assigned to, a new version is created. From that point, all subsequent references will use the new version. The user variable *evolves* as the function logic updates its type. Merge points (e.g. subsequent block to an if-else, the loop body, etc..) in the control flow need extra care. At each merge point, a new version is implicitly created to merge the different variable versions from the incoming paths. The merging of the variable versions may translate into an implicit cast.

Numba uses function overloading to emulate Python duck-typing. The type of a function can contain multiple call signatures that accept different argument types and yield different return types. The process to decide the best signature for an overloaded function is called *overload resolution*. Numba partially implements the C++ overload resolution scheme (ISOCPP 13.3 Overload Resolution). The scheme uses a "best fit" algorithm by ranking each argument symmetrically. The five possible rankings in increasing order of penalty are:

- *Exact*: the expected type is the same as the actual type.

- *Promotion*: the actual type can be upcast to the expected type by extending the precision without changing the behavior.

- *Safe conversion*: the actual type can be cast to the expected type by changing the type without losing information.

- *Unsafe conversion*: the actual type can be cast to the expected type by changing the type or downcasting the type even if it is imprecise.

- *No match*: no valid operation can convert the actual type to the expected type.

It is possible to have an ambiguous resolution. For example, a function with signatures `(int16, int32)` and `(int32, int16)` can become ambiguous if presented with the argument types `(int32, int32)`, because demoting either argument to `int16` is equally "fit". Fortunately, numba can usually resolve such ambiguity by compiling a new version with the exact signature `(int32, int32)`. When compilation is disabled and there are multiple signatures with equal fit, an exception is raised.

Type Inference

The type inference in numba has three important components—type variable, constraint network, and typing context.

- The *typing context* provides all the type information and typing related operations, including the logic for type unification, and the logic for typing of global and constant values. It defines the semantic of the language that can be compiled by numba.

- A *type variable* holds the type of each variable (in the Numba IR). Conceptually, it is initialized to the universal type and, as it is re-assigned, it stores a common type by unifying the new type with the existing type. The common type must be able to represent values of the new type and the existing type. Type conversion is applied as necessary and precision loss is accepted for usability reason.

- The *constraint network* is a dependency graph built from the IR. Each node represents an operation in the Numba IR and updates at least one type variable. There may be cycles due to loops in user code.

The type inference process starts by seeding the argument types. These initial types are propagated in the constraint network, which eventually fills all the type variables. Due to cycles in the network, the process repeats until all type variables converge or it fails with undecidable types.

Type unification always returns a more "general" (quoted because unsafe conversion is allowed) type. Types will converge to the least "general" type that can represent all possible values that the variable can hold. Since unification will never move down the type hierarchy and there is a single top type, the universal type—`object`, the type inference is guaranteed to converge.

A failure in type inference can be caused by two reasons. The first reason is user error due to incorrect use of a type. This type of error will also trigger an exception in regular python execution. The second reason is due to the use of an unsupported feature, but the code is otherwise valid in regular python execution. Upon an error, the type inference will set all types to the object type. As a result, numba will fallback to *object-mode*.

Since functions can be overloaded, the type inference needs to decide the type signature used at each call site. The overload resolution is applied to all known overload versions of the callee function described in *call-templates*. A call-template can either be concrete or abstract. A concrete call-template defines a fixed list of all possible signatures. An abstract call-template defines the logic to compute the accepted signature and it is used to implement generic functions.

Numba-compiled functions are generic functions due to their ability to compile new versions. When it sees a new set of argument types, it triggers type inference to validate and determine the return type. When there are nested calls for numba-compiled functions, each call-site triggers type inference. This poses a problem to recursive functions because the type inference will also be triggered recursively. Currently, simple single recursion is supported if the signature is user-annotated by the user, which avoids unbound recursion in type inference that will never terminate.

8.2.5 NBEP 6: Typing Recursion

Author Siu Kwan Lam

Date Sept 2016

Status Draft

Introduction

This document proposes an enhancement to the type inference algorithm to support recursion without explicitly annotating the function signature. As a result, the proposal enables numba to type-infer both self-recursive and mutual-recursive functions under some limitations. In practice, these limitions can be easily overcome by specifying a compilation order.

The Current State

Recursion support in numba is currently limited to self-recursion with explicit type annotation for the function. This limitation comes from the inability to determine the return type of a recursive call. This is because the callee is either the current function (for self-recursion) or a parent function (mutual-recursion) and its type inference process has been suspended while waiting for the function-type of its callee. This results in the formation of a cyclic dependency. For example, given a function `foo()` that calls `bar()`, which in turns call `foo()`:

```python
def foo(x):
    if x > 0:
        return bar(x)
    else:
        return 1

def bar(x):
    return foo(x - 1)
```

The type inferrence process of `foo()` depends on that of `bar()`, which depends on `foo()`. Therefore `foo()` depends on itself and the type inference algorithm cannot terminate.

The Solution

The proposed solution has two components:

1. The introduction of a compile-time *callstack* that tracks the compiling functions.

2. The allowance of a partial type inference on functions by leveraging the return type on non-recursive control-flow paths.

The compile-time callstack stores typing information of the functions being compiled. Like an ordinary callstack, it pushes a new record every time a function is "called". Since this occurs at compile-time, a "call" triggers a compilation of the callee.

To detect recursion, the compile-time callstack is searched bottom-up (stack grows downward) for a record that matches the callee. As the record contains a reference to the type inference state, the type inference process can be resumed to determine the return type.

Recall that the type inference process cannot be resumed normally because of the cyclic dependency of the return type. In practice, we can assume that a useful program must have a terminating condition, a path that does not recurse. So, the type inference process can make an initial guess for the return-type at the recursive call by using the return-type determined by the non-recursive paths. This allows type information to propagate on the recursive paths to generate

the final return type, which is used to refine the type information by the subsequent iteration in the type inference process.

The following figure illustrates the compile-time callstack when the compiler reaches the recursive call to `foo()` from `bar()`:

At this time, the type inference process of `foo()` is suspended and that of `bar()` is active. The compiler can see that the callee is already compiling by searching the callstack. Knowing that it is a recursive call, the compiler can resume the type-inference on `foo()` by ignoring the paths that contain recursive calls. This means only the `else` branch is considered and we can easily tell that `foo()` returns an `int` in this case. The compiler will then set the initial return type of `foo()` and `bar()` to `int`. The subsequent type propagation can use this information to complete the type inference of both functions, unifying the return-type of all returning paths.

Limitations

For the proposed type inference algorithm to terminate, it assumes that at least one of the control path leads to a return-statement without undertaking a recursive call. Should this not be the case, the algorithm will raise an exception indicating a potential runaway recursion.

For example:

```python
@jit
def first(x):
    # The recursing call must have a path that is non-recursing.
    if x > 0:
        return second(x)
    else:
        return 1

@jit
def second(x):
    return third(x)

@jit
def third(x):
    return first(x - 1)
```

The `first()` function must be the compiled first for the type inference algorithm to complete successfully. Compiling any other function first will lead to a failure in type inference. The type inference algorithm will treat it as a runaway recursion due to the lack of a non-recursive exit in the recursive callee.

For example, compiling `second()` first will move the recursive call to `first()`. When the compiler tries to resume the type inference process of `second()`, it will fail to find a non-recursive path.

This is a small limitation and can be overcome easily by code restructuring or precompiling in a specific order.

Glossary

ahead-of-time compilation

AOT compilation

AOT Compilation of a function in a separate step before running the program code, producing an on-disk binary object which can be distributed independently. This is the traditional kind of compilation known in languages such as C, C++ or Fortran.

bytecode

Python bytecode The original form in which Python functions are executed. Python bytecode describes a stack-machine executing abstract (untyped) operations using operands from both the function stack and the execution environment (e.g. global variables).

compile-time constant An expression whose value Numba can infer and freeze at compile-time. Global variables and closure variables are compile-time constants.

just-in-time compilation

JIT compilation

JIT Compilation of a function at execution time, as opposed to *ahead-of-time compilation*.

JIT function Shorthand for "a function *JIT-compiled* with Numba using the *@jit* decorator."

loop-lifting

loop-jitting A feature of compilation in *object mode* where a loop can be automatically extracted and compiled in *nopython mode*. This allows functions with operations unsupported in nopython mode to see significant performance improvements if they contain loops with only nopython-supported operations.

lowering The act of translating *Numba IR* into LLVM IR. The term "lowering" stems from the fact that LLVM IR is low-level and machine-specific while Numba IR is high-level and abstract.

nopython mode A Numba compilation mode that generates code that does not access the Python C API. This compilation mode produces the highest performance code, but requires that the native types of all values in the function can be *inferred*. Unless otherwise instructed, the `@jit` decorator will automatically fall back to *object mode* if nopython mode cannot be used.

Numba IR

Numba intermediate representation A representation of a piece of Python code which is more amenable to analysis and transformations than the original Python *bytecode*.

object mode A Numba compilation mode that generates code that handles all values as Python objects and uses the Python C API to perform all operations on those objects. Code compiled in object mode will often run no faster than Python interpreted code, unless the Numba compiler can take advantage of *loop-jitting*.

OptionalType An `OptionalType` is effectively a type union of a `type` and `None`. They typically occur in practice due to a variable being set to `None` and then in a branch the variable being set to some other value. It's often not possible at compile time to determine if the branch will execute so to permit *type inference* to complete, the type of the variable becomes the union of a `type` (from the value) and `None`, i.e. `OptionalType(type)`.

type inference The process by which Numba determines the specialized types of all values within a function being compiled. Type inference can fail if arguments or globals have Python types unknown to Numba, or if functions are used that are not recognized by Numba. Successful type inference is a prerequisite for compilation in *nopython mode*.

typing The act of running *type inference* on a value or operation.

ufunc A NumPy universal function. Numba can create new compiled ufuncs with the *@vectorize* decorator.

reflection In numba, when a mutable container is passed as argument to a nopython function from the Python interpreter, the container object and all its contained elements are converted into nopython values. To match the semantics of Python, any mutation on the container inside the nopython function must be visible in the Python interpreter. To do so, Numba must update the container and its elements and convert them back into Python objects during the transition back into the interpreter.

Not to be confused with Python's "reflection" in the context of binary operators (see https://docs.python.org/3.5/reference/datamodel.html).

Release Notes

10.1 Version 0.51.0 (August 12, 2020)

This release continues to add new features to Numba and also contains a significant number of bug fixes and stability improvements.

Highlights of core feature changes include:

- The compilation chain is now based on LLVM 10 (Valentin Haenel).

- Numba has internally switched to prefer non-literal types over literal ones so as to reduce function over-specialisation, this with view of speeding up compile times (Siu Kwan Lam).

- On the CUDA target: Support for CUDA Toolkit 11, Ampere, and Compute Capability 8.0; Printing of SASS code for kernels; Callbacks to Python functions can be inserted into CUDA streams, and streams are async awaitable; Atomic nanmin and nanmax functions are added; Fixes for various miscompilations and segfaults. (mostly Graham Markall; call backs on streams by Peter Würtz).

Intel also kindly sponsored research and development that lead to some exciting new features:

- Support for heterogeneous immutable lists and heterogeneous immutable string key dictionaries. Also optional initial/construction value capturing for all lists and dictionaries containing literal values (Stuart Archibald).

- A new pass-by-reference mutable structure extension type StructRef (Siu Kwan Lam).

- Object mode blocks are now cacheable, with the side effect of numerous bug fixes and performance improvements in caching. This also permits caching of functions defined in closures (Siu Kwan Lam).

Deprecations to note:

To align with other targets, the argtypes and restypes kwargs to @cuda.jit are now deprecated, the bind kwarg is also deprecated. Further the target kwarg to the numba.jit decorator family is deprecated.

General Enhancements:

- PR #5463: Add str(int) impl

- PR #5526: Impl. np.asarray(literal)

- PR #5619: Add support for multi-output ufuncs
- PR #5711: Division with timedelta input
- PR #5763: Support minlength argument to np.bincount
- PR #5779: Return zero array from np.dot when the arguments are empty.
- PR #5796: Add implementation for np.positive
- PR #5849: Setitem for records when index is StringLiteral, including literal unroll
- PR #5856: Add support for conversion of inplace_binop to parfor.
- PR #5893: Allocate 1D iteration space one at a time for more even distribution.
- PR #5922: Reduce objmode and unpickling overhead
- PR #5944: re-enable OpenMP in wheels
- PR #5946: Implement literal dictionaries and lists.
- PR #5956: Update numba_sysinfo.py
- PR #5978: Add structref as a mutable struct that is pass-by-ref
- PR #5980: Deprecate target kwarg for numba.jit.
- PR #6058: Add prefer_literal option to overload API

Fixes:

- PR #5674: Fix #3955. Allow *with objmode* to be cached
- PR #5724: Initialize process lock lazily to prevent multiprocessing issue
- PR #5783: Make np.divide and np.remainder code more similar
- PR #5808: Fix 5665 Block jit(nopython=True, forceobj=True) and suppress njit(forceobj=True)
- PR #5834: Fix the is operator on Ellipsis
- PR #5838: Ensure `Dispatcher.__eq__` always returns a bool
- PR #5841: cleanup: Use PythonAPI.bool_from_bool in more places
- PR #5862: Do not leak loop iteration variables into the numba.np.npyimpl namespace
- PR #5869: Update repomap
- PR #5879: Fix erroneous input mutation in linalg routines
- PR #5882: Type check function in jit decorator
- PR #5925: Use np.inf and -np.inf for max and min float values respectively.
- PR #5935: Fix default arguments with multiprocessing
- PR #5952: Fix "Internal error ... local variable 'errstr' referenced before assignment during BoundFunction(...)"
- PR #5962: Fix SVML tests with LLVM 10 and AVX512
- PR #5972: fix flake8 for numba/runtests.py
- PR #5995: Update setup.py with new llvmlite versions
- PR #5996: Set lower bound for llvmlite to 0.33
- PR #6004: Fix problem in branch pruning with LiteralStrKeyDict

- PR #6017: Fixing up numba_do_raise

- PR #6028: Fix #6023

- PR #6031: Continue 5821

- PR #6035: Fix overspecialize of literal

- PR #6046: Fixes statement reordering bug in maximize fusion step.

- PR #6056: Fix issue on invalid inlining of non-empty build_list by inline_arraycall

- PR #6057: fix aarch64/python_3.8 failure on master

- PR #6070: Fix overspecialized containers

- PR #6071: Remove f-strings in setup.py

- PR #6072: Fix for #6005

- PR #6073: Fixes invalid C prototype in helper function.

- PR #6078: Duplicate NumPy's PyArray_DescrCheck macro

- PR #6081: Fix issue with cross drive use and relpath.

- PR #6083: Fix bug in initial value unify.

- PR #6087: remove invalid sanity check from randrange tests

- PR #6089: Fix invalid reference to TypingError

- PR #6097: Add function code and closure bytes into cache key

- PR #6099: Restrict upper limit of TBB version due to ABI changes.

- PR #6101: Restrict lower limit of icc_rt version due to assumed SVML bug.

- PR #6107: Fix and test #6095

- PR #6109: Fixes an issue reported in #6094

- PR #6111: Decouple LiteralList and LiteralStrKeyDict from tuple

- PR #6116: Fix #6102. Problem with non-unique label.

CUDA Enhancements/Fixes:

- PR #5359: Remove special-casing of 0d arrays

- PR #5709: CUDA: Refactoring of cuda.jit and kernel / dispatcher abstractions

- PR #5732: CUDA Docs: document `forall` method of kernels

- PR #5745: CUDA stream callbacks and async awaitable streams

- PR #5761: Add implmentation for int types for isnan and isinf for CUDA

- PR #5819: Add support for CUDA 11 and Ampere / CC 8.0

- PR #5826: CUDA: Add function to get SASS for kernels

- PR #5846: CUDA: Allow disabling NVVM optimizations, and fix debug issues

- PR #5851: CUDA EMM enhancements - add default get_ipc_handle implementation, skip a test conditionally

- PR #5852: CUDA: Fix `cuda.test()`

- PR #5857: CUDA docs: Add notes on resetting the EMM plugin

- PR #5859: CUDA: Fix reduce docs and style improvements

- PR #6016: Fixes change of list spelling in a cuda test.

- PR #6020: CUDA: Fix #5820, adding atomic nanmin / nanmax

- PR #6030: CUDA: Don't optimize IR before sending it to NVVM

- PR #6052: Fix dtype for atomic_add_double testsuite

- PR #6080: CUDA: Prevent auto-upgrade of atomic intrinsics

- PR #6123: Fix #6121

Documentation Updates:

- PR #5782: Host docs on Read the Docs

- PR #5830: doc: Mention that caching uses pickle

- PR #5963: Fix broken link to numpy ufunc signature docs

- PR #5975: restructure communication section

- PR #5981: Document bounds-checking behavior in python deviations page

- PR #5993: Docs for structref

- PR #6008: Small fix so bullet points are rendered by sphinx

- PR #6013: emphasize cuda kernel functions are asynchronous

- PR #6036: Update deprecation doc from numba.errors to numba.core.errors

- PR #6062: Change references to numba.pydata.org to https

CI updates:

- PR #5850: Updates the "New Issue" behaviour to better redirect users.

- PR #5940: Add discourse badge

- PR #5960: Setting mypy on CI

Enhancements from user contributed PRs (with thanks!):

- Aisha Tammy added the ability to switch off TBB support at compile time in #5821 (continued in #6031 by Stuart Archibald).

- Alexander Stiebing fixed a reference before assignment bug in #5952.

- Alexey Kozlov fixed a bug in tuple getitem for literals in #6028.

- Andrew Eckart updated the repomap in #5869, added support for Read the Docs in #5782, fixed a bug in the `np.dot` implementation to correctly handle empty arrays in #5779 and added support for `minlength` to `np.bincount` in #5763.

- `@bitsisbits` updated `numba_sysinfo.py` to handle HSA agents correctly in #5956.

- Daichi Suzuo Fixed a bug in the threading backend initialisation sequence such that it is now correctly a lazy lock in #5724.

- Eric Wieser contributed a number of patches, particularly in enhancing and improving the `ufunc` capabilities:

    - #5359: Remove special-casing of 0d arrays

    - #5834: Fix the is operator on Ellipsis

    - #5619: Add support for multi-output ufuncs

    - #5841: cleanup: Use PythonAPI.bool_from_bool in more places

- #5862: Do not leak loop iteration variables into the numba.np.npyimpl namespace

- #5838: Ensure `Dispatcher.__eq__` always returns a bool

- #5830: doc: Mention that caching uses pickle

- #5783: Make np.divide and np.remainder code more similar

- Ethan Pronovost added a guard to prevent the common mistake of applying a jit decorator to the same function twice in #5881.

- Graham Markall contributed many patches to the CUDA target, as follows:

  - #6052: Fix dtype for atomic_add_double tests

  - #6030: CUDA: Don't optimize IR before sending it to NVVM

  - #5846: CUDA: Allow disabling NVVM optimizations, and fix debug issues

  - #5826: CUDA: Add function to get SASS for kernels

  - #5851: CUDA EMM enhancements - add default get_ipc_handle implementation, skip a test conditionally

  - #5709: CUDA: Refactoring of cuda.jit and kernel / dispatcher abstractions

  - #5819: Add support for CUDA 11 and Ampere / CC 8.0

  - #6020: CUDA: Fix #5820, adding atomic nanmin / nanmax

  - #5857: CUDA docs: Add notes on resetting the EMM plugin

  - #5859: CUDA: Fix reduce docs and style improvements

  - #5852: CUDA: Fix `cuda.test()`

  - #5732: CUDA Docs: document `forall` method of kernels

- Guilherme Leobas added support for `str(int)` in #5463 and `np.asarray(literal value)`` in #5526.

- Hameer Abbasi deprecated the `target` kwarg for `numba.jit` in #5980.

- Hannes Pahl added a badge to the Numba github page linking to the new discourse forum in #5940 and also fixed a bug that permitted illegal combinations of flags to be passed into `@jit` in #5808.

- Kayran Schmidt emphasized that CUDA kernel functions are asynchronous in the documentation in #6013.

- Leonardo Uieda fixed a broken link to the NumPy ufunc signature docs in #5963.

- Lucio Fernandez-Arjona added mypy to CI and started adding type annotations to the code base in #5960, also fixed a (de)serialization problem on the dispatcher in #5935, improved the undefined variable error message in #5876, added support for division with timedelta input in #5711 and implemented `setitem` for records when the index is a `StringLiteral` in #5849.

- Ludovic Tiako documented Numba's bounds-checking behavior in the python deviations page in #5981.

- Matt Roeschke changed all `http` references `https` in #6062.

- `@niteya-shah` implemented `isnan` and `isinf` for integer types on the CUDA target in #5761 and implemented `np.positive` in #5796.

- Peter Würtz added CUDA stream callbacks and async awaitable streams in #5745.

- `@rht` fixed an invalid import referred to in the deprecation documentation in #6036.

- Sergey Pokhodenko updated the SVML tests for LLVM 10 in #5962.

- Shyam Saladi fixed a Sphinx rendering bug in #6008.

Authors:

- Aisha Tammy

- Alexander Stiebing

- Alexey Kozlov

- Andrew Eckart

- `@bitsisbits`

- Daichi Suzuo

- Eric Wieser

- Ethan Pronovost

- Graham Markall

- Guilherme Leobas

- Hameer Abbasi

- Hannes Pahl

- Kayran Schmidt

- Kozlov, Alexey

- Leonardo Uieda

- Lucio Fernandez-Arjona

- Ludovic Tiako

- Matt Roeschke

- `@niteya-shah`

- Peter Würtz

- Sergey Pokhodenko

- Shyam Saladi

- `@rht`

- Siu Kwan Lam (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

- Valentin Haenel (core dev)

10.2 Version 0.50.1 (Jun 24, 2020)

This is a bugfix release for 0.50.0, it fixes a critical bug in error reporting and a number of other smaller issues:

- PR #5861: Added except for possible Windows get_terminal_size exception

- PR #5876: Improve undefined variable error message

- PR #5884: Update the deprecation notices for 0.50.1

- PR #5889: Fixes literally not forcing re-dispatch for inline='always'

- PR #5912: Fix bad attr access on certain typing templates breaking exceptions.

- PR #5918: Fix cuda test due to #5876

Authors:

- `@pepping_dore`

- Lucio Fernandez-Arjona

- Siu Kwan Lam (core dev)

- Stuart Archibald (core dev)

10.3 Version 0.50.0 (Jun 10, 2020)

This is a more usual release in comparison to the others that have been made in the last six months. It comprises the result of a number of maintenance tasks along with some new features and a lot of bug fixes.

Highlights of core feature changes include:

- The compilation chain is now based on LLVM 9.

- The error handling and reporting system has been improved to reduce the size of error messages, and also improve quality and specificity.

- The CUDA target has more stream constructors available and a new function for compiling to PTX without linking and loading the code to a device. Further, the macro-based system for describing CUDA threads and blocks has been replaced with standard typing and lowering implementations, for improved debugging and extensibility.

IMPORTANT: The backwards compatibility shim, that was present in 0.49.x to accommodate the refactoring of Numba's internals, has been removed. If a module is imported from a moved location an `ImportError` will occur.

General Enhancements:

- PR #5060: Enables np.sum for timedelta64

- PR #5225: Adjust interpreter to make conditionals predicates via bool() call.

- PR #5506: Jitclass static methods

- PR #5580: Revert shim

- PR #5591: Fix #5525 Add figure for total memory to `numba -s` output.

- PR #5616: Simplify the ufunc kernel registration

- PR #5617: Remove /examples from the Numba repo.

- PR #5673: Fix inliners to run all passes on IR and clean up correctly.

- PR #5700: Make it easier to understand type inference: add SSA dump, use for `DEBUG_TYPEINFER`

- PR #5702: Fixes for LLVM 9

- PR #5722: Improve error messages.

- PR #5758: Support NumPy 1.18

Fixes:

- PR #5390: add error handling for lookup_module

- PR #5464: Jitclass drops annotations to avoid error

- PR #5478: Fix #5471. Issue with omitted type not recognized as literal value.

- PR #5517: Fix numba.typed.List extend for singleton and empty iterable
- PR #5549: Check type getitem
- PR #5568: Add skip to entrypoint test on windows
- PR #5581: Revert #5568
- PR #5602: Fix segfault caused by pop from numba.typed.List
- PR #5645: Fix SSA redundant CFG computation
- PR #5686: Fix issue with SSA not minimal
- PR #5689: Fix bug in unified_function_type (issue 5685)
- PR #5694: Skip part of slice array analysis if any part is not analyzable.
- PR #5697: Fix usedef issue with parfor loopnest variables.
- PR #5705: A fix for cases where SSA looks like a reduction variable.
- PR #5714: Fix bug in test
- PR #5717: Initialise Numba extensions ahead of any compilation starting.
- PR #5721: Fix array iterator layout.
- PR #5738: Unbreak master on buildfarm
- PR #5757: Force LLVM to use ZMM registers for vectorization.
- PR #5764: fix flake8 errors
- PR #5768: Interval example: fix import
- PR #5781: Moving record array examples to a test module
- PR #5791: Fix up no cgroups problem
- PR #5795: Restore refct removal pass and make it strict
- PR #5807: Skip failing test on POWER8 due to PPC CTR Loop problem.
- PR #5812: Fix side issue from #5792, @overload inliner cached IR being mutated.
- PR #5815: Pin llvmlite to 0.33
- PR #5833: Fixes the source location appearing incorrectly in error messages.

CUDA Enhancements/Fixes:

- PR #5347: CUDA: Provide more stream constructors
- PR #5388: CUDA: Fix OOB write in test_round{f4,f8}
- PR #5437: Fix #5429: Exception using `.get_ipc_handle(...)` on array from `as_cuda_array(...)`
- PR #5481: CUDA: Replace macros with typing and lowering implementations
- PR #5556: CUDA: Make atomic semantics match Python / NumPy, and fix #5458
- PR #5558: CUDA: Only release primary ctx if retained
- PR #5561: CUDA: Add function for compiling to PTX (+ other small fixes)
- PR #5573: CUDA: Skip tests under cuda-memcheck that hang it
- PR #5578: Implement math.modf for CUDA target
- PR #5704: CUDA Eager compilation: Fix max_registers kwarg

- PR #5718: CUDA lib path tests: unset CUDA_PATH when CUDA_HOME unset

- PR #5800: Fix LLVM 9 IR for NVVM

- PR #5803: CUDA Update expected error messages to fix #5797

Documentation Updates:

- PR #5546: DOC: Add documentation about cost model to inlining notes.

- PR #5653: Update doc with respect to try-finally case

Enhancements from user contributed PRs (with thanks!):

- Elias Kuthe fixed in issue with imports in the Interval example in #5768

- Eric Wieser Simplified the ufunc kernel registration mechanism in #5616

- Ethan Pronovost patched a problem with `__annotations__` in `jitclass` in #5464, fixed a bug that lead to infinite loops in Numba's `Type.__getitem__` in #5549, fixed a bug in `np.arange` testing in #5714 and added support for `@staticmethod` to `jitclass` in #5506.

- Gabriele Gemmi implemented `math.modf` for the CUDA target in #5578

- Graham Markall contributed many patches, largely to the CUDA target, as follows:

    - #5347: CUDA: Provide more stream constructors

    - #5388: CUDA: Fix OOB write in test_round{f4,f8}

    - #5437: Fix #5429: Exception using `.get_ipc_handle(...)` on array from `as_cuda_array(...)`

    - #5481: CUDA: Replace macros with typing and lowering implementations

    - #5556: CUDA: Make atomic semantics match Python / NumPy, and fix #5458

    - #5558: CUDA: Only release primary ctx if retained

    - #5561: CUDA: Add function for compiling to PTX (+ other small fixes)

    - #5573: CUDA: Skip tests under cuda-memcheck that hang it

    - #5648: Unset the memory manager after EMM Plugin tests

    - #5700: Make it easier to understand type inference: add SSA dump, use for `DEBUG_TYPEINFER`

    - #5704: CUDA Eager compilation: Fix max_registers kwarg

    - #5718: CUDA lib path tests: unset CUDA_PATH when CUDA_HOME unset

    - #5800: Fix LLVM 9 IR for NVVM

    - #5803: CUDA Update expected error messages to fix #5797

- Guilherme Leobas updated the documentation surrounding try-finally in #5653

- Hameer Abbasi added documentation about the cost model to the notes on inlining in #5546

- Jacques Gaudin rewrote `numba -s` to produce and consume a dictionary of output about the current system in #5591

- James Bourbeau Updated min/argmin and max/argmax to handle non-leading nans (via #5758)

- Lucio Fernandez-Arjona moved the record array examples to a test module in #5781 and added `np.timedelta64` handling to `np.sum` in #5060

- Pearu Peterson Fixed a bug in unified_function_type in #5689

- Sergey Pokhodenko fixed an issue impacting LLVM 10 regarding vectorization widths on Intel SkyLake processors in #5757

- Shan Sikdar added error handling for `lookup_module` in #5390

- @toddrme2178 add CI testing for NumPy 1.18 (via #5758)

Authors:

- Elias Kuthe

- Eric Wieser

- Ethan Pronovost

- Gabriele Gemmi

- Graham Markall

- Guilherme Leobas

- Hameer Abbasi

- Jacques Gaudin

- James Bourbeau

- Lucio Fernandez-Arjona

- Pearu Peterson

- Sergey Pokhodenko

- Shan Sikdar

- Siu Kwan Lam (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

- `@toddrme2178`

- Valentin Haenel (core dev)

10.4 Version 0.49.1 (May 7, 2020)

This is a bugfix release for 0.49.0, it fixes some residual issues with SSA form, a critical bug in the branch pruning logic and a number of other smaller issues:

- PR #5587: Fixed #5586 Threading Implementation Typos

- PR #5592: Fixes #5583 Remove references to cffi_support from docs and examples

- PR #5614: Fix invalid type in resolve for comparison expr in parfors.

- PR #5624: Fix erroneous rewrite of predicate to bit const on prune.

- PR #5627: Fixes #5623, SSA local def scan based on invalid equality assumption.

- PR #5629: Fixes naming error in array_exprs

- PR #5630: Fix #5570. Incorrect race variable detection due to SSA naming.

- PR #5638: Make literal_unroll function work as a freevar.

- PR #5648: Unset the memory manager after EMM Plugin tests

- PR #5651: Fix some SSA issues

- PR #5652: Pin to sphinx=2.4.4 to avoid problem with C declaration

- PR #5658: Fix unifying undefined first class function types issue

- PR #5669: Update example in 5m guide WRT SSA type stability.

- PR #5676: Restore `numba.types` as public API

Authors:

- Graham Markall

- Juan Manuel Cruz Martinez

- Pearu Peterson

- Sean Law

- Stuart Archibald (core dev)

- Siu Kwan Lam (core dev)

10.5 Version 0.49.0 (Apr 16, 2020)

This release is very large in terms of code changes. Large scale removal of unsupported Python and NumPy versions has taken place along with a significant amount of refactoring to simplify the Numba code base to make it easier for contributors. Numba's intermediate representation has also undergone some important changes to solve a number of long standing issues. In addition some new features have been added and a large number of bugs have been fixed!

IMPORTANT: In this release Numba's internals have moved about a lot. A backwards compatibility "shim" is provided for this release so as to not immediately break projects using Numba's internals. If a module is imported from a moved location the shim will issue a deprecation warning and suggest how to update the import statement for the new location. The shim will be removed in 0.50.0!

Highlights of core feature changes include:

- Removal of all Python 2 related code and also updating the minimum supported Python version to 3.6, the minimum supported NumPy version to 1.15 and the minimum supported SciPy version to 1.0. (Stuart Archibald).

- Refactoring of the Numba code base. The code is now organised into submodules by functionality. This cleans up Numba's top level namespace. (Stuart Archibald).

- Introduction of an `ir.Del` free static single assignment form for Numba's intermediate representation (Siu Kwan Lam and Stuart Archibald).

- An OpenMP-like thread masking API has been added for use with code using the parallel CPU backends (Aaron Meurer and Stuart Archibald).

- For the CUDA target, all kernel launches now require a configuration, this preventing accidental launches of kernels with the old default of a single thread in a single block. The hard-coded autotuner is also now removed, such tuning is deferred to CUDA API calls that provide the same functionality (Graham Markall).

- The CUDA target also gained an External Memory Management plugin interface to allow Numba to use another CUDA-aware library for all memory allocations and deallocations (Graham Markall).

- The Numba Typed List container gained support for construction from iterables (Valentin Haenel).

- Experimental support was added for first-class function types (Pearu Peterson).

Enhancements from user contributed PRs (with thanks!):

- Aaron Meurer added support for thread masking at runtime in #4615.

- Andreas Sodeur fixed a long standing bug that was preventing `cProfile` from working with Numba JIT compiled functions in #4476.

- Arik Funke fixed error messages in `test_array_reductions` (#5278), fixed an issue with test discovery (#5239), made it so the documentation would build again on windows (#5453) and fixed a nested list problem in the docs in #5489.

- Antonio Russo fixed a SyntaxWarning in #5252.

- Eric Wieser added support for inferring the types of object arrays (#5348) and iterating over 2D arrays (#5115), also fixed some compiler warnings due to missing (void) in #5222. Also helped improved the "shim" and associated warnings in #5485, #5488, #5498 and partly #5532.

- Ethan Pronovost fixed a problem with the shim erroneously warning for jitclass use in #5454 and also prevented illegal return values in jitclass `__init__` in #5505.

- Gabriel Majeri added SciPy 2019 talks to the docs in #5106.

- Graham Markall changed the Numba HTML documentation theme to resolve a number of long standing issues in #5346. Also contributed were a large number of CUDA enhancements and fixes, namely:

    - #5519: CUDA: Silence the test suite - Fix #4809, remove autojit, delete prints
    - #5443: Fix #5196: Docs: assert in CUDA only enabled for debug
    - #5436: Fix #5408: test_set_registers_57 fails on Maxwell
    - #5423: Fix #5421: Add notes on printing in CUDA kernels
    - #5400: Fix #4954, and some other small CUDA testsuite fixes
    - #5328: NBEP 7: External Memory Management Plugin Interface
    - #5144: Fix #4875: Make #2655 test with debug expect to pass
    - #5323: Document lifetime semantics of CUDA Array Interface
    - #5061: Prevent kernel launch with no configuration, remove autotuner
    - #5099: Fix #5073: Slices of dynamic shared memory all alias
    - #5136: CUDA: Enable asynchronous operations on the default stream
    - #5085: Support other itemsizes with view
    - #5059: Docs: Explain how to use Memcheck with Numba, fixups in CUDA documentation
    - #4957: Add notes on overwriting gufunc inputs to docs

- Greg Jennings fixed an issue with `np.random.choice` not acknowledging the RNG seed correctly in #3897/#5310.

- Guilherme Leobas added support for `np.isnat` in #5293.

- Henry Schreiner made the llvmlite requirements more explicit in requirements.txt in #5150.

- Ivan Butygin helped fix an issue with parfors sequential lowering in #5114/#5250.

- Jacques Gaudin fixed a bug for Python >= 3.8 in `numba -s` in #5548.

- Jim Pivarski added some hints for debugging entry points in #5280.

- John Kirkham added `numpy.dtype` coercion for the `dtype` argument to CUDA device arrays in #5252.

- Leo Fang added a list of libraries that support `__cuda_array_interface__` in #5104.

- Lucio Fernandez-Arjona added `getitem` for the NumPy record type when the index is a `StringLiteral` type in #5182 and improved the documentation rendering via additions to the TOC and removal of numbering in #5450.

- Mads R. B. Kristensen fixed an issue with `__cuda_array_interface__` not requiring the context in #5189.

- Marcin Tolysz added support for nested modules in AOT compilation in #5174.

- Mike Williams fixed some issues with NumPy records and `getitem` in the CUDA simulator in #5343.

- Pearu Peterson added experimental support for first-class function types in #5287 (and fixes in #5459, #5473/#5429, and #5557).

- Ravi Teja Gutta added support for `np.flip` in #4376/#5313.

- Rohit Sanjay fixed an issue with type refinement for unicode input supplied to typed-list `extend()` (#5295) and fixed unicode `.strip()` to strip all whitespace characters in #5213.

- Vladimir Lukyanov fixed an awkward bug in `typed.dict` in #5361, added a fix to ensure the LLVM and assembly dumps are highlighted correctly in #5357 and implemented a Numba IR Lexer and added highlighting to Numba IR dumps in #5333.

- hdf fixed an issue with the `boundscheck` flag in the CUDA jit target in #5257.

General Enhancements:

- PR #4615: Allow masking threads out at runtime
- PR #4798: Add branch pruning based on raw predicates.
- PR #5115: Add support for iterating over 2D arrays
- PR #5117: Implement ord()/chr()
- PR #5122: Remove Python 2.
- PR #5127: Calling convention adaptor for boxer/unboxer to call jitcode
- PR #5151: implement None-typed typed-list
- PR #5174: Nested modules https://github.com/numba/numba/issues/4739
- PR #5182: Add getitem for Record type when index is StringLiteral
- PR #5185: extract code-gen utilities from closures
- PR #5197: Refactor Numba, part I
- PR #5210: Remove more unsupported Python versions from build tooling.
- PR #5212: Adds support for viewing the CFG of the ELF disassembly.
- PR #5227: Immutable typed-list
- PR #5231: Added support for `np.asarray` to be used with `numba.typed.List`
- PR #5235: Added property `dtype` to `numba.typed.List`
- PR #5272: Refactor parfor: split up ParforPass
- PR #5281: Make IR ir.Del free until legalized.
- PR #5287: First-class function type
- PR #5293: np.isnat
- PR #5294: Create typed-list from iterable

- PR #5295: refine typed-list on unicode input to extend
- PR #5296: Refactor parfor: better exception from passes
- PR #5308: Provide `numba.extending.is_jitted`
- PR #5320: refactor array_analysis
- PR #5325: Let literal_unroll accept types.Named*Tuple
- PR #5330: refactor common operation in parfor lowering into a new util
- PR #5333: Add: highlight Numba IR dump
- PR #5342: Support for tuples passed to parfors.
- PR #5348: Add support for inferring the types of object arrays
- PR #5351: SSA again
- PR #5352: Add shim to accommodate refactoring.
- PR #5356: implement allocated parameter in njit
- PR #5369: Make test ordering more consistent across feature availability
- PR #5428: Wip/deprecate jitclass location
- PR #5441: Additional changes to first class function
- PR #5455: Move to llvmlite 0.32.*
- PR #5457: implement repr for untyped lists

Fixes:

- PR #4476: Another attempt at fixing frame injection in the dispatcher tracing path
- PR #4942: Prevent some parfor aliasing. Rename copied function var to prevent recursive type locking.
- PR #5092: Fix 5087
- PR #5150: More explicit llvmlite requirement in requirements.txt
- PR #5172: fix version spec for llvmlite
- PR #5176: Normalize kws going into fold_arguments.
- PR #5183: pass 'inline' explicitly to overload
- PR #5193: Fix CI failure due to missing files when installed
- PR #5213: Fix `.strip()` to strip all whitespace characters
- PR #5216: Fix namedtuple mistreated by dispatcher as simple tuple
- PR #5222: Fix compiler warnings due to missing (void)
- PR #5232: Fixes a bad import that breaks master
- PR #5239: fix test discovery for unittest
- PR #5247: Continue PR #5126
- PR #5250: Part fix/5098
- PR #5252: Trivially fix SyntaxWarning
- PR #5276: Add prange variant to has_no_side_effect.
- PR #5278: fix error messages in test_array_reductions

- PR #5310: PR #3897 continued

- PR #5313: Continues PR #4376

- PR #5318: Remove AUTHORS file reference from MANIFEST.in

- PR #5327: Add warning if FNV hashing is found as the default for CPython.

- PR #5338: Remove refcount pruning pass

- PR #5345: Disable test failing due to removed pass.

- PR #5357: Small fix to have llvm and asm highlighted properly

- PR #5361: 5081 typed.dict

- PR #5431: Add tolerance to numba extension module entrypoints.

- PR #5432: Fix code causing compiler warnings.

- PR #5445: Remove undefined variable

- PR #5454: Don't warn for numba.experimental.jitclass

- PR #5459: Fixes issue 5448

- PR #5480: Fix for #5477, literal_unroll KeyError searching for getitems

- PR #5485: Show the offending module in "no direct replacement" error message

- PR #5488: Add missing `numba.config` shim

- PR #5495: Fix missing null initializer for variable after phi strip

- PR #5498: Make the shim deprecation warnings work on python 3.6 too

- PR #5505: Better error message if __init__ returns value

- PR #5527: Attempt to fix #5518

- PR #5529: PR #5473 continued

- PR #5532: Make `numba.<mod>` available without an import

- PR #5542: Fixes RC2 module shim bug

- PR #5548: Fix #5537 Removed reference to `platform.linux_distribution`

- PR #5555: Fix #5515 by reverting changes to ArrayAnalysis

- PR #5557: First-class function call cannot use keyword arguments

- PR #5569: Fix RewriteConstGetitems not registering calltype for new expr

- PR #5571: Pin down llvmlite requirement

CUDA Enhancements/Fixes:

- PR #5061: Prevent kernel launch with no configuration, remove autotuner

- PR #5085: Support other itemsizes with view

- PR #5099: Fix #5073: Slices of dynamic shared memory all alias

- PR #5104: Add a list of libraries that support __cuda_array_interface__

- PR #5136: CUDA: Enable asynchronous operations on the default stream

- PR #5144: Fix #4875: Make #2655 test with debug expect to pass

- PR #5189: __cuda_array_interface__ not requiring context

- PR #5253: Coerce `dtype` to `numpy.dtype`

- PR #5257: boundscheck fix

- PR #5319: Make user facing error string use abs path not rel.

- PR #5323: Document lifetime semantics of CUDA Array Interface

- PR #5328: NBEP 7: External Memory Management Plugin Interface

- PR #5343: Fix cuda spoof

- PR #5400: Fix #4954, and some other small CUDA testsuite fixes

- PR #5436: Fix #5408: test_set_registers_57 fails on Maxwell

- PR #5519: CUDA: Silence the test suite - Fix #4809, remove autojit, delete prints

Documentation Updates:

- PR #4957: Add notes on overwriting gufunc inputs to docs

- PR #5059: Docs: Explain how to use Memcheck with Numba, fixups in CUDA documentation

- PR #5106: Add SciPy 2019 talks to docs

- PR #5147: Update master for 0.48.0 updates

- PR #5155: Explain what inlining at Numba IR level will do

- PR #5161: Fix README.rst formatting

- PR #5207: Remove AUTHORS list

- PR #5249: fix target path for See also

- PR #5262: fix typo in inlining docs

- PR #5270: fix 'see also' in typeddict docs

- PR #5280: Added some hints for debugging entry points.

- PR #5297: Update docs with intro to {g,}ufuncs.

- PR #5326: Update installation docs with OpenMP requirements.

- PR #5346: Docs: use sphinx_rtd_theme

- PR #5366: Remove reference to Python 2.7 in install check output

- PR #5423: Fix #5421: Add notes on printing in CUDA kernels

- PR #5438: Update package deps for doc building.

- PR #5440: Bump deprecation notices.

- PR #5443: Fix #5196: Docs: assert in CUDA only enabled for debug

- PR #5450: Docs: remove numbers and add titles to TOC

- PR #5453: fix building docs on windows

- PR #5489: docs: fix rendering of nested bulleted list

CI updates:

- PR #5314: Update the image used in Azure CI for OSX.

- PR #5360: Remove Travis CI badge.

Authors:

- Aaron Meurer
- Andreas Sodeur
- Antonio Russo
- Arik Funke
- Eric Wieser
- Ethan Pronovost
- Gabriel Majeri
- Graham Markall
- Greg Jennings
- Guilherme Leobas
- hdf
- Henry Schreiner
- Ivan Butygin
- Jacques Gaudin
- Jim Pivarski
- John Kirkham
- Leo Fang
- Lucio Fernandez-Arjona
- Mads R. B. Kristensen
- Marcin Tolysz
- Mike Williams
- Pearu Peterson
- Ravi Teja Gutta
- Rohit Sanjay
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Todd A. Anderson (core dev)
- Valentin Haenel (core dev)
- Vladimir Lukyanov

10.6 Version 0.48.0 (Jan 27, 2020)

This release is particularly small as it was present to catch anything that missed the 0.47.0 deadline (the deadline deliberately coincided with the end of support for Python 2.7). The next release will be considerably larger.

The core changes in this release are dominated by the start of the clean up needed for the end of Python 2.7 support, improvements to the CUDA target and support for numerous additional unicode string methods.

Enhancements from user contributed PRs (with thanks!):

- Brian Wignall fixed more spelling typos in #4998.

- Denis Smirnov added support for string methods `capitalize` (#4823), `casefold` (#4824), `swapcase` (#4825), `rsplit` (#4834), `partition` (#4845) and `splitlines` (#4849).

- Elena Totmenina extended support for string methods `startswith` (#4867) and added `endswith` (#4868).

- Eric Wieser made `type_callable` return the decorated function itself in #4760

- Ethan Pronovost added support for `np.argwhere` in #4617

- Graham Markall contributed a large number of CUDA enhancements and fixes, namely:

    - #5068: Remove Python 3.4 backports from utils

    - #4975: Make `device_array_like` create contiguous arrays (Fixes #4832)

    - #5023: Don't launch ForAll kernels with 0 elements (Fixes #5017)

    - #5016: Fix various issues in CUDA library search (Fixes #4979)

    - #5014: Enable use of records and bools for shared memory, remove ddt, add additional transpose tests

    - #4964: Fix #4628: Add more appropriate typing for CUDA device arrays

    - #5007: test_consuming_strides: Keep dev array alive

    - #4997: State that CUDA Toolkit 8.0 required in docs

- James Bourbeau added the Python 3.8 classifier to setup.py in #5027.

- John Kirkham added a clarification to the `__cuda_array_interface__` documentation in #5049.

- Leo Fang Fixed an indexing problem in `dummyarray` in #5012.

- Marcel Bargull fixed a build and test issue for Python 3.8 in #5029.

- Maria Rubtsov added support for string methods `isdecimal` (#4842), `isdigit` (#4843), `isnumeric` (#4844) and `replace` (#4865).

General Enhancements:

- PR #4760: Make type_callable return the decorated function

- PR #5010: merge string prs

    This merge PR included the following:

    - PR #4823: Implement str.capitalize() based on CPython

    - PR #4824: Implement str.casefold() based on CPython

    - PR #4825: Implement str.swapcase() based on CPython

    - PR #4834: Implement str.rsplit() based on CPython

    - PR #4842: Implement str.isdecimal

    - PR #4843: Implement str.isdigit

    - PR #4844: Implement str.isnumeric

    - PR #4845: Implement str.partition() based on CPython

    - PR #4849: Implement str.splitlines() based on CPython

    - PR #4865: Implement str.replace

    - PR #4867: Functionality extension str.startswith() based on CPython

– PR #4868: Add functionality for str.endswith()

- PR #5039: Disable help messages.

- PR #4617: Add coverage for `np.argwhere`

Fixes:

- PR #4724: Only use lives (and not aliases) to create post parfor live set.

- PR #4998: Fix more spelling typos

- PR #5024: Propagate semantic constants ahead of static rewrites.

- PR #5027: Add Python 3.8 classifier to setup.py

- PR #5046: Update setup.py and buildscripts for dependency requirements

- PR #5053: Convert from arrays to names in define() and don't invalidate for multiple consistent defines.

- PR #5058: Permit mixed int types in wrap_index

- PR #5078: Catch the use of global typed-list in JITed functions

- PR #5092: Fix #5087, bug in bytecode analysis.

CUDA Enhancements/Fixes:

- PR #4964: Fix #4628: Add more appropriate typing for CUDA device arrays

- PR #4975: Make `device_array_like` create contiguous arrays (Fixes #4832)

- PR #4997: State that CUDA Toolkit 8.0 required in docs

- PR #5007: test_consuming_strides: Keep dev array alive

- PR #5012: Fix IndexError when accessing the "-1" element of dummyarray

- PR #5014: Enable use of records and bools for shared memory, remove ddt, add additional transpose tests

- PR #5016: Fix various issues in CUDA library search (Fixes #4979)

- PR #5023: Don't launch ForAll kernels with 0 elements (Fixes #5017)

- PR #5068: Remove Python 3.4 backports from utils

Documentation Updates:

- PR #5049: Clarify what dictionary means

- PR #5062: Update docs for updated version requirements

- PR #5090: Update deprecation notices for 0.48.0

CI updates:

- PR #5029: Install optional dependencies for Python 3.8 tests

- PR #5040: Drop Py2.7 and Py3.5 from public CI

- PR #5048: Fix CI py38

Authors:

- Brian Wignall

- Denis Smirnov

- Elena Totmenina

- Eric Wieser

- Ethan Pronovost

- Graham Markall

- James Bourbeau

- John Kirkham

- Leo Fang

- Marcel Bargull

- Maria Rubtsov

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

- Valentin Haenel (core dev)

10.7 Version 0.47.0 (Jan 2, 2020)

This release expands the capability of Numba in a number of important areas and is also significant as it is the last major point release with support for Python 2 and Python 3.5 included. The next release (0.48.0) will be for Python 3.6+ only! (This follows NumPy's deprecation schedule as specified in NEP 29.)

Highlights of core feature changes include:

- Full support for Python 3.8 (Siu Kwan Lam)

- Opt-in bounds checking (Aaron Meurer)

- Support for `map`, `filter` and `reduce` (Stuart Archibald)

Intel also kindly sponsored research and development that lead to some exciting new features:

- Initial support for basic `try`/`except` use (Siu Kwan Lam)

- The ability to pass functions created from closures/lambdas as arguments (Stuart Archibald)

- `sorted` and `list.sort()` now accept the `key` argument (Stuart Archibald and Siu Kwan Lam)

- A new compiler pass triggered through the use of the function `numba.literal_unroll` which permits iteration over heterogeneous tuples and constant lists of constants. (Stuart Archibald)

Enhancements from user contributed PRs (with thanks!):

- Ankit Mahato added a reference to a new talk on Numba at PyCon India 2019 in #4862

- Brian Wignall kindly fixed some spelling mistakes and typos in #4909

- Denis Smirnov wrote numerous methods to considerable enhance string support including:

  - `str.rindex()` in #4861

  - `str.isprintable()` in #4836

  - `str.index()` in #4860

  - `start`/`end` parameters for `str.find()` in #4866

  - `str.isspace()` in #4835

- – `str.isidentifier()` #4837

  – `str.rpartition()` in #4841

  – `str.lower()` and `str.islower()` in #4651

- Elena Totmenina implemented both `str.isalnum()`, `str.isalpha()` and `str.isascii` in #4839, #4840 and #4847 respectively.

- Eric Larson fixed a bug in literal comparison in #4710

- Ethan Pronovost updated the `np.arange` implementation in #4770 to allow the use of the `dtype` key word argument and also added `bool` implementations for several types in #4715.

- Graham Markall fixed some issues with the CUDA target, namely:

  – #4931: Added physical limits for CC 7.0 / 7.5 to CUDA autotune

  – #4934: Fixed bugs in TestCudaWarpOperations

  – #4938: Improved errors / warnings for the CUDA vectorize decorator

- Guilherme Leobas fixed a typo in the `urem` implementation in #4667

- Isaac Virshup contributed a number of patches that fixed bugs, added support for more NumPy functions and enhanced Python feature support. These contributions included:

  – #4729: Allow array construction with mixed type shape tuples

  – #4904: Implementing `np.lcm`

  – #4780: Implement np.gcd and math.gcd

  – #4779: Make slice constructor more similar to python.

  – #4707: Added support for slice.indices

  – #4578: Clarify numba ufunc supported features

- James Bourbeau fixed some issues with tooling, #4794 add `setuptools` as a dependency and #4501 add pre-commit hooks for `flake8` compliance.

- Leo Fang made `numba.dummyarray.Array` iterable in #4629

- Marc Garcia fixed the `numba.jit` parameter name signature_or_function in #4703

- Marcelo Duarte Trevisani patched the llvmlite requirement to `>=0.30.0` in #4725

- Matt Cooper fixed a long standing CI problem in #4737 by remove maxParallel

- Matti Picus fixed an issue with `collections.abc` in #4734 from Azure Pipelines.

- Rob Ennis patched a bug in `np.interp float32` handling in #4911

- VDimir fixed a bug in array transposition layouts in #4777 and re-enabled and fixed some idle tests in #4776.

- Vyacheslav Smirnov Enable support for *str.istitle()'* in #4645

General Enhancements:

- PR #4432: Bounds checking

- PR #4501: Add pre-commit hooks

- PR #4536: Handle kw args in inliner when callee is a function

- PR #4599: Permits closures to become functions, enables map(), filter()

- PR #4611: Implement method title() for unicode based on Cpython

- PR #4645: Enable support for istitle() method for unicode string
- PR #4651: Implement str.lower() and str.islower()
- PR #4652: Implement str.rfind()
- PR #4695: Refactor *overload** and support *jit_options* and *inline*
- PR #4707: Added support for slice.indices
- PR #4715: Add *bool* overload for several types
- PR #4729: Allow array construction with mixed type shape tuples
- PR #4755: Python3.8 support
- PR #4756: Add parfor support for ndarray.fill.
- PR #4768: Update typeconv error message to ask for sys.executable.
- PR #4770: Update *np.arange* implementation with *@overload*
- PR #4779: Make slice constructor more similar to python.
- PR #4780: Implement np.gcd and math.gcd
- PR #4794: Add setuptools as a dependency
- PR #4802: put git hash into build string
- PR #4803: Better compiler error messages for improperly used reduction variables.
- PR #4817: Typed list implement and expose allocation
- PR #4818: Typed list faster copy
- PR #4835: Implement str.isspace() based on CPython
- PR #4836: Implement str.isprintable() based on CPython
- PR #4837: Implement str.isidentifier() based on CPython
- PR #4839: Implement str.isalnum() based on CPython
- PR #4840: Implement str.isalpha() based on CPython
- PR #4841: Implement str.rpartition() based on CPython
- PR #4847: Implement str.isascii() based on CPython
- PR #4851: Add graphviz output for FunctionIR
- PR #4854: Python3.8 looplifting
- PR #4858: Implement str.expandtabs() based on CPython
- PR #4860: Implement str.index() based on CPython
- PR #4861: Implement str.rindex() based on CPython
- PR #4866: Support params start/end for str.find()
- PR #4874: Bump to llvmlite 0.31
- PR #4896: Specialise arange dtype on arch + python version.
- PR #4902: basic support for try except
- PR #4904: Implement np.lcm
- PR #4910: loop canonicalisation and type aware tuple unroller/loop body versioning passes

- PR #4961: Update hash(tuple) for Python 3.8.

- PR #4977: Implement sort/sorted with key.

- PR #4987: Add *is_internal* property to all Type classes.

Fixes:

- PR #4090: Update to LLVM8 memset/memcpy intrinsic

- PR #4582: Convert sub to add and div to mul when doing the reduction across the per-thread reduction array.

- PR #4648: Handle 0 correctly as slice parameter.

- PR #4660: Remove multiply defined variables from all blocks' equivalence sets.

- PR #4672: Fix pickling of dufunc

- PR #4710: BUG: Comparison for literal

- PR #4718: Change get_call_table to support intermediate Vars.

- PR #4725: Requires llvmlite >=0.30.0

- PR #4734: prefer to import from collections.abc

- PR #4736: fix flake8 errors

- PR #4776: Fix and enable idle tests from test_array_manipulation

- PR #4777: Fix transpose output array layout

- PR #4782: Fix issue with SVML (and knock-on function resolution effects).

- PR #4785: Treat 0d arrays like scalars.

- PR #4787: fix missing incref on flags

- PR #4789: fix typos in numba/targets/base.py

- PR #4791: fix typos

- PR #4811: fix spelling in now-failing tests

- PR #4852: windowing test should check equality only up to double precision errors

- PR #4881: fix refining list by using extend on an iterator

- PR #4882: Fix return type in arange and zero step size handling.

- PR #4885: suppress spurious RuntimeWarning about ufunc sizes

- PR #4891: skip the xfail test for now. Py3.8 CFG refactor seems to have changed the test case

- PR #4892: regex needs to accept singular form of "argument"

- PR #4901: fix typed list equals

- PR #4909: Fix some spelling typos

- PR #4911: np.interp bugfix for float32 handling

- PR #4920: fix creating list with JIT disabled

- PR #4921: fix creating dict with JIT disabled

- PR #4935: Better handling of prange with multiple reductions on the same variable.

- PR #4946: Improve the error message for *raise <string>*.

- PR #4955: Move overload of literal_unroll to avoid circular dependency that breaks Python 2.7

- PR #4962: Fix test error on windows

- PR #4973: Fixes a bug in the relabelling logic in literal_unroll.

- PR #4978: Fix overload_method problem with stararg

- PR #4981: Add ind_to_const to enable fewer equivalence classes.

- PR #4991: Continuation of #4588 (Let dead code removal handle removing more of the unneeded code after prange conversion to parfor)

- PR #4994: Remove xfail for test which has since had underlying issue fixed.

- PR #5018: Fix #5011.

- PR #5019: skip pycc test on Python 3.8 + macOS because of distutils issue

CUDA Enhancements/Fixes:

- PR #4629: Make numba.dummyarray.Array iterable

- PR #4675: Bump cuda array interface to version 2

- PR #4741: Update choosing the "CUDA_PATH" for windows

- PR #4838: Permit ravel('A') for contig device arrays in CUDA target

- PR #4931: Add physical limits for CC 7.0 / 7.5 to autotune

- PR #4934: Fix fails in TestCudaWarpOperations

- PR #4938: Improve errors / warnings for cuda vectorize decorator

Documentation Updates:

- PR #4418: Directed graph task roadmap

- PR #4578: Clarify numba ufunc supported features

- PR #4655: fix sphinx build warning

- PR #4667: Fix typo on urem implementation

- PR #4669: Add link to ParallelAccelerator paper.

- PR #4703: Fix numba.jit parameter name signature_or_function

- PR #4862: Addition of PyCon India 2019 talk on Numba

- PR #4947: Document jitclass with numba.typed use.

- PR #4958: Add docs for *try..except*

- PR #4993: Update deprecations for 0.47

CI Updates:

- PR #4737: remove maxParallel from Azure Pipelines

- PR #4767: pin to 2.7.16 for py27 on osx

- PR #4781: WIP/runtest cf pytest

Authors:

- Aaron Meurer

- Ankit Mahato

- Brian Wignall

- Denis Smirnov

- Ehsan Totoni (core dev)

- Elena Totmenina

- Eric Larson

- Ethan Pronovost

- Giovanni Cavallin

- Graham Markall

- Guilherme Leobas

- Isaac Virshup

- James Bourbeau

- Leo Fang

- Marc Garcia

- Marcelo Duarte Trevisani

- Matt Cooper

- Matti Picus

- Rob Ennis

- Rujal Desai

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

- VDimir

- Valentin Haenel (core dev)

- Vyacheslav Smirnov

10.8 Version 0.46.0

This release significantly reworked one of the main parts of Numba, the compiler pipeline, to make it more extensible and easier to use. The purpose of this was to continue enhancing Numba's ability for use as a compiler toolkit. In a similar vein, Numba now has an extension registration mechanism to allow other Numba-using projects to automatically have their Numba JIT compilable functions discovered. There were also a number of other related compiler toolkit enhancement added along with some more NumPy features and a lot of bug fixes.

This release has updated the CUDA Array Interface specification to version 2, which clarifies the *strides* attribute for C-contiguous arrays and specifies the treatment for zero-size arrays. The implementation in Numba has been changed and may affect downstream packages relying on the old behavior (see issue #4661).

Enhancements from user contributed PRs (with thanks!):

- Aaron Meurer fixed some Python issues in the code base in #4345 and #4341.

- Ashwin Srinath fixed a CUDA performance bug via #4576.

- Ethan Pronovost added support for triangular indices functions in #4601 (the NumPy functions `tril_indices`, `tril_indices_from`, `triu_indices`, and `triu_indices_from`).

- Gerald Dalley fixed a tear down race occurring in Python 2.

- Gregory R. Lee fixed the use of deprecated `inspect.getargspec`.

- Guilherme Leobas contributed five PRs, adding support for `np.append` and `np.count_nonzero` in #4518 and #4386. The typed List was fixed to accept unsigned integers in #4510. #4463 made a fix to NamedTuple internals and #4397 updated the docs for `np.sum`.

- James Bourbeau added a new feature to permit the automatic application of the *jit* decorator to a whole module in #4331. Also some small fixes to the docs and the code base were made in #4447 and #4433, and a fix to inplace array operation in #4228.

- Jim Crist fixed a bug in the rendering of patched errors in #4464.

- Leo Fang updated the CUDA Array Interface contract in #4609.

- Pearu Peterson added support for Unicode based NumPy arrays in #4425.

- Peter Andreas Entschev fixed a CUDA concurrency bug in #4581.

- Lucio Fernandez-Arjona extended Numba's `np.sum` support to now accept the `dtype` kwarg in #4472.

- Pedro A. Morales Maries added support for `np.cross` in #4128 and also added the necessary extension `numba.numpy_extensions.cross2d` in #4595.

- David Hoese, Eric Firing, Joshua Adelman, and Juan Nunez-Iglesias all made documentation fixes in #4565, #4482, #4455, #4375 respectively.

- Vyacheslav Smirnov and Rujal Desai enabled support for `count()` on unicode strings in #4606.

General Enhancements:

- PR #4113: Add rewrite for semantic constants.

- PR #4128: Add np.cross support

- PR #4162: Make IR comparable and legalize it.

- PR #4208: R&D inlining, jitted and overloaded.

- PR #4331: Automatic JIT of called functions

- PR #4353: Inspection tool to check what numba supports

- PR #4386: Implement np.count_nonzero

- PR #4425: Unicode array support

- PR #4427: Entrypoints for numba extensions

- PR #4467: Literal dispatch

- PR #4472: Allow dtype input argument in np.sum

- PR #4513: New compiler.

- PR #4518: add support for np.append

- PR #4554: Refactor NRT C-API

- PR #4556: 0.46 scheduled deprecations

- PR #4567: Add env var to disable performance warnings.

- PR #4568: add np.array_equal support

- PR #4595: Implement numba.cross2d
- PR #4601: Add triangular indices functions
- PR #4606: Enable support for count() method for unicode string

Fixes:

- PR #4228: Fix inplace operator error for arrays
- PR #4282: Detect and raise unsupported on generator expressions
- PR #4305: Don't allow the allocation of mutable objects written into a container to be hoisted.
- PR #4311: Avoid deprecated use of inspect.getargspec
- PR #4328: Replace GC macro with function call
- PR #4330: Loosen up typed container casting checks
- PR #4341: Fix some coding lines at the top of some files (utf8 -> utf-8)
- PR #4345: Replace "import *" with explicit imports in numba/types
- PR #4346: Fix incorrect alg in isupper for ascii strings.
- PR #4349: test using jitclass in typed-list
- PR #4361: Add allocation hoisting info to LICM section at diagnostic L4
- PR #4366: Offset search box to avoid wrapping on some pages with Safari. Fixes #4365.
- PR #4372: Replace all "except BaseException" with "except Exception".
- PR #4407: Restore the "free" conda channel for NumPy 1.10 support.
- PR #4408: Add lowering for constant bytes.
- PR #4409: Add exception chaining for better error context
- PR #4411: Name of type should not contain user facing description for debug.
- PR #4412: Fix #4387. Limit the number of return types for recursive functions
- PR #4426: Fixed two module teardown races in py2.
- PR #4431: Fix and test numpy.random.random_sample(n) for np117
- PR #4463: NamedTuple - Raises an error on non-iterable elements
- PR #4464: Add a newline in patched errors
- PR #4474: Fix liveness for remove dead of parfors (and other IR extensions)
- PR #4510: Make List.__getitem__ accept unsigned parameters
- PR #4512: Raise specific error at typing time for iteration on >1D array.
- PR #4532: Fix static_getitem with Literal type as index
- PR #4547: Update to inliner cost model information.
- PR #4557: Use specific random number seed when generating arbitrary test data
- PR #4559: Adjust test timeouts
- PR #4564: Skip unicode array tests on ppc64le that trigger an LLVM bug
- PR #4621: Fix packaging issue due to missing numba/cext
- PR #4623: Fix issue 4520 due to storage model mismatch

- PR #4644: Updates for llvmlite 0.30.0

CUDA Enhancements/Fixes:

- PR #4410: Fix #4111. cudasim mishandling recarray
- PR #4576: Replace use of *np.prod* with *functools.reduce* for computing size from shape
- PR #4581: Prevent taking the GIL in ForAll
- PR #4592: Fix #4589. Just pass NULL for b2d_func for constant dynamic sharedmem
- PR #4609: Update CUDA Array Interface & Enforce Numba compliance
- PR #4619: Implement math.{degrees, radians} for the CUDA target.
- PR #4675: Bump cuda array interface to version 2

Documentation Updates:

- PR #4317: Add docs for ARMv8/AArch64
- PR #4318: Add supported platforms to the docs. Closes #4316
- PR #4375: Add docstrings to inspect methods
- PR #4388: Update Python 2.7 EOL statement
- PR #4397: Add note about np.sum
- PR #4447: Minor parallel performance tips edits
- PR #4455: Clarify docs for typed dict with regard to arrays
- PR #4482: Fix example in guvectorize docstring.
- PR #4541: fix two typos in architecture.rst
- PR #4548: Document numba.extending.intrinsic and inlining.
- PR #4565: Fix typo in jit-compilation docs
- PR #4607: add dependency list to docs
- PR #4614: Add documentation for implementing new compiler passes.

CI Updates:

- PR #4415: Make 32bit incremental builds on linux not use free channel
- PR #4433: Removes stale azure comment
- PR #4493: Fix Overload Inliner wrt CUDA Intrinsics
- PR #4593: Enable Azure CI batching

Contributors:

- Aaron Meurer
- Ashwin Srinath
- David Hoese
- Ehsan Totoni (core dev)
- Eric Firing
- Ethan Pronovost
- Gerald Dalley

- Gregory R. Lee
- Guilherme Leobas
- James Bourbeau
- Jim Crist
- Joshua Adelman
- Juan Nunez-Iglesias
- Leo Fang
- Lucio Fernandez-Arjona
- Pearu Peterson
- Pedro A. Morales Marie
- Peter Andreas Entschev
- Rujal Desai
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Todd A. Anderson (core dev)
- Valentin Haenel (core dev)
- Vyacheslav Smirnov

10.9 Version 0.45.1

This patch release addresses some regressions reported in the 0.45.0 release and adds support for NumPy 1.17:

- PR #4325: accept scalar/0d-arrays
- PR #4338: Fix #4299. Parfors reduction vars not deleted.
- PR #4350: Use process level locks for fork() only.
- PR #4354: Try to fix #4352.
- PR #4357: Fix np1.17 isnan, isinf, isfinite ufuncs
- PR #4363: Fix np.interp for np1.17 nan handling
- PR #4371: Fix nump1.17 random function non-aliasing

Contributors:

- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)
- Valentin Haenel (core dev)

10.10 Version 0.45.0

In this release, Numba gained an experimental *numba.typed.List* container as a future replacement of the *reflected list*. In addition, functions decorated with `parallel=True` can now be cached to reduce compilation overhead associated with the auto-parallelization.

Enhancements from user contributed PRs (with thanks!):

- James Bourbeau added the Numba version to reportable error messages in #4227, added the `signature` parameter to `inspect_types` in #4200, improved the docstring of `normalize_signature` in #4205, and fixed #3658 by adding reference counting to `register_dispatcher` in #4254

- Guilherme Leobas implemented the dominator tree and dominance frontier algorithms in #4216 and #4149, respectively.

- Nick White fixed the issue with `round` in the CUDA target in #4137.

- Joshua Adelman added support for determining if a value is in a *range* (i.e. `x in range(...)`) in #4129, and added windowing functions (`np.bartlett`, `np.hamming`, `np.blackman`, `np.hanning`, `np.kaiser`) from NumPy in #4076.

- Lucio Fernandez-Arjona added support for `np.select` in #4077

- Rob Ennis added support for `np.flatnonzero` in #4157

- Keith Kraus extended the `__cuda_array_interface__` with an optional mask attribute in #4199.

- Gregory R. Lee replaced deprecated use of `inspect.getargspec` in #4311.

General Enhancements:

- PR #4328: Replace GC macro with function call
- PR #4311: Avoid deprecated use of inspect.getargspec
- PR #4296: Slacken window function testing tol on ppc64le
- PR #4254: Add reference counting to register_dispatcher
- PR #4239: Support len() of multi-dim arrays in array analysis
- PR #4234: Raise informative error for np.kron array order
- PR #4232: Add unicodetype db, low level str functions and examples.
- PR #4229: Make hashing cacheable
- PR #4227: Include numba version in reportable error message
- PR #4216: Add dominator tree
- PR #4200: Add signature parameter to inspect_types
- PR #4196: Catch missing imports of internal functions.
- PR #4180: Update use of unlowerable global message.
- PR #4166: Add tests for PR #4149
- PR #4157: Support for np.flatnonzero
- PR #4149: Implement dominance frontier for SSA for the Numba IR
- PR #4148: Call branch pruning in inline_closure_call()
- PR #4132: Reduce usage of inttoptr
- PR #4129: Support contains for range

- PR #4112: better error messages for np.transpose and tuples
- PR #4110: Add range attrs, start, stop, step
- PR #4077: Add np select
- PR #4076: Add numpy windowing functions support (np.bartlett, np.hamming, np.blackman, np.hanning, np.kaiser)
- PR #4095: Support ir.Global/FreeVar in find_const()
- PR #3691: Make TypingError abort compiling earlier
- PR #3646: Log internal errors encountered in typeinfer

Fixes:

- PR #4303: Work around scipy bug 10206
- PR #4302: Fix flake8 issue on master
- PR #4301: Fix integer literal bug in np.select impl
- PR #4291: Fix pickling of jitclass type
- PR #4262: Resolves #4251 - Fix bug in reshape analysis.
- PR #4233: Fixes issue revealed by #4215
- PR #4224: Fix #4223. Looplifting error due to StaticSetItem in objectmode
- PR #4222: Fix bad python path.
- PR #4178: Fix unary operator overload, check with unicode impl
- PR #4173: Fix return type in np.bincount with weights
- PR #4153: Fix slice shape assignment in array analysis
- PR #4152: fix status check in dict lookup
- PR #4145: Use callable instead of checking __module__
- PR #4118: Fix inline assembly support on CPU.
- PR #4088: Resolves #4075 - parfors array_analysis bug.
- PR #4085: Resolves #3314 - parfors array_analysis bug with reshape.

CUDA Enhancements/Fixes:

- PR #4199: Extend *__cuda_array_interface__* with optional mask attribute, bump version to 1
- PR #4137: CUDA - Fix round Builtin
- PR #4114: Support 3rd party activated CUDA context

Documentation Updates:

- PR #4317: Add docs for ARMv8/AArch64
- PR #4318: Add supported platforms to the docs. Closes #4316
- PR #4295: Alter deprecation schedules
- PR #4253: fix typo in pysupported docs
- PR #4252: fix typo on repomap
- PR #4241: remove unused import

- PR #4240: fix typo in jitclass docs
- PR #4205: Update return value order in normalize_signature docstring
- PR #4237: Update doc links to point to latest not dev docs.
- PR #4197: hyperlink repomap
- PR #4170: Clarify docs on accumulating into arrays in prange
- PR #4147: fix docstring for DictType iterables
- PR #3951: A guide to overloading

CI Updates:

- PR #4300: AArch64 has no faulthandler package
- PR #4273: pin to MKL BLAS for testing to get consistent results
- PR #4209: Revert previous network tol patch and try with conda config
- PR #4138: Remove tbb before Azure test only on Python 3, since it was already removed for Python 2

Contributors:

- Ehsan Totoni (core dev)
- Gregory R. Lee
- Guilherme Leobas
- James Bourbeau
- Joshua L. Adelman
- Keith Kraus
- Lucio Fernandez-Arjona
- Nick White
- Rob Ennis
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Todd A. Anderson (core dev)
- Valentin Haenel (core dev)

10.11 Version 0.44.1

This patch release addresses some regressions reported in the 0.44.0 release:

- PR #4165: Fix #4164 issue with NUMBAPRO_NVVM.
- PR #4172: Abandon branch pruning if an arg name is redefined. (Fixes #4163)
- PR #4183: Fix #4156. Problem with defining in-loop variables.

10.12 Version 0.44.0

IMPORTANT: In this release a few significant deprecations (and some less significant ones) are being made, users are encouraged to read the related documentation.

General enhancements in this release include:

- Numba is backed by LLVM 8 on all platforms apart from ppc64le, which, due to bugs, remains on the LLVM 7.x series.

- Numba's dictionary support now includes type inference for keys and values.

- The .view() method now works for NumPy scalar types.

- Newly supported NumPy functions added: np.delete, np.nanquantile, np.quantile, np.repeat, np.shape.

In addition considerable effort has been made to fix some long standing bugs and a large number of other bugs, the "Fixes" section is very large this time!

Enhancements from user contributed PRs (with thanks!):

- Max Bolingbroke added support for the selective use of `fastmath` flags in #3847.

- Rob Ennis made min() and max() work on iterables in #3820 and added np.quantile and np.nanquantile in #3899.

- Sergey Shalnov added numerous unicode string related features, zfill in #3978, ljust in #4001, rjust and center in #4044 and strip, lstrip and rstrip in #4048.

- Guilherme Leobas added support for np.delete in #3890

- Christoph Deil exposed the Numba CLI via `python -m numba` in #4066 and made numerous documentation fixes.

- Leo Schwarz wrote the bulk of the code for jitclass default constructor arguments in #3852.

- Nick White enhanced the CUDA backend to use min/max PTX instructions where possible in #4054.

- Lucio Fernandez-Arjona implemented the unicode string __mul__ function in #3952.

- Dimitri Vorona wrote the bulk of the code to implement getitem and setitem for jitclass in #3861.

General Enhancements:

- PR #3820: Min max on iterables

- PR #3842: Unicode type iteration

- PR #3847: Allow fine-grained control of fastmath flags to partially address #2923

- PR #3852: Continuation of PR #2894

- PR #3861: Continuation of PR #3730

- PR #3890: Add support for np.delete

- PR #3899: Support for np.quantile and np.nanquantile

- PR #3900: Fix 3457 :: Implements np.repeat

- PR #3928: Add .view() method for NumPy scalars

- PR #3939: Update icc_rt clone recipe.

- PR #3952: __mul__ for strings, initial implementation and tests

- PR #3956: Type-inferred dictionary

- PR #3959: Create a view for string slicing to avoid extra allocations

- PR #3978: zfill operation implementation
- PR #4001: ljust operation implementation
- PR #4010: Support *dict()* and *{}*
- PR #4022: Support for llvm 8
- PR #4034: Make type.Optional str more representative
- PR #4041: Deprecation warnings
- PR #4044: rjust and center operations implementation
- PR #4048: strip, lstrip and rstrip operations implementation
- PR #4066: Expose numba CLI via python -m numba
- PR #4081: Impl *np.shape* and support function for *asarray*.
- PR #4091: Deprecate the use of iternext_impl without RefType

CUDA Enhancements/Fixes:

- PR #3933: Adds *.nbytes* property to CUDA device array objects.
- PR #4011: Add .inspect_ptx() to cuda device function
- PR #4054: CUDA: Use min/max PTX Instructions
- PR #4096: Update env-vars for CUDA libraries lookup

Documentation Updates:

- PR #3867: Code repository map
- PR #3918: adding Joris' Fosdem 2019 presentation
- PR #3926: order talks on applications of Numba by date
- PR #3943: fix two small typos in vectorize docs
- PR #3944: Fixup jitclass docs
- PR #3990: mention preprint repo in FAQ. Fixes #3981
- PR #4012: Correct runtests command in contributing.rst
- PR #4043: fix typo
- PR #4047: Ambiguous Documentation fix for guvectorize.
- PR #4060: Remove remaining mentions of autojit in docs
- PR #4063: Fix annotate example in docstring
- PR #4065: Add FAQ entry explaining Numba project name
- PR #4079: Add Documentation for atomicity of typed.Dict
- PR #4105: Remove info about CUDA ENVVAR potential replacement

Fixes:

- PR #3719: Resolves issue #3528. Adds support for slices when not using parallel=True.
- PR #3727: Remove dels for known dead vars.
- PR #3845: Fix mutable flag transmission in .astype
- PR #3853: Fix some minor issues in the C source.

- PR #3862: Correct boolean reinterpretation of data
- PR #3863: Comments out the appveyor badge
- PR #3869: fixes flake8 after merge
- PR #3871: Add assert to ir.py to help enforce correct structuring
- PR #3881: fix preparfor dtype transform for datetime64
- PR #3884: Prevent mutation of objmode fallback IR.
- PR #3885: Updates for llvmlite 0.29
- PR #3886: Use *safe_load* from pyyaml.
- PR #3887: Add tolerance to network errors by permitting conda to retry
- PR #3893: Fix casting in namedtuple ctor.
- PR #3894: Fix array inliner for multiple array definition.
- PR #3905: Cherrypick #3903 to main
- PR #3920: Raise better error if unsupported jump opcode found.
- PR #3927: Apply flake8 to the numpy related files
- PR #3935: Silence DeprecationWarning
- PR #3938: Better error message for unknown opcode
- PR #3941: Fix typing of ufuncs in parfor conversion
- PR #3946: Return variable renaming dict from inline_closurecall
- PR #3962: Fix bug in alignment computation of *Record.make_c_struct*
- PR #3967: Fix error with pickling unicode
- PR #3964: Unicode split algo versioning
- PR #3975: Add handler for unknown locale to numba -s
- PR #3991: Permit Optionals in ufunc machinery
- PR #3995: Remove assert in type inference causing poor error message.
- PR #3996: add is_ascii flag to UnicodeType
- PR #4009: Prevent zero division error in np.linalg.cond
- PR #4014: Resolves #4007.
- PR #4021: Add a more specific error message for invalid write to a global.
- PR #4023: Fix handling of titles in record dtype
- PR #4024: Do a check if a call is const before saying that an object is multiply defined.
- PR #4027: Fix issue #4020. Turn off no_cpython_wrapper flag when compiling for. . .
- PR #4033: [WIP] Fixing wrong dtype of array inside reflected list #4028
- PR #4061: Change IPython cache dir name to numba_cache
- PR #4067: Delete examples/notebooks/LinearRegr.py
- PR #4070: Catch writes to global typed.Dict and raise.
- PR #4078: Check tuple length

- PR #4084: Fix missing incref on optional return None
- PR #4089: Make the warnings fixer flush work for warning comparing on type.
- PR #4094: Fix function definition finding logic for commented def
- PR #4100: Fix alignment check on 32-bit.
- PR #4104: Use PEP 508 compliant env markers for install deps

Contributors:

- Benjamin Zaitlen
- Christoph Deil
- David Hirschfeld
- Dimitri Vorona
- Ehsan Totoni (core dev)
- Guilherme Leobas
- Leo Schwarz
- Lucio Fernandez-Arjona
- Max Bolingbroke
- NanduTej
- Nick White
- Ravi Teja Gutta
- Rob Ennis
- Sergey Shalnov
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Todd A. Anderson (core dev)
- Valentin Haenel (core dev)

10.13 Version 0.43.1

This is a bugfix release that provides minor changes to fix: a bug in branch pruning, bugs in *np.interp* functionality, and also fully accommodate the NumPy 1.16 release series.

- PR #3826: NumPy 1.16 support
- PR #3850: Refactor np.interp
- PR #3883: Rewrite pruned conditionals as their evaluated constants.

Contributors:

- Rob Ennis
- Siu Kwan Lam (core dev)
- Stuart Archibald (core dev)

10.14 Version 0.43.0

In this release, the major new features are:

- Initial support for statically typed dictionaries
- Improvements to *hash()* to match Python 3 behavior
- Support for the heapq module
- Ability to pass C structs to Numba
- More NumPy functions: asarray, trapz, roll, ptp, extract

NOTE:

The vast majority of NumPy 1.16 behaviour is supported, however `datetime` and `timedelta` use involving `NaT` matches the behaviour present in earlier release. The ufunc suite has not been extending to accommodate the two new time computation related additions present in NumPy 1.16. In addition the functions `ediff1d` and `interp` have known minor issues in replicating outputs exactly when `NaN`'s occur in certain input patterns.

General Enhancements:

- PR #3563: Support for np.roll
- PR #3572: Support for np.ptp
- PR #3592: Add dead branch prune before type inference.
- PR #3598: Implement np.asarray()
- PR #3604: Support for np.interp
- PR #3607: Some simplication to lowering
- PR #3612: Exact match flag in dispatcher
- PR #3627: Support for np.trapz
- PR #3630: np.where with broadcasting
- PR #3633: Support for np.extract
- PR #3657: np.max, np.min, np.nanmax, np.nanmin - support for complex dtypes
- PR #3661: Access C Struct as Numpy Structured Array
- PR #3678: Support for str.split and str.join
- PR #3684: Support C array in C struct
- PR #3696: Add intrinsic to help debug refcount
- PR #3703: Implementations of type hashing.
- PR #3715: Port CPython3.7 dictionary for numba internal use
- PR #3716: Support inplace concat of strings
- PR #3718: Add location to ConstantInferenceError exceptions.
- PR #3720: improve error msg about invalid signature
- PR #3731: Support for heapq
- PR #3754: Updates for llvmlite 0.28
- PR #3760: Overloadable operator.setitem

- PR #3775: Support overloading operator.delitem
- PR #3777: Implement compiler support for dictionary
- PR #3791: Implement interpreter-side interface for numba dict
- PR #3799: Support refcount'ed types in numba dict

CUDA Enhancements/Fixes:

- PR #3713: Fix the NvvmSupportError message when CC too low
- PR #3722: Fix #3705: slicing error with negative strides
- PR #3755: Make cuda.to_device accept readonly host array
- PR #3773: Adapt library search to accommodate multiple locations

Documentation Updates:

- PR #3651: fix link to berryconda in docs
- PR #3668: Add Azure Pipelines build badge
- PR #3749: DOC: Clarify when prange is different from range
- PR #3771: fix a few typos
- PR #3785: Clarify use of range as function only.
- PR #3829: Add docs for typed-dict

Fixes:

- PR #3614: Resolve #3586
- PR #3618: Skip gdb tests on ARM.
- PR #3643: Remove support_literals usage
- PR #3645: Enforce and fix that AbstractTemplate.generic must be returning a Signature
- PR #3648: Fail on @overload signature mismatch.
- PR #3660: Added Ignore message to test numba.tests.test_lists.TestLists.test_mul_error
- PR #3662: Replace six with numba.six
- PR #3663: Removes coverage computation from travisci builds
- PR #3672: Avoid leaking memory when iterating over uniform tuple
- PR #3676: Fixes constant string lowering inside tuples
- PR #3677: Ensure all referenced compiled functions are linked properly
- PR #3692: Fix test failure due to overly strict test on floating point values.
- PR #3693: Intercept failed import to help users.
- PR #3694: Fix memory leak in enumerate iterator
- PR #3695: Convert return of None from intrinsic implementation to dummy value
- PR #3697: Fix for issue #3687
- PR #3701: Fix array.T analysis (fixes #3700)
- PR #3704: Fixes for overload_method
- PR #3706: Don't push call vars recursively into nested parfors. Resolves #3686.

- PR #3710: Set as non-hoistable if a mutable variable is passed to a function in a loop. Resolves #3699.

- PR #3712: parallel=True to use better builtin mechanism to resolve call types. Resolves issue #3671

- PR #3725: Fix invalid removal of dead empty list

- PR #3740: add uintp as a valid type to the tuple operator.getitem

- PR #3758: Fix target definition update in inlining

- PR #3782: Raise typing error on yield optional.

- PR #3792: Fix non-module object used as the module of a function.

- PR #3800: Bugfix for np.interp

- PR #3808: Bump macro to include VS2014 to fix py3.5 build

- PR #3809: Add debug guard to debug only C function.

- PR #3816: Fix array.sum(axis) 1d input return type.

- PR #3821: Replace PySys_WriteStdout with PySys_FormatStdout to ensure no truncation.

- PR #3830: Getitem should not return optional type

- PR #3832: Handle single string as path in find_file()

Contributors:

- Ehsan Totoni

- Gryllos Prokopis

- Jonathan J. Helmus

- Kayla Ngan

- lalitparate

- luk-f-a

- Matyt

- Max Bolingbroke

- Michael Seifert

- Rob Ennis

- Siu Kwan Lam

- Stan Seibert

- Stuart Archibald

- Todd A. Anderson

- Tao He

- Valentin Haenel

10.15 Version 0.42.1

Bugfix release to fix the incorrect hash in OSX wheel packages. No change in source code.

10.16 Version 0.42.0

In this release the major features are:

- The capability to launch and attach the GDB debugger from within a jitted function.

- The upgrading of LLVM to version 7.0.0.

We added a draft of the project roadmap to the developer manual. The roadmap is for informational purposes only as priorities and resources may change.

Here are some enhancements from contributed PRs:

- #3532. Daniel Wennberg improved the `cuda.{pinned, mapped}` API so that the associated memory is released immediately at the exit of the context manager.

- #3531. Dimitri Vorona enabled the inlining of jitclass methods.

- #3516. Simon Perkins added the support for passing numpy dtypes (i.e. `np.dtype("int32")`) and their type constructor (i.e. `np.int32`) into a jitted function.

- #3509. Rob Ennis added support for `np.corrcoef`.

A regression issue (#3554, #3461) relating to making an empty slice in parallel mode is resolved by #3558.

General Enhancements:

- PR #3392: Launch and attach gdb directly from Numba.

- PR #3437: Changes to accommodate LLVM 7.0.x

- PR #3509: Support for np.corrcoef

- PR #3516: Typeof dtype values

- PR #3520: Fix @stencil ignoring cval if out kwarg supplied.

- PR #3531: Fix jitclass method inlining and avoid unnecessary increfs

- PR #3538: Avoid future C-level assertion error due to invalid visibility

- PR #3543: Avoid implementation error being hidden by the try-except

- PR #3544: Add *long_running* test flag and feature to exclude tests.

- PR #3549: ParallelAccelerator caching improvements

- PR #3558: Fixes array analysis for inplace binary operators.

- PR #3566: Skip alignment tests on armv7l.

- PR #3567: Fix unifying literal types in namedtuple

- PR #3576: Add special copy routine for NumPy out arrays

- PR #3577: Fix example and docs typos for *objmode* context manager. reorder statements.

- PR #3580: Use alias information when determining whether it is safe to

- PR #3583: Use *ir.unknown_loc* for unknown *Loc*, as #3390 with tests

- PR #3587: Fix llvm.memset usage changes in llvm7

- PR #3596: Fix Array Analysis for Global Namedtuples

- PR #3597: Warn users if threading backend init unsafe.

- PR #3605: Add guard for writing to read only arrays from ufunc calls

- PR #3606: Improve the accuracy of error message wording for undefined type.

- PR #3611: gdb test guard needs to ack ptrace permissions

- PR #3616: Skip gdb tests on ARM.

CUDA Enhancements:

- PR #3532: Unregister temporarily pinned host arrays at once

- PR #3552: Handle broadcast arrays correctly in host->device transfer.

- PR #3578: Align cuda and cuda simulator kwarg names.

Documentation Updates:

- PR #3545: Fix @njit description in 5 min guide

- PR #3570: Minor documentation fixes for numba.cuda

- PR #3581: Fixing minor typo in *reference/types.rst*

- PR #3594: Changing @*stencil* docs to correctly reflect *func_or_mode* param

- PR #3617: Draft roadmap as of Dec 2018

Contributors:

- Aaron Critchley

- Daniel Wennberg

- Dimitri Vorona

- Dominik Stańczak

- Ehsan Totoni (core dev)

- Iskander Sharipov

- Rob Ennis

- Simon Muller

- Simon Perkins

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

10.17 Version 0.41.0

This release adds the following major features:

- Diagnostics showing the optimizations done by ParallelAccelerator

- Support for profiling Numba-compiled functions in Intel VTune

- Additional NumPy functions: partition, nancumsum, nancumprod, ediff1d, cov, conj, conjugate, tri, tril, triu

- Initial support for Python 3 Unicode strings

General Enhancements:

- PR #1968: armv7 support
- PR #2983: invert mapping b/w binop operators and the operator module #2297
- PR #3160: First attempt at parallel diagnostics
- PR #3307: Adding NUMBA_ENABLE_PROFILING envvar, enabling jit event
- PR #3320: Support for np.partition
- PR #3324: Support for np.nancumsum and np.nancumprod
- PR #3325: Add location information to exceptions.
- PR #3337: Support for np.ediff1d
- PR #3345: Support for np.cov
- PR #3348: Support user pipeline class in with lifting
- PR #3363: string support
- PR #3373: Improve error message for empty imprecise lists.
- PR #3375: Enable overload(operator.getitem)
- PR #3402: Support negative indexing in tuple.
- PR #3414: Refactor Const type
- PR #3416: Optimized usage of alloca out of the loop
- PR #3424: Updates for llvmlite 0.26
- PR #3462: Add support for *np.conj/np.conjugate*.
- PR #3480: np.tri, np.tril, np.triu - default optional args
- PR #3481: Permit dtype argument as sole kwarg in np.eye

CUDA Enhancements:

- PR #3399: Add max_registers Option to cuda.jit

Continuous Integration / Testing:

- PR #3303: CI with Azure Pipelines
- PR #3309: Workaround race condition with apt
- PR #3371: Fix issues with Azure Pipelines
- PR #3362: Fix #3360: *RuntimeWarning: 'numba.runtests' found in sys.modules*
- PR #3374: Disable openmp in wheel building
- PR #3404: Azure Pipelines templates
- PR #3419: Fix cuda tests and error reporting in test discovery
- PR #3491: Prevent faulthandler installation on armv7l
- PR #3493: Fix CUDA test that used negative indexing behaviour that's fixed.
- PR #3495: Start Flake8 checking of Numba source

Fixes:

- PR #2950: Fix dispatcher to only consider contiguous-ness.
- PR #3124: Fix 3119, raise for 0d arrays in reductions

- PR #3228: Reduce redundant module linking
- PR #3329: Fix AOT on windows.
- PR #3335: Fix memory management of __cuda_array_interface__ views.
- PR #3340: Fix typo in error name.
- PR #3365: Fix the default unboxing logic
- PR #3367: Allow non-global reference to objmode() context-manager
- PR #3381: Fix global reference in objmode for dynamically created function
- PR #3382: CUDA_ERROR_MISALIGNED_ADDRESS Using Multiple Const Arrays
- PR #3384: Correctly handle very old versions of colorama
- PR #3394: Add 32bit package guard for non-32bit installs
- PR #3397: Fix with-objmode warning
- PR #3403 Fix label offset in call inline after parfor pass
- PR #3429: Fixes raising of user defined exceptions for exec(<string>).
- PR #3432: Fix error due to function naming in CI in py2.7
- PR #3444: Fixed TBB's single thread execution and test added for #3440
- PR #3449: Allow matching non-array objects in find_callname()
- PR #3455: Change getiter and iternext to not be pure. Resolves #3425
- PR #3467: Make ir.UndefinedType singleton class.
- PR #3478: Fix np.random.shuffle sideeffect
- PR #3487: Raise unsupported for kwargs given to *print()*
- PR #3488: Remove dead script.
- PR #3498: Fix stencil support for boolean as return type
- PR #3511: Fix handling make_function literals (regression of #3414)
- PR #3514: Add missing unicode != unicode
- PR #3527: Fix complex math sqrt implementation for large -ve values
- PR #3530: This adds arg an check for the pattern supplied to Parfors.
- PR #3536: Sets list dtor linkage to *linkonce_odr* to fix visibility in AOT

Documentation Updates:

- PR #3316: Update 0.40 changelog with additional PRs
- PR #3318: Tweak spacing to avoid search box wrapping onto second line
- PR #3321: Add note about memory leaks with exceptions to docs. Fixes #3263
- PR #3322: Add FAQ on CUDA + fork issue. Fixes #3315.
- PR #3343: Update docs for argsort, kind kwarg partially supported.
- PR #3357: Added mention of njit in 5minguide.rst
- PR #3434: Fix parallel reduction example in docs.
- PR #3452: Fix broken link and mark up problem.

- PR #3484: Size Numba logo in docs in em units. Fixes #3313

- PR #3502: just two typos

- PR #3506: Document string support

- PR #3513: Documentation for parallel diagnostics.

- PR #3526: Fix 5 min guide with respect to @njit decl

Contributors:

- Alex Ford

- Andreas Sodeur

- Anton Malakhov

- Daniel Stender

- Ehsan Totoni (core dev)

- Henry Schreiner

- Marcel Bargull

- Matt Cooper

- Nick White

- Nicolas Hug

- rjenc29

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Todd A. Anderson (core dev)

10.18 Version 0.40.1

This is a PyPI-only patch release to ensure that PyPI wheels can enable the TBB threading backend, and to disable the OpenMP backend in the wheels. Limitations of manylinux1 and variation in user environments can cause segfaults when OpenMP is enabled on wheel builds. Note that this release has no functional changes for users who obtained Numba 0.40.0 via conda.

Patches:

- PR #3338: Accidentally left Anton off contributor list for 0.40.0

- PR #3374: Disable OpenMP in wheel building

- PR #3376: Update 0.40.1 changelog and docs on OpenMP backend

10.19 Version 0.40.0

This release adds a number of major features:

- A new GPU backend: kernels for AMD GPUs can now be compiled using the ROCm driver on Linux.

- The thread pool implementation used by Numba for automatic multithreading is configurable to use TBB, OpenMP, or the old "workqueue" implementation. (TBB is likely to become the preferred default in a future release.)

- New documentation on thread and fork-safety with Numba, along with overall improvements in thread-safety.

- Experimental support for executing a block of code inside a nopython mode function in object mode.

- Parallel loops now allow arrays as reduction variables

- CUDA improvements: FMA, faster float64 atomics on supporting hardware, records in const memory, and improved datatime dtype support

- More NumPy functions: vander, tri, triu, tril, fill_diagonal

General Enhancements:

- PR #3017: Add facility to support with-contexts

- PR #3033: Add support for multidimensional CFFI arrays

- PR #3122: Add inliner to object mode pipeline

- PR #3127: Support for reductions on arrays.

- PR #3145: Support for np.fill_diagonal

- PR #3151: Keep a queue of references to last N deserialized functions. Fixes #3026

- PR #3154: Support use of list() if typeable.

- PR #3166: Objmode with-block

- PR #3179: Updates for llvmlite 0.25

- PR #3181: Support function extension in alias analysis

- PR #3189: Support literal constants in typing of object methods

- PR #3190: Support passing closures as literal values in typing

- PR #3199: Support inferring stencil index as constant in simple unary expressions

- PR #3202: Threading layer backend refactor/rewrite/reinvention!

- PR #3209: Support for np.tri, np.tril and np.triu

- PR #3211: Handle unpacking in building tuple (BUILD_TUPLE_UNPACK opcode)

- PR #3212: Support for np.vander

- PR #3227: Add NumPy 1.15 support

- PR #3272: Add MemInfo_data to runtime._nrt_python.c_helpers

- PR #3273: Refactor. Removing thread-local-storage based context nesting.

- PR #3278: compiler threadsafety lockdown

- PR #3291: Add CPU count and CFS restrictions info to numba -s.

CUDA Enhancements:

- PR #3152: Use cuda driver api to get best blocksize for best occupancy

- PR #3165: Add FMA intrinsic support

- PR #3172: Use float64 add Atomics, Where Available

- PR #3186: Support Records in CUDA Const Memory

- PR #3191: CUDA: fix log size
- PR #3198: Fix GPU datetime timedelta types usage
- PR #3221: Support datetime/timedelta scalar argument to a CUDA kernel.
- PR #3259: Add DeviceNDArray.view method to reinterpret data as a different type.
- PR #3310: Fix IPC handling of sliced cuda array.

ROCm Enhancements:

- PR #3023: Support for AMDGCN/ROCm.
- PR #3108: Add ROC info to *numba -s* output.
- PR #3176: Move ROC vectorize init to npyufunc
- PR #3177: Add auto_synchronize support to ROC stream
- PR #3178: Update ROC target documentation.
- PR #3294: Add compiler lock to ROC compilation path.
- PR #3280: Add wavebits property to the HSA Agent.
- PR #3281: Fix ds_permute types and add tests

Continuous Integration / Testing:

- PR #3091: Remove old recipes, switch to test config based on env var.
- PR #3094: Add higher ULP tolerance for products in complex space.
- PR #3096: Set exit on error in incremental scripts
- PR #3109: Add skip to test needing jinja2 if no jinja2.
- PR #3125: Skip cudasim only tests
- PR #3126: add slack, drop flowdock
- PR #3147: Improve error message for arg type unsupported during typing.
- PR #3128: Fix recipe/build for jetson tx2/ARM
- PR #3167: In build script activate env before installing.
- PR #3180: Add skip to broken test.
- PR #3216: Fix libcuda.so loading in some container setup
- PR #3224: Switch to new Gitter notification webhook URL and encrypt it
- PR #3235: Add 32bit Travis CI jobs
- PR #3257: This adds scipy/ipython back into windows conda test phase.

Fixes:

- PR #3038: Fix random integer generation to match results from NumPy.
- PR #3045: Fix #3027 - Numba reassigns sys.stdout
- PR #3059: Handler for known LoweringErrors.
- PR #3060: Adjust attribute error for NumPy functions.
- PR #3067: Abort simulator threads on exception in thread block.
- PR #3079: Implement +/-(types.boolean) Fix #2624

- PR #3080: Compute np.var and np.std correctly for complex types.
- PR #3088: Fix #3066 (array.dtype.type in prange)
- PR #3089: Fix invalid ParallelAccelerator hoisting issue.
- PR #3136: Fix #3135 (lowering error)
- PR #3137: Fix for issue3103 (race condition detection)
- PR #3142: Fix Issue #3139 (parfors reuse of reduction variable across prange blocks)
- PR #3148: Remove dead array equal @infer code
- PR #3153: Fix canonicalize_array_math typing for calls with kw args
- PR #3156: Fixes issue with missing pygments in testing and adds guards.
- PR #3168: Py37 bytes output fix.
- PR #3171: Fix #3146. Fix CFUNCTYPE void* return-type handling
- PR #3193: Fix setitem/getitem resolvers
- PR #3222: Fix #3214. Mishandling of POP_BLOCK in while True loop.
- PR #3230: Fixes liveness analysis issue in looplifting
- PR #3233: Fix return type difference for 32bit ctypes.c_void_p
- PR #3234: Fix types and layout for *np.where*.
- PR #3237: Fix DeprecationWarning about imp module
- PR #3241: Fix #3225. Normalize 0nd array to scalar in typing of indexing code.
- PR #3256: Fix #3251: Move imports of ABCs to collections.abc for Python >= 3.3
- PR #3292: Fix issue3279.
- PR #3302: Fix error due to mismatching dtype

Documentation Updates:

- PR #3104: Workaround for #3098 (test_optional_unpack Heisenbug)
- PR #3132: Adds an ~5 minute guide to Numba.
- PR #3194: Fix docs RE: np.random generator fork/thread safety
- PR #3242: Page with Numba talks and tutorial links
- PR #3258: Allow users to choose the type of issue they are reporting.
- PR #3260: Fixed broken link
- PR #3266: Fix cuda pointer ownership problem with user/externally allocated pointer
- PR #3269: Tweak typography with CSS
- PR #3270: Update FAQ for functions passed as arguments
- PR #3274: Update installation instructions
- PR #3275: Note pyobject and voidptr are types in docs
- PR #3288: Do not need to call parallel optimizations "experimental" anymore
- PR #3318: Tweak spacing to avoid search box wrapping onto second line

Contributors:

- Anton Malakhov

- Alex Ford

- Anthony Bisulco

- Ehsan Totoni (core dev)

- Leonard Lausen

- Matthew Petroff

- Nick White

- Ray Donnelly

- rjenc29

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Stuart Reynolds

- Todd A. Anderson (core dev)

10.20 Version 0.39.0

Here are the highlights for the Numba 0.39.0 release.

- This is the first version that supports Python 3.7.

- With help from Intel, we have fixed the issues with SVML support (related issues #2938, #2998, #3006).

- List has gained support for containing reference-counted types like NumPy arrays and *list*. Note, list still cannot hold heterogeneous types.

- We have made a significant change to the internal calling-convention, which should be transparent to most users, to allow for a future feature that will permitting jumping back into python-mode from a nopython-mode function. This also fixes a limitation to *print* that disabled its use from nopython functions that were deep in the call-stack.

- For CUDA GPU support, we added a *__cuda_array_interface__* following the NumPy array interface specification to allow Numba to consume externally defined device arrays. We have opened a corresponding pull request to CuPy to test out the concept and be able to use a CuPy GPU array.

- The Numba dispatcher *inspect_types()* method now supports the kwarg *pretty* which if set to *True* will produce ANSI/HTML output, showing the annotated types, when invoked from ipython/jupyter-notebook respectively.

- The NumPy functions *ndarray.dot*, *np.percentile* and *np.nanpercentile*, and *np.unique* are now supported.

- Numba now supports the use of a per-project configuration file to permanently set behaviours typically set via *NUMBA_** family environment variables.

- Support for the *ppc64le* architecture has been added.

Enhancements:

- PR #2793: Simplify and remove javascript from html_annotate templates.

- PR #2840: Support list of refcounted types

- PR #2902: Support for np.unique

- PR #2926: Enable fence for all architecture and add developer notes
- PR #2928: Making error about untyped list more informative.
- PR #2930: Add configuration file and color schemes.
- PR #2932: Fix encoding to 'UTF-8' in *check_output* decode.
- PR #2938: Python 3.7 compat: _Py_Finalizing becomes _Py_IsFinalizing()
- PR #2939: Comprehensive SVML unit test
- PR #2946: Add support for *ndarray.dot* method and tests.
- PR #2953: percentile and nanpercentile
- PR #2957: Add new 3.7 opcode support.
- PR #2963: Improve alias analysis to be more comprehensive
- PR #2984: Support for namedtuples in array analysis
- PR #2986: Fix environment propagation
- PR #2990: Improve function call matching for intrinsics
- PR #3002: Second pass at error rewrites (interpreter errors).
- PR #3004: Add numpy.empty to the list of pure functions.
- PR #3008: Augment SVML detection with llvmlite SVML patch detection.
- PR #3012: Make use of the common spelling of heterogeneous/homogeneous.
- PR #3032: Fix pycc ctypes test due to mismatch in calling-convention
- PR #3039: Add SVML detection to Numba environment diagnostic tool.
- PR #3041: This adds @needs_blas to tests that use BLAS
- PR #3056: Require llvmlite>=0.24.0

CUDA Enhancements:

- PR #2860: __cuda_array_interface__
- PR #2910: More CUDA intrinsics
- PR #2929: Add Flag To Prevent Unneccessary D->H Copies
- PR #3037: Add CUDA IPC support on non-peer-accessible devices

CI Enhancements:

- PR #3021: Update appveyor config.
- PR #3040: Add fault handler to all builds
- PR #3042: Add catchsegv
- PR #3077: Adds optional number of processes for *-m* in testing

Fixes:

- PR #2897: Fix line position of delete statement in numba ir
- PR #2905: Fix for #2862
- PR #3009: Fix optional type returning in recursive call
- PR #3019: workaround and unittest for issue #3016

- PR #3035: [TESTING] Attempt delayed removal of Env
- PR #3048: [WIP] Fix cuda tests failure on buildfarm
- PR #3054: Make test work on 32-bit
- PR #3062: Fix cuda.In freeing devary before the kernel launch
- PR #3073: Workaround #3072
- PR #3076: Avoid ignored exception due to missing globals at interpreter teardown

Documentation Updates:

- PR #2966: Fix syntax in env var docs.
- PR #2967: Fix typo in CUDA kernel layout example.
- PR #2970: Fix docstring copy paste error.

Contributors:

The following people contributed to this release.

- Anton Malakhov
- Ehsan Totoni (core dev)
- Julia Tatz
- Matthias Bussonnier
- Nick White
- Ray Donnelly
- Siu Kwan Lam (core dev)
- Stan Seibert (core dev)
- Stuart Archibald (core dev)
- Todd A. Anderson (core dev)
- Rik-de-Kort
- rjenc29

10.21 Version 0.38.1

This is a critical bug fix release addressing: https://github.com/numba/numba/issues/3006

The bug does not impact users using conda packages from Anaconda or Intel Python Distribution (but it does impact conda-forge). It does not impact users of pip using wheels from PyPI.

This only impacts a small number of users where:

- The ICC runtime (specifically libsvml) is present in the user's environment.
- The user is using an llvmlite statically linked against a version of LLVM that has not been patched with SVML support.
- The platform is 64-bit.

The release fixes a code generation path that could lead to the production of incorrect results under the above situation.

Fixes:

- PR #3007: Augment SVML detection with llvmlite SVML patch detection.

Contributors:

The following people contributed to this release.

- Stuart Archibald (core dev)

10.22 Version 0.38.0

Following on from the bug fix focus of the last release, this release swings back towards the addition of new features and usability improvements based on community feedback. This release is comparatively large! Three key features/ changes to note are:

- Numba (via llvmlite) is now backed by LLVM 6.0, general vectorization is improved as a result. A significant long standing LLVM bug that was causing corruption was also found and fixed.

- Further considerable improvements in vectorization are made available as Numba now supports Intel's short vector math library (SVML). Try it out with *conda install -c numba icc_rt*.

- CUDA 8.0 is now the minimum supported CUDA version.

Other highlights include:

- Bug fixes to *parallel=True* have enabled more vectorization opportunities when using the ParallelAccelerator technology.

- Much effort has gone into improving error reporting and the general usability of Numba. This includes highlighted error messages and performance tips documentation. Try it out with *conda install colorama*.

- A number of new NumPy functions are supported, *np.convolve*, *np.correlate np.reshape*, *np.transpose*, *np.permutation*, *np.real*, *np.imag*, and *np.searchsorted* now supports the'side' kwarg. Further, *np.argsort* now supports the *kind* kwarg with *quicksort* and *mergesort* available.

- The Numba extension API has gained the ability operate more easily with functions from Cython modules through the use of *numba.extending.get_cython_function_address* to obtain function addresses for direct use in *ctypes.CFUNCTYPE*.

- Numba now allows the passing of jitted functions (and containers of jitted functions) as arguments to other jitted functions.

- The CUDA functionality has gained support for a larger selection of bit manipulation intrinsics, also SELP, and has had a number of bugs fixed.

- Initial work to support the PPC64LE platform has been added, full support is however waiting on the LLVM 6.0.1 release as it contains critical patches not present in 6.0.0. It is hoped that any remaining issues will be fixed in the next release.

- The capacity for advanced users/compiler engineers to define their own compilation pipelines.

Enhancements:

- PR #2660: Support bools from cffi in nopython.
- PR #2741: Enhance error message for undefined variables.
- PR #2744: Add diagnostic error message to test suite discovery failure.
- PR #2748: Added Intel SVML optimizations as opt-out choice working by default
- PR #2762: Support transpose with axes arguments.
- PR #2777: Add support for np.correlate and np.convolve

- PR #2779: Implement np.random.permutation
- PR #2801: Passing jitted functions as args
- PR #2802: Support np.real() and np.imag()
- PR #2807: Expose *import_cython_function*
- PR #2821: Add kwarg 'side' to np.searchsorted
- PR #2822: Adds stable argsort
- PR #2832: Fixups for llvmlite 0.23/llvm 6
- PR #2836: Support *index* method on tuples
- PR #2839: Support for np.transpose and np.reshape.
- PR #2843: Custom pipeline
- PR #2847: Replace signed array access indices in unsiged prange loop body
- PR #2859: Add support for improved error reporting.
- PR #2880: This adds a github issue template.
- PR #2881: Build recipe to clone Intel ICC runtime.
- PR #2882: Update TravisCI to test SVML
- PR #2893: Add reference to the data buffer in array.ctypes object
- PR #2895: Move to CUDA 8.0

Fixes:

- PR #2737: Fix #2007 (part 1). Empty array handling in np.linalg.
- PR #2738: Fix install_requires to allow pip getting pre-release version
- PR #2740: Fix 2208. Generate better error message.
- PR #2765: Fix Bit-ness
- PR #2780: PowerPC reference counting memory fences
- PR #2805: Fix six imports.
- PR #2813: Fix #2812: gufunc scalar output bug.
- PR #2814: Fix the build post #2727
- PR #2831: Attempt to fix #2473
- PR #2842: Fix issue with test discovery and broken CUDA drivers.
- PR #2850: Add rtsys init guard and test.
- PR #2852: Skip vectorization test with targets that are not x86
- PR #2856: Prevent printing to stdout in *test_extending.py*
- PR #2864: Correct C code to prevent compiler warnings.
- PR #2889: Attempt to fix #2386.
- PR #2891: Removed test skipping for inspect_cfg
- PR #2898: Add guard to parallel test on unsupported platforms
- PR #2907: Update change log for PPC64LE LLVM dependency.

- PR #2911: Move build requirement to llvmlite>=0.23.0dev0
- PR #2912: Fix random permutation test.
- PR #2914: Fix MD list syntax in issue template.

Documentation Updates:

- PR #2739: Explicitly state default value of error_model in docstring
- PR #2803: DOC: parallel vectorize requires signatures
- PR #2829: Add Python 2.7 EOL plan to docs
- PR #2838: Use automatic numbering syntax in list.
- PR #2877: Add performance tips documentation.
- PR #2883: Fix #2872: update rng doc about thread/fork-safety
- PR #2908: Add missing link and ref to docs.
- PR #2909: Tiny typo correction

ParallelAccelerator enhancements/fixes:

- PR #2727: Changes to enable vectorization in ParallelAccelerator.
- PR #2816: Array analysis for transpose with arbitrary arguments
- PR #2874: Fix dead code eliminator not to remove a call with side-effect
- PR #2886: Fix ParallelAccelerator arrayexpr repr

CUDA enhancements:

- PR #2734: More Constants From cuda.h
- PR #2767: Add len(..) Support to DeviceNDArray
- PR #2778: Add More Device Array API Functions to CUDA Simulator
- PR #2824: Add CUDA Primitives for Population Count
- PR #2835: Emit selp Instructions to Avoid Branching
- PR #2867: Full support for CUDA device attributes

CUDA fixes: * PR #2768: Don't Compile Code on Every Assignment * PR #2878: Fixes a Win64 issue with the test in Pr/2865

Contributors:

The following people contributed to this release.

- Abutalib Aghayev
- Alex Olivas
- Anton Malakhov
- Dong-hee Na
- Ehsan Totoni (core dev)
- John Zwinck
- Josh Wilson
- Kelsey Jordahl

- Nick White

- Olexa Bilaniuk

- Rik-de-Kort

- Siu Kwan Lam (core dev)

- Stan Seibert (core dev)

- Stuart Archibald (core dev)

- Thomas Arildsen

- Todd A. Anderson (core dev)

10.23 Version 0.37.0

This release focuses on bug fixing and stability but also adds a few new features including support for Numpy 1.14. The key change for Numba core was the long awaited addition of the final tranche of thread safety improvements that allow Numba to be run concurrently on multiple threads without hitting known thread safety issues inside LLVM itself. Further, a number of fixes and enhancements went into the CUDA implementation and ParallelAccelerator gained some new features and underwent some internal refactoring.

Misc enhancements:

- PR #2627: Remove hacks to make llvmlite threadsafe

- PR #2672: Add ascontiguousarray

- PR #2678: Add Gitter badge

- PR #2691: Fix #2690: add intrinsic to convert array to tuple

- PR #2703: Test runner feature: failed-first and last-failed

- PR #2708: Patch for issue #1907

- PR #2732: Add support for array.fill

Misc Fixes:

- PR #2610: Fix #2606 lowering of optional.setattr

- PR #2650: Remove skip for win32 cosine test

- PR #2668: Fix empty_like from readonly arrays.

- PR #2682: Fixes 2210, remove _DisableJitWrapper

- PR #2684: Fix #2340, generator error yielding bool

- PR #2693: Add travis-ci testing of NumPy 1.14, and also check on Python 2.7

- PR #2694: Avoid type inference failure due to a typing template rejection

- PR #2695: Update llvmlite version dependency.

- PR #2696: Fix tuple indexing codegeneration for empty tuple

- PR #2698: Fix #2697 by deferring deletion in the simplify_CFG loop.

- PR #2701: Small fix to avoid tempfiles being created in the current directory

- PR #2725: Fix 2481, LLVM IR parsing error due to mutated IR

- PR #2726: Fix #2673: incorrect fork error msg.

- PR #2728: Alternative to #2620. Remove dead code ByteCodeInst.get.
- PR #2730: Add guard for test needing SciPy/BLAS

Documentation updates:

- PR #2670: Update communication channels
- PR #2671: Add docs about diagnosing loop vectorizer
- PR #2683: Add docs on const arg requirements and on const mem alloc
- PR #2722: Add docs on numpy support in cuda
- PR #2724: Update doc: warning about unsupported arguments

ParallelAccelerator enhancements/fixes:

Parallel support for *np.arange* and *np.linspace*, also *np.mean*, *np.std* and *np.var* are added. This was performed as part of a general refactor and cleanup of the core ParallelAccelerator code.

- PR #2674: Core pa
- PR #2704: Generate Dels after parfor sequential lowering
- PR #2716: Handle matching directly supported functions

CUDA enhancements:

- PR #2665: CUDA DeviceNDArray: Support numpy tranpose API
- PR #2681: Allow Assigning to DeviceNDArrays
- PR #2702: Make DummyArray do High Dimensional Reshapes
- PR #2714: Use CFFI to Reuse Code

CUDA fixes:

- PR #2667: Fix CUDA DeviceNDArray slicing
- PR #2686: Fix #2663: incorrect offset when indexing cuda array.
- PR #2687: Ensure Constructed Stream Bound
- PR #2706: Workaround for unexpected warp divergence due to exception raising code
- PR #2707: Fix regression: cuda test submodules not loading properly in runtests
- PR #2731: Use more challenging values in slice tests.
- PR #2720: A quick testsuite fix to not run the new cuda testcase in the multiprocess pool

Contributors:

The following people contributed to this release.

- Coutinho Menezes Nilo
- Daniel
- Ehsan Totoni
- Nick White
- Paul H. Liu
- Siu Kwan Lam
- Stan Seibert

- Stuart Archibald

- Todd A. Anderson

10.24 Version 0.36.2

This is a bugfix release that provides minor changes to address:

- PR #2645: Avoid CPython bug with `exec` in older 2.7.x.

- PR #2652: Add support for CUDA 9.

10.25 Version 0.36.1

This release continues to add new features to the work undertaken in partnership with Intel on ParallelAccelerator technology. Other changes of note include the compilation chain being updated to use LLVM 5.0 and the production of conda packages using conda-build 3 and the new compilers that ship with it.

NOTE: A version 0.36.0 was tagged for internal use but not released.

ParallelAccelerator:

NOTE: The ParallelAccelerator technology is under active development and should be considered experimental.

New features relating to ParallelAccelerator, from work undertaken with Intel, include the addition of the *@stencil* decorator for ease of implementation of stencil-like computations, support for general reductions, and slice and range fusion for parallel slice/bit-array assignments. Documentation on both the use and implementation of the above has been added. Further, a new debug environment variable *NUMBA_DEBUG_ARRAY_OPT_STATS* is made available to give information about which operators/calls are converted to parallel for-loops.

ParallelAccelerator features:

- PR #2457: Stencil Computations in ParallelAccelerator

- PR #2548: Slice and range fusion, parallelizing bitarray and slice assignment

- PR #2516: Support general reductions in ParallelAccelerator

ParallelAccelerator fixes:

- PR #2540: Fix bug #2537

- PR #2566: Fix issue #2564.

- PR #2599: Fix nested multi-dimensional parfor type inference issue

- PR #2604: Fixes for stencil tests and cmath sin().

- PR #2605: Fixes issue #2603.

Additional features of note:

This release of Numba (and llvmlite) is updated to use LLVM version 5.0 as the compiler back end, the main change to Numba to support this was the addition of a custom symbol tracker to avoid the calls to LLVM's *ExecutionEngine* that was crashing when asking for non-existent symbol addresses. Further, the conda packages for this release of Numba are built using conda build version 3 and the new compilers/recipe grammar that are present in that release.

- PR #2568: Update for LLVM 5

- PR #2607: Fixes abort when getting address to "nrt_unresolved_abort"

- PR #2615: Working towards conda build 3

Thanks to community feedback and bug reports, the following fixes were also made.

Misc fixes/enhancements:

- PR #2534: Add tuple support to np.take.
- PR #2551: Rebranding fix
- PR #2552: relative doc links
- PR #2570: Fix issue #2561, handle missing successor on loop exit
- PR #2588: Fix #2555. Disable libpython.so linking on linux
- PR #2601: Update llvmlite version dependency.
- PR #2608: Fix potential cache file collision
- PR #2612: Fix NRT test failure due to increased overhead when running in coverage
- PR #2619: Fix dubious pthread_cond_signal not in lock
- PR #2622: Fix *np.nanmedian* for all NaN case.
- PR #2633: Fix markdown in CONTRIBUTING.md
- PR #2635: Make the dependency on compilers for AOT optional.

CUDA support fixes:

- PR #2523: Fix invalid cuda context in memory transfer calls in another thread
- PR #2575: Use CPU to initialize xoroshiro states for GPU RNG. Fixes #2573
- PR #2581: Fix cuda gufunc mishandling of scalar arg as array and out argument

10.26 Version 0.35.0

This release includes some exciting new features as part of the work performed in partnership with Intel on ParallelAccelerator technology. There are also some additions made to Numpy support and small but significant fixes made as a result of considerable effort spent chasing bugs and implementing stability improvements.

ParallelAccelerator:

NOTE: The ParallelAccelerator technology is under active development and should be considered experimental.

New features relating to ParallelAccelerator, from work undertaken with Intel, include support for a larger range of *np.random* functions in *parallel* mode, printing Numpy arrays in no Python mode, the capacity to initialize Numpy arrays directly from list comprehensions, and the axis argument to *.sum()*. Documentation on the ParallelAccelerator technology implementation has also been added. Further, a large amount of work on equivalence relations was undertaken to enable runtime checks of broadcasting behaviours in parallel mode.

ParallelAccelerator features:

- PR #2400: Array comprehension
- PR #2405: Support printing Numpy arrays
- PR #2438: from Support more np.random functions in ParallelAccelerator
- PR #2482: Support for sum with axis in nopython mode.
- PR #2487: Adding developer documentation for ParallelAccelerator technology.
- PR #2492: Core PA refactor adds assertions for broadcast semantics

ParallelAccelerator fixes:

- PR #2478: Rename cfg before parfor translation (#2477)

- PR #2479: Fix broken array comprehension tests on unsupported platforms

- PR #2484: Fix array comprehension test on win64

- PR #2506: Fix for 32-bit machines.

Additional features of note:

Support for *np.take*, *np.finfo*, *np.iinfo* and *np.MachAr* in no Python mode is added. Further, three new environment variables are added, two for overriding CPU target/features and another to warn if *parallel=True* was set no such transform was possible.

- PR #2490: Implement np.take and ndarray.take

- PR #2493: Display a warning if parallel=True is set but not possible.

- PR #2513: Add np.MachAr, np.finfo, np.iinfo

- PR #2515: Allow environ overriding of cpu target and cpu features.

Due to expansion of the test farm and a focus on fixing bugs, the following fixes were also made.

Misc fixes/enhancements:

- PR #2455: add contextual information to runtime errors

- PR #2470: Fixes #2458, poor performance in np.median

- PR #2471: Ensure LLVM threadsafety in {g,}ufunc building.

- PR #2494: Update doc theme

- PR #2503: Remove hacky code added in 2482 and feature enhancement

- PR #2505: Serialise env mutation tests during multithreaded testing.

- PR #2520: Fix failing cpu-target override tests

CUDA support fixes:

- PR #2504: Enable CUDA toolkit version testing

- PR #2509: Disable tests generating code unavailable in lower CC versions.

- PR #2511: Fix Windows 64 bit CUDA tests.

10.27 Version 0.34.0

This release adds a significant set of new features arising from combined work with Intel on ParallelAccelerator technology. It also adds list comprehension and closure support, support for Numpy 1.13 and a new, faster, CUDA reduction algorithm. For Linux users this release is the first to be built on Centos 6, which will be the new base platform for future releases. Finally a number of thread-safety, type inference and other smaller enhancements and bugs have been fixed.

ParallelAccelerator features:

NOTE: The ParallelAccelerator technology is under active development and should be considered experimental.

The ParallelAccelerator technology is accessed via a new "nopython" mode option "parallel". The ParallelAccelerator technology attempts to identify operations which have parallel semantics (for instance adding a scalar to a vector),

fuse together adjacent such operations, and then parallelize their execution across a number of CPU cores. This is essentially auto-parallelization.

In addition to the auto-parallelization feature, explicit loop based parallelism is made available through the use of *prange* in place of *range* as a loop iterator.

More information and examples on both auto-parallelization and *prange* are available in the documentation and examples directory respectively.

As part of the necessary work for ParallelAccelerator, support for closures and list comprehensions is added:

- PR #2318: Transfer ParallelAccelerator technology to Numba
- PR #2379: ParallelAccelerator Core Improvements
- PR #2367: Add support for len(range(. . .))
- PR #2369: List comprehension
- PR #2391: Explicit Parallel Loop Support (prange)

The ParallelAccelerator features are available on all supported platforms and Python versions with the exceptions of (with view of supporting in a future release):

- The combination of Windows operating systems with Python 2.7.
- Systems running 32 bit Python.

CUDA support enhancements:

- PR #2377: New GPU reduction algorithm

CUDA support fixes:

- PR #2397: Fix #2393, always set alignment of cuda static memory regions

Misc Fixes:

- PR #2373, Issue #2372: 32-bit compatibility fix for parfor related code
- PR #2376: Fix #2375 missing stdint.h for py2.7 vc9
- PR #2378: Fix deadlock in parallel gufunc when kernel acquires the GIL.
- PR #2382: Forbid unsafe casting in bitwise operation
- PR #2385: docs: fix Sphinx errors
- PR #2396: Use 64-bit RHS operand for shift
- PR #2404: Fix threadsafety logic issue in ufunc compilation cache.
- PR #2424: Ensure consistent iteration order of blocks for type inference.
- PR #2425: Guard code to prevent the use of 'parallel' on win32 + py27
- PR #2426: Basic test for Enum member type recovery.
- PR #2433: Fix up the parfors tests with respect to windows py2.7
- PR #2442: Skip tests that need BLAS/LAPACK if scipy is not available.
- PR #2444: Add test for invalid array setitem
- PR #2449: Make the runtime initialiser threadsafe
- PR #2452: Skip CFG test on 64bit windows

Misc Enhancements:

- PR #2366: Improvements to IR utils

- PR #2388: Update README.rst to indicate the proper version of LLVM

- PR #2394: Upgrade to llvmlite 0.19.*

- PR #2395: Update llvmlite version to 0.19

- PR #2406: Expose environment object to ufuncs

- PR #2407: Expose environment object to target-context inside lowerer

- PR #2413: Add flags to pass through to conda build for buildbot

- PR #2414: Add cross compile flags to local recipe

- PR #2415: A few cleanups for rewrites

- PR #2418: Add getitem support for Enum classes

- PR #2419: Add support for returning enums in vectorize

- PR #2421: Add copyright notice for Intel contributed files.

- PR #2422: Patch code base to work with np 1.13 release

- PR #2448: Adds in warning message when using 'parallel' if cache=True

- PR #2450: Add test for keyword arg on .sum-like and .cumsum-like array methods

10.28 Version 0.33.0

This release resolved several performance issues caused by atomic reference counting operations inside loop bodies. New optimization passes have been added to reduce the impact of these operations. We observe speed improvements between 2x-10x in affected programs due to the removal of unnecessary reference counting operations.

There are also several enhancements to the CUDA GPU support:

- A GPU random number generator based on xoroshiro128+ algorithm is added. See details and examples in *documentation*.

- @cuda.jit CUDA kernels can now call @jit and @njit CPU functions and they will automatically be compiled as CUDA device functions.

- CUDA IPC memory API is exposed for sharing memory between proceses. See usage details in *documentation*.

Reference counting enhancements:

- PR #2346, Issue #2345, #2248: Add extra refcount pruning after inlining

- PR #2349: Fix refct pruning not removing refct op with tail call.

- PR #2352, Issue #2350: Add refcount pruning pass for function that does not need refcount

CUDA support enhancements:

- PR #2023: Supports CUDA IPC for device array

- PR #2343, Issue #2335: Allow CPU jit decorated function to be used as cuda device function

- PR #2347: Add random number generator support for CUDA device code

- PR #2361: Update autotune table for CC: 5.3, 6.0, 6.1, 6.2

Misc fixes:

- PR #2362: Avoid test failure due to typing to int32 on 32-bit platforms

- PR #2359: Fixed nogil example that threw a TypeError when executed.

- PR #2357, Issue #2356: Fix fragile test that depends on how the script is executed.

- PR #2355: Fix cpu dispatcher referenced as attribute of another module

- PR #2354: Fixes an issue with caching when function needs NRT and refcount pruning

- PR #2342, Issue #2339: Add warnings to inspection when it is used on unserialized cached code

- PR #2329, Issue #2250: Better handling of missing op codes

Misc enhancements:

- PR #2360: Adds missing values in error mesasge interp.

- PR #2353: Handle when get_host_cpu_features() raises RuntimeError

- PR #2351: Enable SVML for erf/erfc/gamma/lgamma/log2

- PR #2344: Expose error_model setting in jit decorator

- PR #2337: Align blocking terminate support for fork() with new TBB version

- PR #2336: Bump llvmlite version to 0.18

- PR #2330: Core changes in PR #2318

10.29 Version 0.32.0

In this release, we are upgrading to LLVM 4.0. A lot of work has been done to fix many race-condition issues inside LLVM when the compiler is used concurrently, which is likely when Numba is used with Dask.

Improvements:

- PR #2322: Suppress test error due to unknown but consistent error with tgamma

- PR #2320: Update llvmlite dependency to 0.17

- PR #2308: Add details to error message on why cuda support is disabled.

- PR #2302: Add os x to travis

- PR #2294: Disable remove_module on MCJIT due to memory leak inside LLVM

- PR #2291: Split parallel tests and recycle workers to tame memory usage

- PR #2253: Remove the pointer-stuffing hack for storing meminfos in lists

Fixes:

- PR #2331: Fix a bug in the GPU array indexing

- PR #2326: Fix #2321 docs referring to non-existing function.

- PR #2316: Fixing more race-condition problems

- PR #2315: Fix #2314. Relax strict type check to allow optional type.

- PR #2310: Fix race condition due to concurrent compilation and cache loading

- PR #2304: Fix intrinsic 1st arg not a typing.Context as stated by the docs.

- PR #2287: Fix int64 atomic min-max

- PR #2286: Fix #2285 @*overload_method* not linking dependent libs

- PR #2303: Missing import statements to interval-example.rst

10.30 Version 0.31.0

In this release, we added preliminary support for debugging with GDB version >= 7.0. The feature is enabled by setting the `debug=True` compiler option, which causes GDB compatible debug info to be generated. The CUDA backend also gained limited debugging support so that source locations are showed in memory-checking and profiling tools. For details, see *Troubleshooting and tips*.

Also, we added the `fastmath=True` compiler option to enable unsafe floating-point transformations, which allows LLVM to auto-vectorize more code.

Other important changes include upgrading to LLVM 3.9.1 and adding support for Numpy 1.12.

Improvements:

- PR #2281: Update for numpy1.12
- PR #2278: Add CUDA atomic.{max, min, compare_and_swap}
- PR #2277: Add about section to conda recipies to identify license and other metadata in Anaconda Cloud
- PR #2271: Adopt itanium C++-style mangling for CPU and CUDA targets
- PR #2267: Add fastmath flags
- PR #2261: Support dtype.type
- PR #2249: Changes for llvm3.9
- PR #2234: Bump llvmlite requirement to 0.16 and add install_name_tool_fixer to mviewbuf for OS X
- PR #2230: Add python3.6 to TravisCi
- PR #2227: Enable caching for gufunc wrapper
- PR #2170: Add debugging support
- PR #2037: inspect_cfg() for easier visualization of the function operation

Fixes:

- PR #2274: Fix nvvm ir patch in mishandling "load"
- PR #2272: Fix breakage to cuda7.5
- PR #2269: Fix caching of copy_strides kernel in cuda.reduce
- PR #2265: Fix #2263: error when linking two modules with dynamic globals
- PR #2252: Fix path separator in test
- PR #2246: Fix overuse of memory in some system with fork
- PR #2241: Fix #2240: __module__ in dynamically created function not a str
- PR #2239: Fix fingerprint computation failure preventing fallback

10.31 Version 0.30.1

This is a bug-fix release to enable Python 3.6 support. In addition, there is now early Intel TBB support for parallel ufuncs when building from source with TBBROOT defined. The TBB feature is not enabled in our official builds.

Fixes:

- PR #2232: Fix name clashes with _Py_hashtable_xxx in Python 3.6.

Improvements:

- PR #2217: Add Intel TBB threadpool implementation for parallel ufunc.

10.32 Version 0.30.0

This release adds preliminary support for Python 3.6, but no official build is available yet. A new system reporting tool
(`numba --sysinfo`) is added to provide system information to help core developers in replication and debugging.
See below for other improvements and bug fixes.

Improvements:

- PR #2209: Support Python 3.6.
- PR #2175: Support `np.trace()`, `np.outer()` and `np.kron()`.
- PR #2197: Support `np.nanprod()`.
- PR #2190: Support caching for ufunc.
- PR #2186: Add system reporting tool.

Fixes:

- PR #2214, Issue #2212: Fix memory error with ndenumerate and flat iterators.
- PR #2206, Issue #2163: Fix `zip()` consuming extra elements in early exhaustion.
- PR #2185, Issue #2159, #2169: Fix rewrite pass affecting objmode fallback.
- PR #2204, Issue #2178: Fix annotation for liftedloop.
- PR #2203: Fix Appveyor segfault with Python 3.5.
- PR #2202, Issue #2198: Fix target context not initialized when loading from ufunc cache.
- PR #2172, Issue #2171: Fix optional type unpacking.
- PR #2189, Issue #2188: Disable freezing of big (>1MB) global arrays.
- PR #2180, Issue #2179: Fix invalid variable version in looplifting.
- PR #2156, Issue #2155: Fix divmod, floordiv segfault on CUDA.

10.33 Version 0.29.0

This release extends the support of recursive functions to include direct and indirect recursion without explicit function
type annotations. See new example in *examples/mergesort.py*. Newly supported numpy features include array stacking
functions, np.linalg.eig* functions, np.linalg.matrix_power, np.roots and array to array broadcasting in assignments.

This release depends on llvmlite 0.14.0 and supports CUDA 8 but it is not required.

Improvements:

- PR #2130, #2137: Add type-inferred recursion with docs and examples.
- PR #2134: Add `np.linalg.matrix_power`.
- PR #2125: Add `np.roots`.
- PR #2129: Add `np.linalg.{eigvals,eigh,eigvalsh}`.
- PR #2126: Add array-to-array broadcasting.

- PR #2069: Add hstack and related functions.

- PR #2128: Allow for vectorizing a jitted function. (thanks to @dhirschfeld)

- PR #2117: Update examples and make them test-able.

- PR #2127: Refactor interpreter class and its results.

Fixes:

- PR #2149: Workaround MSVC9.0 SP1 fmod bug kb982107.

- PR #2145, Issue #2009: Fixes kwargs for jitclass `__init__` method.

- PR #2150: Fix slowdown in objmode fallback.

- PR #2050, Issue #1259: Fix liveness problem with some generator loops.

- PR #2072, Issue #1995: Right shift of unsigned LHS should be logical.

- PR #2115, Issue #1466: Fix inspect_types() error due to mangled variable name.

- PR #2119, Issue #2118: Fix array type created from record-dtype.

- PR #2122, Issue #1808: Fix returning a generator due to datamodel error.

10.34 Version 0.28.1

This is a bug-fix release to resolve packaging issues with setuptools dependency.

10.35 Version 0.28.0

Amongst other improvements, this version improves again the level of support for linear algebra – functions from the `numpy.linalg` module. Also, our random generator is now guaranteed to be thread-safe and fork-safe.

Improvements:

- PR #2019: Add the `@intrinsic` decorator to define low-level subroutines callable from JIT functions (this is considered a private API for now).

- PR #2059: Implement `np.concatenate` and `np.stack`.

- PR #2048: Make random generation fork-safe and thread-safe, producing independent streams of random numbers for each thread or process.

- PR #2031: Add documentation of floating-point pitfalls.

- Issue #2053: Avoid polling in parallel CPU target (fixes severe performance regression on Windows).

- Issue #2029: Make default arguments fast.

- PR #2052: Add logging to the CUDA driver.

- PR #2049: Implement the built-in `divmod()` function.

- PR #2036: Implement the `argsort()` method on arrays.

- PR #2046: Improving CUDA memory management by deferring deallocations until certain thresholds are reached, so as to avoid breaking asynchronous execution.

- PR #2040: Switch the CUDA driver implementation to use CUDA's "primary context" API.

- PR #2017: Allow `min(tuple)` and `max(tuple)`.

- PR #2039: Reduce fork() detection overhead in CUDA.

- PR #2021: Handle structured dtypes with titles.

- PR #1996: Rewrite looplifting as a transformation on Numba IR.

- PR #2014: Implement `np.linalg.matrix_rank`.

- PR #2012: Implement `np.linalg.cond`.

- PR #1985: Rewrite even trivial array expressions, which opens the door for other optimizations (for example, `array ** 2` can be converted into `array * array`).

- PR #1950: Have `typeof()` always raise ValueError on failure. Previously, it would either raise or return None, depending on the input.

- PR #1994: Implement `np.linalg.norm`.

- PR #1987: Implement `np.linalg.det` and `np.linalg.slogdet`.

- Issue #1979: Document integer width inference and how to workaround.

- PR #1938: Numba is now compatible with LLVM 3.8.

- PR #1967: Restrict `np.linalg` functions to homogeneous dtypes. Users wanting to pass mixed-typed inputs have to convert explicitly, which makes the performance implications more obvious.

Fixes:

- PR #2006: `array(float32) ** int` should return `array(float32)`.

- PR #2044: Allow reshaping empty arrays.

- Issue #2051: Fix refcounting issue when concatenating tuples.

- Issue #2000: Make Numpy optional for setup.py, to allow `pip install` to work without Numpy pre-installed.

- PR #1989: Fix assertion in `Dispatcher.disable_compile()`.

- Issue #2028: Ignore filesystem errors when caching from multiple processes.

- Issue #2003: Allow unicode variable and function names (on Python 3).

- Issue #1998: Fix deadlock in parallel ufuncs that reacquire the GIL.

- PR #1997: Fix random crashes when AOT compiling on certain Windows platforms.

- Issue #1988: Propagate jitclass docstring.

- Issue #1933: Ensure array constants are emitted with the right alignment.

10.36 Version 0.27.0

Improvements:

- Issue #1976: improve error message when non-integral dimensions are given to a CUDA kernel.

- PR #1970: Optimize the power operator with a static exponent.

- PR #1710: Improve contextual information for compiler errors.

- PR #1961: Support printing constant strings.

- PR #1959: Support more types in the print() function.

- PR #1823: Support `compute_50` in CUDA backend.

- PR #1955: Support `np.linalg.pinv`.

- PR #1896: Improve the `SmartArray` API.

- PR #1947: Support `np.linalg.solve`.

- Issue #1943: Improve error message when an argument fails typing.4

- PR #1927: Support `np.linalg.lstsq`.

- PR #1934: Use system functions for hypot() where possible, instead of our own implementation.

- PR #1929: Add cffi support to `@cfunc` objects.

- PR #1932: Add user-controllable thread pool limits for parallel CPU target.

- PR #1928: Support self-recursion when the signature is explicit.

- PR #1890: List all lowering implementations in the developer docs.

- Issue #1884: Support `np.lib.stride_tricks.as_strided()`.

Fixes:

- Issue #1960: Fix sliced assignment when source and destination areas are overlapping.

- PR #1963: Make CUDA print() atomic.

- PR #1956: Allow 0d array constants.

- Issue #1945: Allow using Numpy ufuncs in AOT compiled code.

- Issue #1916: Fix documentation example for `@generated_jit`.

- Issue #1926: Fix regression when caching functions in an IPython session.

- Issue #1923: Allow non-intp integer arguments to carray() and farray().

- Issue #1908: Accept non-ASCII unicode docstrings on Python 2.

- Issue #1874: Allow `del container[key]` in object mode.

- Issue #1913: Fix set insertion bug when the lookup chain contains deleted entries.

- Issue #1911: Allow function annotations on jitclass methods.

10.37 Version 0.26.0

This release adds support for `cfunc` decorator for exporting numba jitted functions to 3rd party API that takes C callbacks. Most of the overhead of using jitclasses inside the interpreter are eliminated. Support for decompositions in `numpy.linalg` are added. Finally, Numpy 1.11 is supported.

Improvements:

- PR #1889: Export BLAS and LAPACK wrappers for pycc.

- PR #1888: Faster array power.

- Issue #1867: Allow "out" keyword arg for dufuncs.

- PR #1871: `carray()` and `farray()` for creating arrays from pointers.

- PR #1855: `@cfunc` decorator for exporting as ctypes function.

- PR #1862: Add support for `numpy.linalg.qr`.

- PR #1851: jitclass support for '_' and '__' prefixed attributes.

- PR #1842: Optimize jitclass in Python interpreter.

- Issue #1837: Fix CUDA simulator issues with device function.

- PR #1839: Add support for decompositions from `numpy.linalg`.

- PR #1829: Support Python enums.

- PR #1828: Add support for `numpy.random.rand()`` and `numpy.random.randn()`

- Issue #1825: Use of 0-darray in place of scalar index.

- Issue #1824: Scalar arguments to object mode gufuncs.

- Issue #1813: Let bitwise bool operators return booleans, not integers.

- Issue #1760: Optional arguments in generators.

- PR #1780: Numpy 1.11 support.

10.38 Version 0.25.0

This release adds support for `set` objects in nopython mode. It also adds support for many missing Numpy features and functions. It improves Numba's compatibility and performance when using a distributed execution framework such as dask, distributed or Spark. Finally, it removes compatibility with Python 2.6, Python 3.3 and Numpy 1.6.

Improvements:

- Issue #1800: Add erf(), erfc(), gamma() and lgamma() to CUDA targets.

- PR #1793: Implement more Numpy functions: np.bincount(), np.diff(), np.digitize(), np.histogram(), np.searchsorted() as well as NaN-aware reduction functions (np.nansum(), np.nanmedian(), etc.)

- PR #1789: Optimize some reduction functions such as np.sum(), np.prod(), np.median(), etc.

- PR #1752: Make CUDA features work in dask, distributed and Spark.

- PR #1787: Support np.nditer() for fast multi-array indexing with broadcasting.

- PR #1799: Report JIT-compiled functions as regular Python functions when profiling (allowing to see the filename and line number where a function is defined).

- PR #1782: Support np.any() and np.all().

- Issue #1788: Support the iter() and next() built-in functions.

- PR #1778: Support array.astype().

- Issue #1775: Allow the user to set the target CPU model for AOT compilation.

- PR #1758: Support creating random arrays using the `size` parameter to the np.random APIs.

- PR #1757: Support len() on array.flat objects.

- PR #1749: Remove Numpy 1.6 compatibility.

- PR #1748: Remove Python 2.6 and 3.3 compatibility.

- PR #1735: Support the `not in` operator as well as operator.contains().

- PR #1724: Support homogeneous sets in nopython mode.

- Issue #875: make compilation of array constants faster.

Fixes:

- PR #1795: Fix a massive performance issue when calling Numba functions with distributed, Spark or a similar mechanism using serialization.

- Issue #1784: Make jitclasses usable with NUMBA_DISABLE_JIT=1.

- Issue #1786: Allow using linear algebra functions when profiling.

- Issue #1796: Fix np.dot() memory leak on non-contiguous inputs.

- PR #1792: Fix static negative indexing of tuples.

- Issue #1771: Use fallback cache directory when __pycache__ isn't writable, such as when user code is installed in a system location.

- Issue #1223: Use Numpy error model in array expressions (e.g. division by zero returns `inf` or `nan` instead of raising an error).

- Issue #1640: Fix np.random.binomial() for large n values.

- Issue #1643: Improve error reporting when passing an invalid spec to `@jitclass`.

- PR #1756: Fix slicing with a negative step and an omitted start.

10.39 Version 0.24.0

This release introduces several major changes, including the `@generated_jit` decorator for flexible specializations as with Julia's "`@generated`" macro, or the SmartArray array wrapper type that allows seamless transfer of array data between the CPU and the GPU.

This will be the last version to support Python 2.6, Python 3.3 and Numpy 1.6.

Improvements:

- PR #1723: Improve compatibility of JIT functions with the Python profiler.

- PR #1509: Support array.ravel() and array.flatten().

- PR #1676: Add SmartArray type to support transparent data management in multiple address spaces (host & GPU).

- PR #1689: Reduce startup overhead of importing Numba.

- PR #1705: Support registration of CFFI types as corresponding to known Numba types.

- PR #1686: Document the extension API.

- PR #1698: Improve warnings raised during type inference.

- PR #1697: Support np.dot() and friends on non-contiguous arrays.

- PR #1692: cffi.from_buffer() improvements (allow more pointer types, allow non-Numpy buffer objects).

- PR #1648: Add the `@generated_jit` decorator.

- PR #1651: Implementation of np.linalg.inv using LAPACK. Thanks to Matthieu Dartiailh.

- PR #1674: Support np.diag().

- PR #1673: Improve error message when looking up an attribute on an unknown global.

- Issue #1569: Implement runtime check for the LLVM locale bug.

- PR #1612: Switch to LLVM 3.7 in sync with llvmlite.

- PR #1624: Allow slice assignment of sequence to array.

- PR #1622: Support slicing tuples with a constant slice.

Fixes:

- Issue #1722: Fix returning an optional boolean (bool or None).

- Issue #1734: NRT decref bug when variable is del'ed before being defined, leading to a possible memory leak.

- PR #1732: Fix tuple getitem regression for CUDA target.

- PR #1718: Mishandling of optional to optional casting.

- PR #1714: Fix .compile() on a JIT function not respecting ._can_compile.

- Issue #1667: Fix np.angle() on arrays.

- Issue #1690: Fix slicing with an omitted stop and a negative step value.

- PR #1693: Fix gufunc bug in handling scalar formal arg with non-scalar input value.

- PR #1683: Fix parallel testing under Windows.

- Issue #1616: Use system-provided versions of C99 math where possible.

- Issue #1652: Reductions of bool arrays (e.g. sum() or mean()) should return integers or floats, not bools.

- Issue #1664: Fix regression when indexing a record array with a constant index.

- PR #1661: Disable AVX on old Linux kernels.

- Issue #1636: Allow raising an exception looked up on a module.

10.40 Version 0.23.1

This is a bug-fix release to address several regressions introduced in the 0.23.0 release, and a couple other issues.

Fixes:

- Issue #1645: CUDA ufuncs were broken in 0.23.0.

- Issue #1638: Check tuple sizes when passing a list of tuples.

- Issue #1630: Parallel ufunc would keep eating CPU even after finishing under Windows.

- Issue #1628: Fix ctypes and cffi tests under Windows with Python 3.5.

- Issue #1627: Fix xrange() support.

- PR #1611: Rewrite variable liveness analysis.

- Issue #1610: Allow nested calls between explicitly-typed ufuncs.

- Issue #1593: Fix *args in object mode.

10.41 Version 0.23.0

This release introduces JIT classes using the new `@jitclass` decorator, allowing user-defined structures for nopython mode. Other improvements and bug fixes are listed below.

Improvements:

- PR #1609: Speed up some simple math functions by inlining them in their caller

- PR #1571: Implement JIT classes

- PR #1584: Improve typing of array indexing
- PR #1583: Allow printing booleans
- PR #1542: Allow negative values in np.reshape()
- PR #1560: Support vector and matrix dot product, including `np.dot()` and the `@` operator in Python 3.5
- PR #1546: Support field lookup on record arrays and scalars (i.e. `array['field']` in addition to `array.field`)
- PR #1440: Support the HSA wavebarrier() and activelanepermute_wavewidth() intrinsics
- PR #1540: Support np.angle()
- PR #1543: Implement CPU multithreaded gufuncs (target="parallel")
- PR #1551: Allow scalar arguments in np.where(), np.empty_like().
- PR #1516: Add some more examples from NumbaPro
- PR #1517: Support np.sinc()

Fixes:

- Issue #1603: Fix calling a non-cached function from a cached function
- Issue #1594: Ensure a list is homogeneous when unboxing
- Issue #1595: Replace deprecated use of get_pointer_to_function()
- Issue #1586: Allow tests to be run by different users on the same machine
- Issue #1587: Make CudaAPIError picklable
- Issue #1568: Fix using Numba from inside Visual Studio 2015
- Issue #1559: Fix serializing a jit function referring a renamed module
- PR #1508: Let reshape() accept integer argument(s), not just a tuple
- Issue #1545: Improve error checking when unboxing list objects
- Issue #1538: Fix array broadcasting in CUDA gufuncs
- Issue #1526: Fix a reference count handling bug

10.42 Version 0.22.1

This is a bug-fix release to resolve some packaging issues and other problems found in the 0.22.0 release.

Fixes:

- PR #1515: Include MANIFEST.in in MANIFEST.in so that sdist still works from source tar files.
- PR #1518: Fix reference counting bug caused by hidden alias
- PR #1519: Fix erroneous assert when passing nopython=True to guvectorize.
- PR #1521: Fix cuda.test()

10.43 Version 0.22.0

This release features several highlights: Python 3.5 support, Numpy 1.10 support, Ahead-of-Time compilation of extension modules, additional vectorization features that were previously only available with the proprietary extension NumbaPro, improvements in array indexing.

Improvements:

- PR #1497: Allow scalar input type instead of size-1 array to @guvectorize

- PR #1480: Add distutils support for AOT compilation

- PR #1460: Create a new API for Ahead-of-Time (AOT) compilation

- PR #1451: Allow passing Python lists to JIT-compiled functions, and reflect mutations on function return

- PR #1387: Numpy 1.10 support

- PR #1464: Support cffi.FFI.from_buffer()

- PR #1437: Propagate errors raised from Numba-compiled ufuncs; also, let "division by zero" and other math errors produce a warning instead of exiting the function early

- PR #1445: Support a subset of fancy indexing

- PR #1454: Support "out-of-line" CFFI modules

- PR #1442: Improve array indexing to support more kinds of basic slicing

- PR #1409: Support explicit CUDA memory fences

- PR #1435: Add support for vectorize() and guvectorize() with HSA

- PR #1432: Implement numpy.nonzero() and numpy.where()

- PR #1416: Add support for vectorize() and guvectorize() with CUDA, as originally provided in NumbaPro

- PR #1424: Support in-place array operators

- PR #1414: Python 3.5 support

- PR #1404: Add the parallel ufunc functionality originally provided in NumbaPro

- PR #1393: Implement sorting on arrays and lists

- PR #1415: Add functions to estimate the occupancy of a CUDA kernel

- PR #1360: The JIT cache now stores the compiled object code, yielding even larger speedups.

- PR #1402: Fixes for the ARMv7 (armv7l) architecture under Linux

- PR #1400: Add the cuda.reduce() decorator originally provided in NumbaPro

Fixes:

- PR #1483: Allow np.empty_like() and friends on non-contiguous arrays

- Issue #1471: Allow caching JIT functions defined in IPython

- PR #1457: Fix flat indexing of boolean arrays

- PR #1421: Allow calling Numpy ufuncs, without an explicit output, on non-contiguous arrays

- Issue #1411: Fix crash when unpacking a tuple containing a Numba-allocated array

- Issue #1394: Allow unifying range_state32 and range_state64

- Issue #1373: Fix code generation error on lists of bools

10.44 Version 0.21.0

This release introduces support for AMD's Heterogeneous System Architecture, which allows memory to be shared directly between the CPU and the GPU. Other major enhancements are support for lists and the introduction of an opt-in compilation cache.

Improvements:

- PR #1391: Implement print() for CUDA code
- PR #1366: Implement integer typing enhancement proposal (NBEP 1)
- PR #1380: Support the one-argument type() builtin
- PR #1375: Allow boolean evaluation of lists and tuples
- PR #1371: Support array.view() in CUDA mode
- PR #1369: Support named tuples in nopython mode
- PR #1250: Implement numpy.median().
- PR #1289: Make dispatching faster when calling a JIT-compiled function from regular Python
- Issue #1226: Improve performance of integer power
- PR #1321: Document features supported with CUDA
- PR #1345: HSA support
- PR #1343: Support lists in nopython mode
- PR #1356: Make Numba-allocated memory visible to tracemalloc
- PR #1363: Add an environment variable NUMBA_DEBUG_TYPEINFER
- PR #1051: Add an opt-in, per-function compilation cache

Fixes:

- Issue #1372: Some array expressions would fail rewriting when involved the same variable more than once, or a unary operator
- Issue #1385: Allow CUDA local arrays to be declared anywhere in a function
- Issue #1285: Support datetime64 and timedelta64 in Numpy reduction functions
- Issue #1332: Handle the EXTENDED_ARG opcode.
- PR #1329: Handle the `in` operator in object mode
- Issue #1322: Fix augmented slice assignment on Python 2
- PR #1357: Fix slicing with some negative bounds or step values.

10.45 Version 0.20.0

This release updates Numba to use LLVM 3.6 and CUDA 7 for CUDA support. Following the platform deprecation in CUDA 7, Numba's CUDA feature is no longer supported on 32-bit platforms. The oldest supported version of Windows is Windows 7.

Improvements:

- Issue #1203: Support indexing ndarray.flat

- PR #1200: Migrate cgutils to llvmlite
- PR #1190: Support more array methods: .transpose(), .T, .copy(), .reshape(), .view()
- PR #1214: Simplify setup.py and avoid manual maintenance
- PR #1217: Support datetime64 and timedelta64 constants
- PR #1236: Reload environment variables when compiling
- PR #1225: Various speed improvements in generated code
- PR #1252: Support cmath module in CUDA
- PR #1238: Use 32-byte aligned allocator to optimize for AVX
- PR #1258: Support numpy.frombuffer()
- PR #1274: Use TravisCI container infrastructure for lower wait time
- PR #1279: Micro-optimize overload resolution in call dispatch
- Issue #1248: Improve error message when return type unification fails

Fixes:

- Issue #1131: Handling of negative zeros in np.conjugate() and np.arccos()
- Issue #1188: Fix slow array return
- Issue #1164: Avoid warnings from CUDA context at shutdown
- Issue #1229: Respect the writeable flag in arrays
- Issue #1244: Fix bug in refcount pruning pass
- Issue #1251: Fix partial left-indexing of Fortran contiguous array
- Issue #1264: Fix compilation error in array expression
- Issue #1254: Fix error when yielding array objects
- Issue #1276: Fix nested generator use

10.46 Version 0.19.2

This release fixes the source distribution on pypi. The only change is in the setup.py file. We do not plan to provide a conda package as this release is essentially the same as 0.19.1 for conda users.

10.47 Version 0.19.1

- Issue #1196:
  - fix double-free segfault due to redundant variable deletion in the Numba IR (#1195)
  - fix use-after-delete in array expression rewrite pass

10.48 Version 0.19.0

This version introduces memory management in the Numba runtime, allowing to allocate new arrays inside Numba-compiled functions. There is also a rework of the ufunc infrastructure, and an optimization pass to collapse cascading array operations into a single efficient loop.

> **Warning:** Support for Windows XP and Vista with all compiler targets and support for 32-bit platforms (Win/Mac/Linux) with the CUDA compiler target are deprecated. In the next release of Numba, the oldest version of Windows supported will be Windows 7. CPU compilation will remain supported on 32-bit Linux and Windows platforms.

Known issues:

- There are some performance regressions in very short running `nopython` functions due to the additional overhead incurred by memory management. We will work to reduce this overhead in future releases.

Features:

- Issue #1181: Add a Frequently Asked Questions section to the documentation.

- Issue #1162: Support the `cumsum()` and `cumprod()` methods on Numpy arrays.

- Issue #1152: Support the `*args` argument-passing style.

- Issue #1147: Allow passing character sequences as arguments to JIT-compiled functions.

- Issue #1110: Shortcut deforestation and loop fusion for array expressions.

- Issue #1136: Support various Numpy array constructors, for example numpy.zeros() and numpy.zeros_like().

- Issue #1127: Add a CUDA simulator running on the CPU, enabled with the NUMBA_ENABLE_CUDASIM environment variable.

- Issue #1086: Allow calling standard Numpy ufuncs without an explicit output array from `nopython` functions.

- Issue #1113: Support keyword arguments when calling numpy.empty() and related functions.

- Issue #1108: Support the `ctypes.data` attribute of Numpy arrays.

- Issue #1077: Memory management for array allocations in `nopython` mode.

- Issue #1105: Support calling a ctypes function that takes ctypes.py_object parameters.

- Issue #1084: Environment variable NUMBA_DISABLE_JIT disables compilation of `@jit` functions, instead calling into the Python interpreter when called. This allows easier debugging of multiple jitted functions.

- Issue #927: Allow gufuncs with no output array.

- Issue #1097: Support comparisons between tuples.

- Issue #1075: Numba-generated ufuncs can now be called from `nopython` functions.

- Issue #1062: `@vectorize` now allows omitting the signatures, and will compile the required specializations on the fly (like `@jit` does).

- Issue #1027: Support numpy.round().

- Issue #1085: Allow returning a character sequence (as fetched from a structured array) from a JIT-compiled function.

Fixes:

- Issue #1170: Ensure `ndindex()`, `ndenumerate()` and `ndarray.flat` work properly inside generators.

- Issue #1151: Disallow unpacking of tuples with the wrong size.

- Issue #1141: Specify install dependencies in setup.py.

- Issue #1106: Loop-lifting would fail when the lifted loop does not produce any output values for the function tail.

- Issue #1103: Fix mishandling of some inputs when a JIT-compiled function is called with multiple array layouts.

- Issue #1089: Fix range() with large unsigned integers.

- Issue #1088: Install entry-point scripts (numba, pycc) from the conda build recipe.

- Issue #1081: Constant structured scalars now work properly.

- Issue #1080: Fix automatic promotion of booleans to integers.

10.49 Version 0.18.2

Bug fixes:

- Issue #1073: Fixes missing template file for HTML annotation

- Issue #1074: Fixes CUDA support on Windows machine due to NVVM API mismatch

10.50 Version 0.18.1

Version 0.18.0 is not officially released.

This version removes the old deprecated and undocumented `argtypes` and `restype` arguments to the `@jit` decorator. Function signatures should always be passed as the first argument to `@jit`.

Features:

- Issue #960: Add inspect_llvm() and inspect_asm() methods to JIT-compiled functions: they output the LLVM IR and the native assembler source of the compiled function, respectively.

- Issue #990: Allow passing tuples as arguments to JIT-compiled functions in `nopython` mode.

- Issue #774: Support two-argument round() in `nopython` mode.

- Issue #987: Support missing functions from the math module in nopython mode: frexp(), ldexp(), gamma(), lgamma(), erf(), erfc().

- Issue #995: Improve code generation for round() on Python 3.

- Issue #981: Support functions from the random and numpy.random modules in `nopython` mode.

- Issue #979: Add cuda.atomic.max().

- Issue #1006: Improve exception raising and reporting. It is now allowed to raise an exception with an error message in `nopython` mode.

- Issue #821: Allow ctypes- and cffi-defined functions as arguments to `nopython` functions.

- Issue #901: Allow multiple explicit signatures with `@jit`. The signatures must be passed in a list, as with `@vectorize`.

- Issue #884: Better error message when a JIT-compiled function is called with the wrong types.

- Issue #1010: Simpler and faster CUDA argument marshalling thanks to a refactoring of the data model.

- Issue #1018: Support arrays of scalars inside Numpy structured types.

- Issue #808: Reduce Numba import time by half.

- Issue #1021: Support the buffer protocol in `nopython` mode. Buffer-providing objects, such as `bytearray`, `array.array` or `memoryview` support array-like operations such as indexing and iterating. Furthermore, some standard attributes on the `memoryview` object are supported.

- Issue #1030: Support nested arrays in Numpy structured arrays.

- Issue #1033: Implement the inspect_types(), inspect_llvm() and inspect_asm() methods for CUDA kernels.

- Issue #1029: Support Numpy structured arrays with CUDA as well.

- Issue #1034: Support for generators in nopython and object mode.

- Issue #1044: Support default argument values when calling Numba-compiled functions.

- Issue #1048: Allow calling Numpy scalar constructors from CUDA functions.

- Issue #1047: Allow indexing a multi-dimensional array with a single integer, to take a view.

- Issue #1050: Support len() on tuples.

- Issue #1011: Revive HTML annotation.

Fixes:

- Issue #977: Assignment optimization was too aggressive.

- Issue #561: One-argument round() now returns an int on Python 3.

- Issue #1001: Fix an unlikely bug where two closures with the same name and id() would compile to the same LLVM function name, despite different closure values.

- Issue #1006: Fix reference leak when a JIT-compiled function is disposed of.

- Issue #1017: Update instructions for CUDA in the README.

- Issue #1008: Generate shorter LLVM type names to avoid segfaults with CUDA.

- Issue #1005: Properly clean up references when raising an exception from object mode.

- Issue #1041: Fix incompatibility between Numba and the third-party library "future".

- Issue #1053: Fix the size attribute of CUDA shared arrays.

10.51 Version 0.17.0

The major focus in this release has been a rewrite of the documentation. The new documentation is better structured and has more detailed coverage of Numba features and APIs. It can be found online at http://numba.pydata.org/numba-doc/dev/index.html

Features:

- Issue #895: LLVM can now inline nested function calls in `nopython` mode.

- Issue #863: CUDA kernels can now infer the types of their arguments ("autojit"-like).

- Issue #833: Support numpy.{min,max,argmin,argmax,sum,mean,var,std} in `nopython` mode.

- Issue #905: Add a `nogil` argument to the `@jit` decorator, to release the GIL in `nopython` mode.

- Issue #829: Add a `identity` argument to `@vectorize` and `@guvectorize`, to set the identity value of the ufunc.

- Issue #843: Allow indexing 0-d arrays with the empty tuple.

- Issue #933: Allow named arguments, not only positional arguments, when calling a Numba-compiled function.

- Issue #902: Support numpy.ndenumerate() in `nopython` mode.

- Issue #950: AVX is now enabled by default except on Sandy Bridge and Ivy Bridge CPUs, where it can produce slower code than SSE.

- Issue #956: Support constant arrays of structured type.

- Issue #959: Indexing arrays with floating-point numbers isn't allowed anymore.

- Issue #955: Add support for 3D CUDA grids and thread blocks.

- Issue #902: Support numpy.ndindex() in `nopython` mode.

- Issue #951: Numpy number types (`numpy.int8`, etc.) can be used as constructors for type conversion in `nopython` mode.

Fixes:

- Issue #889: Fix `NUMBA_DUMP_ASSEMBLY` for the CUDA backend.

- Issue #903: Fix calling of stdcall functions with ctypes under Windows.

- Issue #908: Allow lazy-compiling from several threads at once.

- Issue #868: Wrong error message when multiplying a scalar by a non-scalar.

- Issue #917: Allow vectorizing with datetime64 and timedelta64 in the signature (only with unit-less values, though, because of a Numpy limitation).

- Issue #431: Allow overloading of cuda device function.

- Issue #917: Print out errors occurred in object mode ufuncs.

- Issue #923: Numba-compiled ufuncs now inherit the name and doc of the original Python function.

- Issue #928: Fix boolean return value in nested calls.

- Issue #915: `@jit` called with an explicit signature with a mismatching type of arguments now raises an error.

- Issue #784: Fix the truth value of NaNs.

- Issue #953: Fix using shared memory in more than one function (kernel or device).

- Issue #970: Fix an uncommon double to uint64 conversion bug on CentOS5 32-bit (C compiler issue).

10.52 Version 0.16.0

This release contains a major refactor to switch from llvmpy to llvmlite as our code generation backend. The switch is necessary to reconcile different compiler requirements for LLVM 3.5 (needs C++11) and Python extensions (need specific compiler versions on Windows). As a bonus, we have found the use of llvmlite speeds up compilation by a factor of 2!

Other Major Changes:

- Faster dispatch for numpy structured arrays

- Optimized array.flat()

- Improved CPU feature selection

- Fix constant tuple regression in macro expansion code

Known Issues:

- AVX code generation is still disabled by default due to performance regressions when operating on misaligned NumPy arrays. We hope to have a workaround in the future.

- In *extremely* rare circumstances, a known issue with LLVM 3.5 code generation can cause an ELF relocation error on 64-bit Linux systems.

10.53 Version 0.15.1

(This was a bug-fix release that superceded version 0.15 before it was announced.)

Fixes:

- Workaround for missing __ftol2 on Windows XP.

- Do not lift loops for compilation that contain break statements.

- Fix a bug in loop-lifting when multiple values need to be returned to the enclosing scope.

- Handle the loop-lifting case where an accumulator needs to be updated when the loop count is zero.

10.54 Version 0.15

Features:

- Support for the Python `cmath` module. (NumPy complex functions were already supported.)

- Support for `.real`, `.imag`, and *.conjugate()`* on non-complex numbers.

- Add support for `math.isfinite()` and `math.copysign()`.

- Compatibility mode: If enabled (off by default), a failure to compile in object mode will fall back to using the pure Python implementation of the function.

- *Experimental* support for serializing JIT functions with cloudpickle.

- Loop-jitting in object mode now works with loops that modify scalars that are accessed after the loop, such as accumulators.

- `@vectorize` functions can be compiled in object mode.

- Numba can now be built using the Visual C++ Compiler for Python 2.7 on Windows platforms.

- CUDA JIT functions can be returned by factory functions with variables in the closure frozen as constants.

- Support for "optional" types in nopython mode, which allow `None` to be a valid value.

Fixes:

- If nopython mode compilation fails for any reason, automatically fall back to object mode (unless nopython=True is passed to @jit) rather than raise an exeception.

- Allow function objects to be returned from a function compiled in object mode.

- Fix a linking problem that caused slower platform math functions (such as `exp()`) to be used on Windows, leading to performance regressions against NumPy.

- `min()` and `max()` no longer accept scalars arguments in nopython mode.

- Fix handling of ambigous type promotion among several compiled versions of a JIT function. The dispatcher will now compile a new version to resolve the problem. (issue #776)

- Fix float32 to uint64 casting bug on 32-bit Linux.

- Fix type inference to allow forced casting of return types.

- Allow the shape of a 1D `cuda.shared.array` and `cuda.local.array` to be a one-element tuple.

- More correct handling of signed zeros.

- Add custom implementation of `atan2()` on Windows to handle special cases properly.

- Eliminated race condition in the handling of the pagelocked staging area used when transferring CUDA arrays.

- Fix non-deterministic type unification leading to varying performance. (issue #797)

10.55 Version 0.14

Features:

- Support for nearly all the Numpy math functions (including comparison, logical, bitwise and some previously missing float functions) in nopython mode.

- The Numpy datetime64 and timedelta64 dtypes are supported in nopython mode with Numpy 1.7 and later.

- Support for Numpy math functions on complex numbers in nopython mode.

- ndarray.sum() is supported in nopython mode.

- Better error messages when unsupported types are used in Numpy math functions.

- Set NUMBA_WARNINGS=1 in the environment to see which functions are compiled in object mode vs. nopython mode.

- Add support for the two-argument pow() builtin function in nopython mode.

- New developer documentation describing how Numba works, and how to add new types.

- Support for Numpy record arrays on the GPU. (Note: Improper alignment of dtype fields will cause an exception to be raised.)

- Slices on GPU device arrays.

- GPU objects can be used as Python context managers to select the active device in a block.

- GPU device arrays can be bound to a CUDA stream. All subsequent operations (such as memory copies) will be queued on that stream instead of the default. This can prevent unnecessary synchronization with other streams.

Fixes:

- Generation of AVX instructions has been disabled to avoid performance bugs when calling external math functions that may use SSE instructions, especially on OS X.

- JIT functions can be removed by the garbage collector when they are no longer accessible.

- Various other reference counting fixes to prevent memory leaks.

- Fixed handling of exception when input argument is out of range.

- Prevent autojit functions from making unsafe numeric conversions when called with different numeric types.

- Fix a compilation error when an unhashable global value is accessed.

- Gracefully handle failure to enable faulthandler in the IPython Notebook.

- Fix a bug that caused loop lifting to fail if the loop was inside an `else` block.

- Fixed a problem with selecting CUDA devices in multithreaded programs on Linux.

- The `pow()` function (and `**` operation) applied to two integers now returns an integer rather than a float.

- Numpy arrays using the object dtype no longer cause an exception in the autojit.

- Attempts to write to a global array will cause compilation to fall back to object mode, rather than attempt and fail at nopython mode.

- `range()` works with all negative arguments (ex: `range(-10, -12, -1)`)

10.56 Version 0.13.4

Features:

- Setting and deleting attributes in object mode

- Added documentation of supported and currently unsupported numpy ufuncs

- Assignment to 1-D numpy array slices

- Closure variables and functions can be used in object mode

- All numeric global values in modules can be used as constants in JIT compiled code

- Support for the start argument in enumerate()

- Inplace arithmetic operations (+=, -=, etc.)

- Direct iteration over a 1D numpy array (e.g. "for x in array: . . . ") in nopython mode

Fixes:

- Support for NVIDIA compute capability 5.0 devices (such as the GTX 750)

- Vectorize no longer crashes/gives an error when bool_ is used as return type

- Return the correct dictionary when globals() is used in JIT functions

- Fix crash bug when creating dictionary literals in object

- Report more informative error message on import if llvmpy is too old

- Temporarily disable pycc –header, which generates incorrect function signatures.

10.57 Version 0.13.3

Features:

- Support for enumerate() and zip() in nopython mode

- Increased LLVM optimization of JIT functions to -O1, enabling automatic vectorization of compiled code in some cases

- Iteration over tuples and unpacking of tuples in nopython mode

- Support for dict and set (Python >= 2.7) literals in object mode

Fixes:

- JIT functions have the same __name__ and __doc__ as the original function.

- Numerous improvements to better match the data types and behavior of Python math functions in JIT compiled code on different platforms.

- Importing Numba will no longer throw an exception if the CUDA driver is present, but cannot be initialized.

- guvectorize now properly supports functions with scalar arguments.

- CUDA driver is lazily initialized

10.58 Version 0.13.2

Features:

- @vectorize ufunc now can generate SIMD fast path for unit strided array
- Added cuda.gridsize
- Added preliminary exception handling (raise exception class)

Fixes:

- UNARY_POSITIVE
- Handling of closures and dynamically generated functions
- Global None value

10.59 Version 0.13.1

Features:

- Initial support for CUDA array slicing

Fixes:

- Indirectly fixes numbapro when the system has a incompatible CUDA driver
- Fix numba.cuda.detect
- Export numba.intp and numba.intc

10.60 Version 0.13

Features:

- Opensourcing NumbaPro CUDA python support in *numba.cuda*
- Add support for ufunc array broadcasting
- Add support for mixed input types for ufuncs
- Add support for returning tuple from jitted function

Fixes:

- Fix store slice bytecode handling for Python2
- Fix inplace subtract
- Fix pycc so that correct header is emitted
- Allow vectorize to work on functions with jit decorator

10.61 Version 0.12.2

Fixes:

- Improved NumPy ufunc support in nopython mode

- Misc bug fixes

10.62 Version 0.12.1

This version fixed many regressions reported by user for the 0.12 release. This release contains a new loop-lifting mechanism that specializes certains loop patterns for nopython mode compilation. This avoid direct support for heap-allocating and other very dynamic operations.

Improvements:

- Add loop-lifting–jit-ing loops in nopython for object mode code. This allows functions to allocate NumPy arrays and use Python objects, while the tight loops in the function can still be compiled in nopython mode. Any arrays that the tight loop uses should be created before the loop is entered.

Fixes:

- Add support for majority of "math" module functions

- Fix for...else handling

- Add support for builtin round()

- Fix tenary if...else support

- Revive "numba" script

- Fix problems with some boolean expressions

- Add support for more NumPy ufuncs

10.63 Version 0.12

Version 0.12 contains a big refactor of the compiler. The main objective for this refactor was to simplify the code base to create a better foundation for further work. A secondary objective was to improve the worst case performance to ensure that compiled functions in object mode never run slower than pure Python code (this was a problem in several cases with the old code base). This refactor is still a work in progress and further testing is needed.

Main improvements:

- Major refactor of compiler for performance and maintenance reasons

- Better fallback to object mode when native mode fails

- Improved worst case performance in object mode

The public interface of numba has been slightly changed. The idea is to make it cleaner and more rational:

- jit decorator has been modified, so that it can be called without a signature. When called without a signature, it behaves as the old autojit. Autojit has been deprecated in favour of this approach.

- Jitted functions can now be overloaded.

- Added a "njit" decorator that behaves like "jit" decorator with nopython=True.

- The numba.vectorize namespace is gone. The vectorize decorator will be in the main numba namespace.

- Added a guvectorize decorator in the main numba namespace. It is similar to numba.vectorize, but takes a dimension signature. It generates gufuncs. This is a replacement for the GUVectorize gufunc factory which has been deprecated.

Main regressions (will be fixed in a future release):

- Creating new NumPy arrays is not supported in nopython mode

- Returning NumPy arrays is not supported in nopython mode

- NumPy array slicing is not supported in nopython mode

- lists and tuples are not supported in nopython mode

- string, datetime, cdecimal, and struct types are not implemented yet

- Extension types (classes) are not supported in nopython mode

- Closures are not supported

- Raise keyword is not supported

- Recursion is not support in nopython mode

10.64 Version 0.11

- Experimental support for NumPy datetime type

10.65 Version 0.10

- Annotation tool (./bin/numba –annotate –fancy) (thanks to Jay Bourque)

- Open sourced prange

- Support for raise statement

- Pluggable array representation

- Support for enumerate and zip (thanks to Eugene Toder)

- Better string formatting support (thanks to Eugene Toder)

- Builtins min(), max() and bool() (thanks to Eugene Toder)

- Fix some code reloading issues (thanks to Björn Linse)

- Recognize NumPy scalar objects (thanks to Björn Linse)

10.66 Version 0.9

- Improved math support

- Open sourced generalized ufuncs

- Improved array expressions

10.67 Version 0.8

- **Support for autojit classes**

    - Inheritance not yet supported

- Python 3 support for pycc

- **Allow retrieval of ctypes function wrapper**

    - And hence support retrieval of a pointer to the function

- Fixed a memory leak of array slicing views

10.68 Version 0.7.2

- Official Python 3 support (python 3.2 and 3.3)

- Support for intrinsics and instructions

- Various bug fixes (see https://github.com/numba/numba/issues?milestone=7&state=closed)

10.69 Version 0.7.1

- Various bug fixes

10.70 Version 0.7

- Open sourced single-threaded ufunc vectorizer

- Open sourced NumPy array expression compilation

- Open sourced fast NumPy array slicing

- Experimental Python 3 support

- **Support for typed containers**

    - typed lists and tuples

- Support for iteration over objects

- Support object comparisons

- **Preliminary CFFI support**

    - Jit calls to CFFI functions (passed into autojit functions)

    - TODO: Recognize ffi_lib.my_func attributes

- Improved support for ctypes

- Allow declaring extension attribute types as through class attributes

- **Support for type casting in Python**

    - Get the same semantics with or without numba compilation

- **Support for recursion**

- **–** For jit methods and extension classes

- Allow jit functions as C callbacks

- Friendlier error reporting

- Internal improvements

- A variety of bug fixes

10.71 Version 0.6.1

- Support for bitwise operations

10.72 Version 0.6

- Python 2.6 support

- **Programmable typing**

    - **–** Allow users to add type inference for external code

- **Better NumPy type inference**

    - **–** outer, inner, dot, vdot, tensordot, nonzero, where, binary ufuncs + methods (reduce, accumulate, reduceat, outer)

- **Type based alias analysis**

    - **–** Support for strict aliasing

- Much faster autojit dispatch when calling from Python

- Faster numerical loops through data and stride pre-loading

- Integral overflow and underflow checking for conversions from objects

- Make Meta dependency optional

10.73 Version 0.5

- **SSA-based type inference**

    - **–** Allows variable reuse

    - **–** Allow referring to variables before lexical definition

- Support multiple comparisons

- Support for template types

- List comprehensions

- Support for pointers

- Many bug fixes

- Added user documentation

10.74 Version 0.4

10.75 Version 0.3.2

- Add support for object arithmetic (issue 56).
- Bug fixes (issue 55).

10.76 Version 0.3

- Changed default compilation approach to ast
- Added support for cross-module linking
- Added support for closures (can jit inner functions and return them) (see examples/closure.py)
- Added support for dtype structures (can access elements of structure with attribute access) (see examples/structures.py)
- Added support for extension types (numba classes) (see examples/numbaclasses.py)
- Added support for general Python code (use nopython to raise an error if Python C-API is used to avoid unexpected slowness because of lack of implementation defaulting to generic Python)
- Fixed many bugs
- Added support to detect math operations.
- Added with python and with nopython contexts
- Added more examples

Many features need to be documented still. Look at examples and tests for more information.

10.77 Version 0.2

- Added an ast approach to compilation
- Removed d, f, i, b from numba namespace (use f8, f4, i4, b1)
- Changed function to autojit2
- Added autojit function to decorate calls to the function and use types of the variable to create compiled versions.
- changed keyword arguments to jit and autojit functions to restype and argtypes to be consistent with ctypes module.
- Added pycc – a python to shared library compiler

Python Module Index

n

Index

Symbols

A

B

C

O

object mode, **502**

objmode() (*in module numba*), 42

open() (*numba.cuda.cudadrv.devicearray.IpcArrayHandle method*), 173

open_ipc_array() (*numba.cuda method*), 173

OptionalType, **502**

outer() (*numba.DUFunc method*), 94

output_dir (*numba.pycc.CC attribute*), 95

output_file (*numba.pycc.CC attribute*), 95

P

parallel_diagnostics() (*Dispatcher method*), 91

per_thread_default_stream() (*in module numba.cuda*), 189

pinned() (*in module numba.cuda*), 197

pinned_array() (*in module numba.cuda*), 196

PinnedMemory (*class in numba.cuda*), 182

pipeline (*Rewrite attribute*), 277

pop() (*numba.cuda.cudadrv.driver.Context method*), 185

profile_start() (*in module numba.cuda*), 188

profile_stop() (*in module numba.cuda*), 188

profiling() (*in module numba.cuda*), 188

push() (*numba.cuda.cudadrv.driver.Context method*), 186

py_func (*Dispatcher attribute*), 90

Python bytecode, **501**

Python Enhancement Proposals
PEP 465, 131
PEP 7, 238
PEP 8, 238

Q

query() (*numba.cuda.cudadrv.driver.Event method*), 188

R

ravel() (*numba.cuda.cudadrv.devicearray.DeviceNDArray method*), 197

recompile() (*Dispatcher method*), 91

record() (*numba.cuda.cudadrv.driver.Event method*), 188

Reduce (*class in numba.cuda*), 171

reduce() (*numba.DUFunc method*), 93

reduceat() (*numba.DUFunc method*), 94

reflection, **502**

register() (*in module numba.experimental.structref*), 222

require_context() (*in module numba.cuda*), 186

reset() (*numba.cuda.BaseCUDAMemoryManager method*), 179

reset() (*numba.cuda.cudadrv.driver.Context method*), 186

reset() (*numba.cuda.cudadrv.driver.Device method*), 187

reset() (*numba.cuda.HostOnlyCUDAMemoryManager method*), 180

reshape() (*numba.cuda.cudadrv.devicearray.DeviceNDArray method*), 197

Rewrite (*built-in class*), 277

roc.deregister() (*built-in function*), 206

roc.register() (*built-in function*), 206

Runtime (*class in numba.cuda.cudadrv.runtime*), 190

S

select_device() (*in module numba.cuda*), 186

sentry_literal_args() (*in module numba.extending*), 462

SentryLiteralArgs (*class in numba.extending*), 461

set_memory_manager() (*in module numba.cuda*), 183

set_num_threads() (*in module numba*), 51

signature(), 86

specialize() (*numba.cuda.compiler.Dispatcher method*), 192

specialized (*numba.cuda.compiler.Dispatcher attribute*), 192

split() (*numba.cuda.cudadrv.devicearray.DeviceNDArray method*), 197

split() (*numba.cuda.cudadrv.devicearray.MappedNDArray method*), 199

Stream (*class in numba.cuda.cudadrv.driver*), 189

stream() (*in module numba.cuda*), 189

StructRefProxy (*class in numba.experimental.structref*), 221

synchronize() (*in module numba.cuda*), 186

synchronize() (*numba.cuda.cudadrv.driver.Event method*), 188

synchronize() (*numba.cuda.cudadrv.driver.Stream method*), 189

T

target_cpu (*numba.pycc.CC attribute*), 95

threadIdx (*numba.cuda attribute*), 192

to_device() (*in module numba.cuda*), 196

total (*numba.cuda.MemoryInfo attribute*), 182

type inference, **502**

type_callable() (*built-in function*), 222

typeof_impl.register() (*built-in function*), 222

types (*numba.DUFunc attribute*), 93

typing, **502**

U

ufunc, **502**

ufunc (*numba.DUFunc attribute*), 93
unbox() (*built-in function*), 224
unliteral() (*in module numba.types*), 459

V

verbose (*numba.pycc.CC attribute*), 95

W

wait() (*numba.cuda.cudadrv.driver.Event   method*),
        188
warpsize (*numba.cuda attribute*), 192