

# RAPIDS

## Numba for CUDA Programmers

Graham Markall - [gmarkall@nvidia.com](mailto:gmarkall@nvidia.com)

# Hello!



- ▶ I work on Numba and the CUDA target
- ▶ Joined NVIDIA Dec 2019
- ▶ Other Numba development: Anaconda, 2014-2016
- ▶ Background: languages and compilers for numerical methods and HPC

Thank you for joining this course.

Before we get started I'd like to take a moment to introduce myself, so you can understand where I'm coming from - I'm a senior software engineer in the RAPIDS team, and I started at NVIDIA in December 2019, to work on improving Numba for the CUDA target.

I do a mix of feature development, bug fixing, writing examples, support, and evangelism for Numba.

I had worked on Numba in the past - between 2014 and 2016 I was employed by Anaconda, where part of my job was using and developing it in customer projects.

I won't go through my whole career history because that would be boring and self-indulgent, but in summary my background in languages and compilers for numerical methods and HPC.

You can also see what I look like. I'm happy in this picture, because I'm on holiday!

# Training aims

- ▶ “Power-user” level of knowledge of Numba
  - ▶ How to use it
  - ▶ What it does internally
  - ▶ Understanding performance
  - ▶ Debugging
- ▶ Why “power-user”?
  - ▶ Numba is easy to get started with
  - ▶ For non-trivial applications, deeper knowledge needed

RAPIDS

3

What I hope you’ll get out of the training is to reach the level of knowledge and experience of what I’d call a “power user”.

I think of a Numba power user as someone who is familiar with the user-facing API, and also knows what it’s doing internally.

Knowledge of the internals is vital to understanding and optimising performance, as well as debugging Numba-compiled code effectively.

It’s easy to get started with Numba and accelerate some straightforward use cases. But for non-trivial applications, you really need this level of knowledge to use it effectively.

# Training plan

## Assumptions:

- ▶ Knowledge of CUDA C/C++
- ▶ Low level of familiarity with Python:
- ▶ Will go over relevant details

## Session Plan (adjusted as necessary):

- ▶ Today: Introduction to Python & Numba, background, usage
- ▶ Numba internals
- ▶ TBC...

RAPIDS

4

So how are we going to become power users?

This training material builds on a couple of assumptions:

- Firstly, that you have some knowledge of CUDA C / C++. I'm not planning to go over CUDA concepts, only how to use and apply them with Numba.
- Secondly, I'm trying not to assume too much Python experience.
  - I'm planning to fill in the relevant Python details as we go along.
  - If you've only a vague familiarity with Python I hope you still get something out of this training.

This session is mostly introductory.

- We'll look at Numba and the Python ecosystem and libraries it fits into.
- We'll see what the public API looks like, and how to use it.

In the next session we can dive a bit deeper into how it works, and that will support learning how to optimise performance and debug Numba code.

# What is Numba?

A *Just-in-time (JIT)* compiler for Python functions.

- ▶ **Opt-in:** Numba only compiles the functions you specify
- ▶ Focused on array-oriented and numerical code
  - ▶ **Trade-off:** subset of Python for better performance
- ▶ Alternative to native code, e.g. C / Fortran / Cython / CUDA C/C++
- ▶ Targets:
  - ▶ CPUs: x86, PPC, ARMv7 / v8
  - ▶ GPUs: CUDA, AMD ROC

RAPIDS

5

Now let's finally get on and look at Numba, and what it is. The headline summary is that it's a just-in-time compiler for Python functions.

To use it, you don't need to compile a whole program. You opt-in to using it for specific functions you want to accelerate.

The functions it works best with, especially for CUDA, tend to be those that are heavily numerically focused, with large arrays and lots of data.

Not every Python language feature and library is supported. We'll look more closely at what is supported later.

So, using Numba is a tradeoff - you're working with a limited subset of Python in return for much better performance.

You could avoid using Numba by rewriting your Python as native code using CUDA C or C++, but using Numba provides you with a way to keep all your code in Python.

Numba targets CPUs and GPUs. The CPU target is the most well-developed and featureful, and receives most of the development effort.

The CUDA target works well and has a useful subset of features, but hasn't yet received as much development effort. In some ways that makes it a bit more challenging to use than the CPU target, and we'll go over the differences.

## Why Numba and Python?

- ▶ **PyData ecosystem strength:**

- ▶ Libraries: NumPy, Pandas, scikit-learn, etc.
- ▶ GPU: RAPIDS, PyTorch, CuPy, TensorFlow, JAX, etc.

- ▶ **Comfort Zone:** keeping all code as Python code

- ▶ Allows focus on algorithmic development
- ▶ Minimise development time
- ▶ Maintain interoperability

RAPIDS

6

So why use Numba and keep everything in Python, instead of just rewriting the performance-critical sections of code in something lower level like CUDA C?

I think one reason is that the PyData ecosystem is very strong and well developed - that ecosystem facilitates lots of scientists, engineers, software developers; people who are working in any numerical methods, machine learning or AI applications, and others, are able to do all their work efficiently in Python. There's a few examples of the commonly-used libraries slide, with links.

So, keeping all code as Python code is much more within the comfort zone of the community. It makes it easier for those kinds of users to focus on their algorithmic development, instead of mastering another technology. It also maintains the interoperability that all of that ecosystem already has.

# NumPy

▶ “*NumPy is the fundamental package for scientific computing with Python.*”

- ▶ N-dimensional array objects
- ▶ Many functions operating on array objects
- ▶ Interfaces for third-party libraries to implement

▶ Every Python numerical library built on NumPy

- ▶ Or speaks its interfaces

```
import numpy as np  
  
x = np.arange(10) # Array of integers from 0 to 9  
x = x * 2         # Elementwise multiplication  
np.cos(x)         # One of many functions for  
                  # operating on arrays
```

RAPIDS

7

The fundamental building block of all these libraries in the PyData ecosystem is NumPy. In case you're not familiar with it, we'll take a look at it.





NumPy is a Python library that provides a very convenient way of working with N-dimensional array objects, and it gives you lots of functions that operate on these objects. It also defines an interface for getting at the data of an array, and for wrapping up data from elsewhere as an array. All this provides a pretty powerful foundation for the whole ecosystem.

If you've not looked much at NumPy before, there's a couple of very simple examples that demonstrate it on the slide.

- One shows how you can create an array object and manipulate it element by element with an interface that's very similar to how you'd write the expression down mathematically (for example, multiplying every element by 2).
- The other demonstrates one of the many functions it provides for operating on arrays (cos in this case).

## Who is using Numba?

► PyPI: 50,000 / Conda 15,000 downloads per day

► Github:  Used by 16.3k  Unwatch 209  Star 5.2k  Fork 634

► Random sample of applications:

- [Poliastro](#) (astrodynamics)
- [FBPIC](#) (CUDA-accelerated plasma physics)
- [UMAP](#) (manifold learning)
- [RAPIDS](#) (data science)
- [Talks on more applications](#) in Numba docs

RAPIDS

8

How big is the Numba user community? In all honesty I don't know, but my impression is that it's a pretty successful open source project used by a lot of libraries.

Some metrics that give a picture of its usage come from PyPI, and Conda, which are two popular Python package repositories, showing about 65,000 downloads per day in total.

If we look at the Github repo, there's some interesting numbers there too - I'm not going to try to interpret them, but I think they're large enough to show that Numba is interesting to more than just a handful of people.

I did a quick search to find a few applications of Numba, from different fields of endeavour. Some of the projects, like FBPIC, use Numba for CUDA. RAPIDS itself is a heavy user, particularly in cuDF, for compiling user-defined functions.

There are more applications linked in the Numba documentation, and I've no doubt there are plenty of others too.



## (CUDA) Python vs. (CUDA) C/C++

► Not an exhaustive comparison! just some relevant features to this training:

Feature	C/C++	Python
Typing	Mostly static, strong	Weaker, “duck”
Memory	Programmer-managed	Garbage-collected
Compilation	Ahead-of-time	Runtime to bytecode, interpreted
Usage mode	Write, compile, then run	Write then run / interactive

RAPIDS

9

Just before we start talking about Numba itself, I want to draw attention to some of the relevant differences between Python and C++ to help make sense of why things are the way in Numba, if you're not too familiar with Python.

This comparison is not too exhaustive or detailed - it's just to give an idea. Let's look at each of the rows:

- Python's typing is pretty loose - any type of object is accepted anywhere as long as it supports the right operations and members for where it's being used. It's nicknamed duck typing, because “if it quacks like a duck, it's a duck”.
- You don't have manual memory management in Python. Allocation and deletion is automatic, and objects are garbage collected when they've gone out of scope in all namespaces.
- You don't have to compile Python code ahead of time. When you run it in the interpreter, it is compiled to bytecode and then interpreted, but that's transparent to you as the user. This is in contrast to the compilation and linking process you normally go through with C or C++.
- Also, it's quite common in Python for a developer to work at an interactive prompt, typing in statements and executing them, particularly for exploring and working with data.

## Example 1

```
# A simple example - called once for every pixel

def mandelbrot(x, y, max_iters):
    c = complex(x,y)
    z = 0j
    for i in range(max_iters):
        z = z*z + c
        if z.real * z.real + z.imag * z.imag >= 4:
            return 255 * i // max_iters

    return 255
```

RAPIDS

10

To start getting acquainted with Numba we'll have a look at a few examples of its usage

.

Other Numba tutorials that you might have seen usually save compiling functions and kernels for later, but here I think it's appropriate to start with it as it's the most common way to use Numba.

We'll work an example function that determines if a point is in the mandelbrot set or not.

This is the pure Python implementation of the function.

## Example 1 - CPU JIT

```
# A simple example - called once for every pixel
from numba import jit

@jit
def mandelbrot(x, y, max_iters):
    c = complex(x,y)
    z = 0j
    for i in range(max_iters):
        z = z*z + c
        if z.real * z.real + z.imag * z.imag >= 4:
            return 255 * i // max_iters

    return 255
```

RAPIDS

11

If we want to use Numba to compile this function for the CPU, then we do two things: First, we import the jit decorator from Numba, then we “decorate” the function with it.

If you haven’t seen this syntax before: writing @jit changes the definition of the function. Usually when you define a function, python will then compile it to bytecode as soon as it sees the definition. When you decorate using @, Python will first pass the function to the named decorator (jit) and then the decorator can transform the function definition in some way.

We don’t change anything about the way we call this function. So, for a simple example, there’s very little work involved in turning Pure python code into compiled native code.

## Example 1 - CUDA JIT

```
# A simple example - called once for every pixel
from numba import cuda

@cuda.jit(device=True)
def mandelbrot(x, y, max_iters):
    c = complex(x,y)
    z = 0j
    for i in range(max_iters):
        z = z*z + c
        if z.real * z.real + z.imag * z.imag >= 4:
            return 255 * i // max_iters

    return 255
```

RAPIDS

12

How would we do the same thing - compiling the function, but to run on a GPU? It's fairly similar, except we use the `cuda.jit` decorator instead.

For this example, I've also passed it the `device=True` keyword argument. This is so that rather than creating a global kernel function, we compile a device function instead. This is because this example only operates on a single pixel, which is not a very good work granularity to launch on a GPU.

So on the next slide we'll look at the outer loop.

## Example 1 continued - caller

```
# The whole image loop

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 = mandelbrot(real, imag, iters)
            image[y, x] = color
```

RAPIDS

13

This is the calling function. It takes a 2D array called image, and calls the mandelbrot function for every pixel in that image.

This is a bit more suitable for a GPU, with millions of pixels in a typical image.

## Example 1 continued - caller JIT

```
# The whole image loop
from numba import jit

@jit
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 = mandelbrot(real, imag, iters)
            image[y, x] = color
```

RAPIDS

14

If we wanted to compile the outer function for the CPU, then we would use the `jit` decorator, like we did for the `mandelbrot` function.

This works fine - one compiled, jitted function can call another one.

## Example 1 continued - caller CUDA JIT

```
# The whole image loop
from numba import cuda

@cuda.jit
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

    x, y = cuda.grid(2) # x = blockIdx.x * blockDim.x + threadIdx.x
    if x < width and y < height:
        real = min_x + x * pixel_size_x
        imag = min_y + y * pixel_size_y
        color = mandelbrot(real, imag, iters)
        image[y, x] = color
```

RAPIDS

15

Now let's look at compiling it into a CUDA kernel with Numba. We use `cuda.jit` again, but that's not the only change we have to make.

The original code contained two for loops: one over the width of the image and one over the height. To parallelise those loops across threads, the loop structure is flattened so that each pixel is assigned to one thread.

Numba provides a convenience function called `grid`, which we're using here. You call it with the number of dimensions of the grid, and it returns N linear indices. You can think of this as a shorthand for computations with the block dimensions and index, and the thread index.

The grid can often be a little bigger than the image, so we add a guard to make sure that only threads within the image bounds do the computation.

Those are all the changes made to make this into a global kernel function. Conceptually they're very similar to CUDA C, and only the language is a little bit different.

## Example 1 continued - call site

Pure Python / CPU JIT:

```
create_fractal(-2.0, 1.0, -1.0, 1.0, image, 20)
```

CUDA JIT:

```
# Create grid of 32x32 blocks, one thread per pixel
nthreads = 32
nblocksy = (height // nthreads) + 1
nblocksx = (width // nthreads) + 1

config = (nblocksx, nblocksy), (nthreads, nthreads)
create_fractal[config](-2.0, 1.0, -1.0, 1.0, image, 20)
# C: create_fractal<<<>>>
```

RAPIDS

16

Now, what about calling the compiled function, or launching the kernel?

For the pure Python implementation, and the CPU-compiled version, the call site is the same. It's just a straightforward function call.

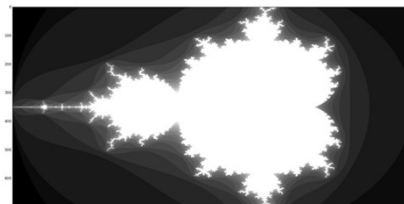
For the CUDA kernel, similar to CUDA C, we need to configure the kernel for launch. The syntax is a little different in CUDA Python because there's no way for Python to accept the herringbone syntax for launch parameters. Instead, we use square brackets for the configuration.

Here we create a grid of 32x32 blocks that's just big enough to contain the image, and launch with that configuration.



## Example 1 - Performance

- ▶ i7-6700K
- ▶ Quadro RTX 8000
- ▶ 20 iterations of Mandelbrot:



Implementation	Speedup
CPython	1x
Numba (CPU)	71x
Numba (CUDA Kernel)	190x
Numba (CUDA UFunc)	347x

RAPIDS

17

For now I'm not going into the specifics of performance measurements too much, but I have these numbers to give an idea of the orders of magnitudes involved.

The CPython implementation is the baseline. CPython is the most commonly used Python interpreter, and it executes the uncompiled version of the function.

The Numba-jitted version compiled for the CPU is about 70 times faster than pure Python. That's in line with what you'd expect the performance of the function to be if it were written in C.

The version rewritten as a CUDA kernel is about 190x faster than the CPython baseline. Another CUDA version, using a *universal function*, or *ufunc*, is nearly twice as fast again. We haven't looked at ufuncs so far, but will come to them later on.

Obviously we can expect wide variations in these speedup numbers for different problems, different CPUs and GPUs, etc. However, this example does provide some illustration.

## Example summary

- ▶ CUDA Python is to Python as CUDA C/C++ is to C/C++
- ▶ To compile CUDA kernels from Python:
  - ▶ Import the `cuda.jit` decorator
  - ▶ Decorate appropriate functions with `cuda.jit` (and `device=True`)
  - ▶ Add launch configuration to call sites
- ▶ Differences:
  - ▶ Absence of compiler invocation
  - ▶ Absence of types
- ▶ [CUDA Python user documentation](#)
- ▶ [CUDA Python reference](#)

RAPIDS

18

Some of the takeaways from the example we've just seen are:

- CUDA in Python is very similar to CUDA in C in a lot of ways, even though there's some syntactic differences.
- To create cuda kernels in Python, you need to import the `cuda.jit` decorator, decorate appropriate functions with it, and add the launch configuration to the call sites.

There's also some obvious differences:

- What happened with compilation? We didn't invoke anything like `nvcc`, for example.
- Where are the types?

We'll spend some time in the rest of this session looking at both these things.

## What just happened?

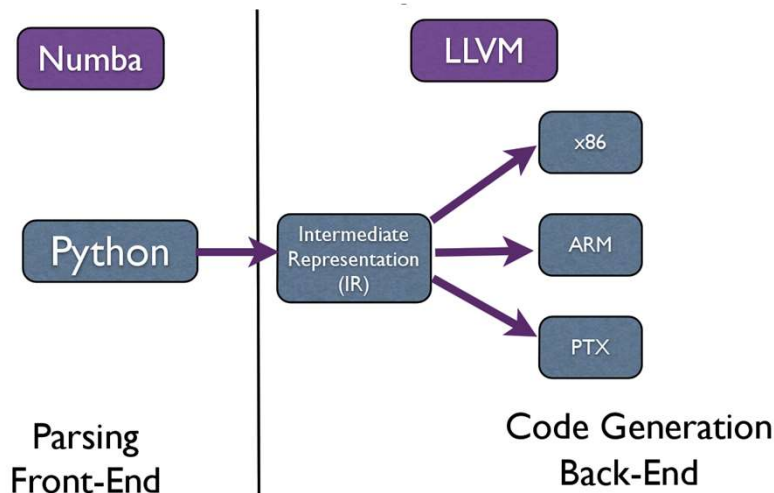
- **Just-in-time:** compilation at call time
- **Type-specialisation:** typing inferred from argument types

When we call a jitted function, the compilation happens *Just-In-Time (JIT)*: if we haven't already compiled the function, Numba does it there and then.

It compiles a version of the function for each different set of argument types.

# Architecture overview

LLVM: A compiler infrastructure



RAPIDS 20

We'll have a quick high-level look at what Numba does when it compiles a function.

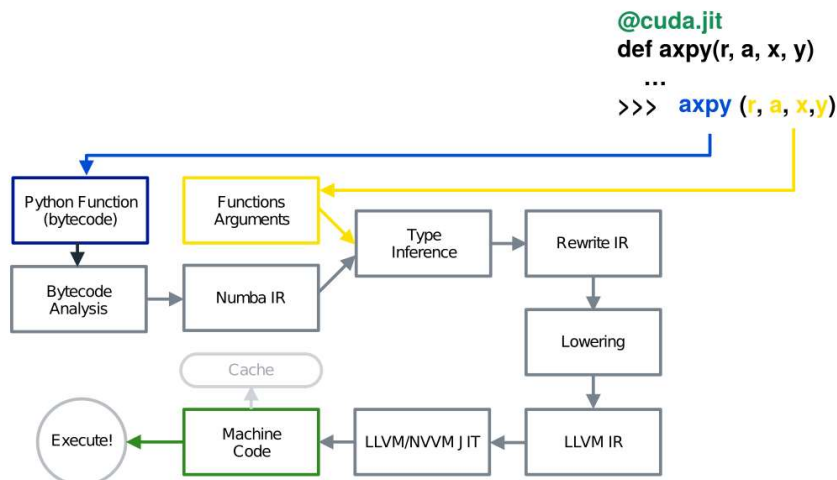
It takes the Python code of the function and transforms it into LLVM *Intermediate Representation (IR)*. Then, it uses LLVM to generate assembly code for the target hardware.

If you haven't come across LLVM before, it's a fairly widely-used compiler infrastructure that supports various different targets.

LLVM is used in lots of compiler implementations, including NVIDIA's own NVVM for generating PTX code.

LLVM makes it comparatively easy to write a compiler because you only have to write a language frontend that creates LLVM IR, and saves you from having to do any target code generation yourself.

## Pipeline in detail



RAPIDS 21

How does Numba translate Python into LLVM IR? Here's an overview of the pipeline that does that.

Numba doesn't start with the source code itself; first, it uses the Python interpreter's bytecode compiler to generate bytecode, which is much simpler to interpret than the source.

It needs to translate the bytecode into the Numba Intermediate Representation (Numba IR) that it can use to work out the types of all values in the function. The Bytecode Analysis stage produces a Control Flow Graph (CFG) and Data Flow Graph (DFG) that is used to construct the Numba IR.

Then Type inference assigns a type to every input, intermediate value, and output in the Numba IR.

There are some rewrite passes on the IR that optimize and simplify it - these aren't covered further in this course.

The next stage is Lowering, which generates the LLVM Intermediate Representation (LLVM IR). The process of translating the typed Numba IR into LLVM IR is relatively mechanical.

To generating PTX it uses NVVM, which is an LLVM-based PTX generator from NVIDIA.

After that, the compiled code can be put in the cache for re-use, and then executed.

## Dispatch process

Calling a @jit function:

1. Lookup types of arguments
2. Do any compiled versions match the types of these arguments?
  - a) Yes: retrieve the compiled code from the cache
  - b) No: compile a new specialisation
3. Marshal arguments to native values
4. Call the native code function
5. Marshal the native return value to a Python value

RAPIDS

22

At the time of a call to a jitted function, Numba's dispatch process takes over.

The dispatcher inspects the function's arguments to see what types they are, then checks whether there's already a compiled version of the function that matches these argument types.

If it finds a match, it retrieves that compiled version from the cache. If not, it kicks off the compilation pipeline we saw in the previous slide.

Python objects can't be directly passed to native code functions - the data within them needs to be "unboxed" into its native form to pass to them, so Numba unboxes all the arguments.

It then calls the native function with the unboxed arguments, and gets a return value back from it.

Finally, the native return value gets boxed up into a Python object to pass back to the interpreter.

## CUDA Dispatch process

Extra work for calling a @cuda.jit function:

1. **Compilation:** use NVVM for LLVM IR -> PTX
2. **Linking:** create a module (cubin) with driver API
3. **Loading:** load module with driver API
4. **Data transfer:** move data in host memory to GPU
5. **Kernel launch:** more marshalling, call through driver API

RAPIDS

23

When a CUDA-jitted function is called, there's a little more work for Numba to do.

After it's got the PTX code from NVVM, it uses the CUDA Driver API to link the PTX, which creates a module that can be loaded to the device.

If any of the inputs are in host memory, then they need to be transferred to the device.

Once all data is on the device, the kernel can be launched. Numba creates a list of parameters to pass to the kernel launch and uses the Driver API to launch with the requested configuration.

## A deeper look at CUDA Python

- Components
- Memory Management
- Supported Python features

We've spent some time looking at the compiler. Now, we'll look at bit more at the other components of CUDA Python.



# What actually is CUDA Python?

- ▶ A Python-to-PTX compiler that uses NVVM
  - ▶ (`@cuda.jit`)
- ▶ A Python wrapper to the driver API
  - ▶ `cuInit`, `cuCtxCreate`, and many more...
  - ▶ Transparent for most use cases
- ▶ A NumPy-like array library for CUDA GPUs
  - ▶ *Device Arrays*
  - ▶ c.f. [CuPy](#)

RAPIDS

25

So, what does it consist of?

Besides the compiler, it also contains a Python wrapper to the driver API. This is what's used under the hood, and most of the time you don't need to know what it's doing.

The other component is a NumPy-like array library, used for managing arrays on CUDA devices. These are referred to as device arrays.

There are other NumPy-like libraries, with CuPy being the most widely used one at the moment. Numba can work with these objects too, but for now we'll stick to focusing on Numba's Device Arrays.

## Memory Management

```
import numpy as np
from numba import cuda

@cuda.jit
def add(r, x, y):
    i = cuda.grid(1)
    if i < len(r):
        r[i] = x[i] + y[i]

# Create arrays on host
x = np.arange(10)
y = x * 2
r = np.zeros_like(x)

# Transfers to and from host memory implied
add[config](r, x, y)
```

RAPIDS

26

For a look at memory management and Numba device arrays, we'll use this short example. It's a kernel that takes two input vectors, `x` and `y`, adds their elements, and stores the result in another vector.

If we create some NumPy arrays on the host then call the kernel, Numba automatically determines that it needs to transfer them to the device, and also transfer them back after execution.

That's convenient, but not particularly efficient - in most cases, you want to keep data on the GPU and launch lots of kernels on it without being interrupted by transfers between the device and the host.

## Manual memory management

```
d_x = cuda.to_device(x)
d_y = cuda.to_device(y)
d_r = cuda.device_array_like(x)

# No transfer implied
add[config](d_r, d_x, d_y)

# Bring data back when needed
r = d_r.copy_to_host()
```

► Creating device arrays:

```
# 1024 x 2048 matrix of float32
arr = cuda.device_array((1024, 2048), dtype=np.float32)
```

RAPIDS

27

So, we can use the APIs for working with device arrays to manually manage the transfer and creation of arrays on the device

.

Here we transfer the x and y arrays to the device ahead of time. What's in the r array doesn't really matter because it's going to be overwritten. For that, we can just create a device array that's the same shape and data type as x.

Now when we call the add function with these objects, Numba recognises that these are all device arrays and no transfer is invoked.

If we do want to view the results on the host and not just feed them into another kernel, then we can use the `copy_to_host()` method to get a NumPy array back on the host.

You don't have to have an array on the host to create one on the device - you can also use the device array constructor to create one. In the second example, we're creating a matrix of float32 data.

# Freeing memory

- ▶ No direct `cudaFree` / `cuMemFree` equivalent
- ▶ Python is garbage-collected
- ▶ When array object GC'd, Numba finalizer releases memory (eventually)

```
# Remove reference from current namespace:  
del d_r
```

- ▶ [Deallocation Behaviour](#)
- ▶ [Numba Memory Management documentation](#)

RAPIDS 28

Having allocated memory on the device, how do we deallocate it again?

As I mentioned before, there is no explicit freeing of Python objects, so there's no direct way to tell Numba to free an array.

Numba handles deallocation by freeing the underlying memory when the device array object holding it is garbage collected.

In a nutshell, objects are garbage collected when they have no references from any namespaces. If you want to remove a reference to an object from the current namespace, you can use the `del` operator.

In the example on the slide we use `del d_r`, and as long as there are no other reference stored anywhere, Numba will mark the underlying memory as ready to be freed, and then free it the next time it does a cleanup.

Cleanup can be a bit of a complicated topic - there is some explanation of the exact strategy used for cleaning up in the manual section on deallocation behaviour.

There's also some more general documentation on the APIs for allocating memory, and creating device arrays that covers all the different ways of allocating memory.

## Supported Python syntax

Inside functions decorated with `@cuda.jit`:

- ▶ assignment, indexing, arithmetic
- ▶ if / else / for / while / break / continue
- ▶ raising exceptions
- ▶ assert, when passing `debug=True`
- ▶ calling other compiled functions (CPU or CUDA jit)

[Documentation on supported language constructs](#)

As I mentioned earlier, not all of Python is supported in jitted functions. The list here gives an overview of Python syntax that is supported inside CUDA kernels. There is a more complete list in the linked documentation.

# Unsupported Python syntax

Also inside functions decorated with `@cuda.jit`:

- ▶ `try / except / finally`
- ▶ `with`
- ▶ (list, set, dict) comprehensions
- ▶ generators

Classes cannot be decorated with `@cuda.jit`

Here's an overview of the syntactic features that aren't supported - again, the documentation linked on the previous slide gives more details.

Classes are unsupported for CUDA - JIT class support is limited to the CPU target.

# Supported Python features

- ▶ Types:

- ▶ int, bool, float, complex
- ▶ tuple, None
- ▶ [Documentation on supported types](#)

- ▶ Built-in functions:

- ▶ abs, enumerate, len, min, max, print, range, round, zip
- ▶ [Documentation on supported builtins](#)

The documentation also enumerates the supported types and builtins. The main supported builtin types that are used in CUDA kernels are the numeric types and tuples.

Some of the supported builtin functions are also listed here.

## Supported Python modules

- ▶ Standard library:
  - ▶ cmath, math, operator
  - ▶ [Comprehensive list in documentation](#)
- ▶ NumPy:
  - ▶ Arrays: scalar and structured type
    - ▶ except when containing Python objects
  - ▶ Array attributes: shape, strides, etc.
  - ▶ indexing, slicing
  - ▶ Scalar types and values (including datetime types)
  - ▶ Scalar ufuncs (e.g. np.sin)

Numba needs support built-in to it for each Python module that it compiles. From the standard library, the maths and operator modules are supported in CUDA.

The NumPy API is partially supported - in general arrays and scalar operations on them are well-supported, alongside indexing and slicing them.

Scalar types and values are also supported.



## Unsupported NumPy functions

- ▶ Array creation
- ▶ Functions returning a new array
- ▶ Array methods (e.g. `x.mean()`)

Presently it's not possible to allocate arrays inside kernels. All arrays need to be passed into kernels instead.

This also means that functions that would return a new array aren't supported.

Methods that operate on entire arrays are also unsupported - usually when porting a function to run as a CUDA kernel, methods like these need to be re-written using loops, indexing, and scalar operations.

## Creating Ufuncs and GUFuncs

The final section of this session looks at using Numba to write *universal functions* (*ufuncs*) in Python. These provide a simple way of writing element-by-element on arrays, and are easier to write than JITted kernels, but are more limited in how they access data.

## Example 2 - vectorize

- ▶ **UFuncs:** operate element-by-element on arrays
- ▶ Supports broadcasting, reduction, accumulation, etc.

```
@vectorize
def rel_diff(x, y):
    return 2 * (x - y) / (x + y)
```

Call:

```
a = np.arange(1000, dtype = float32)
b = a * 2 + 1
rel_diff(a, b)
```

RAPIDS

35

We'll start by looking at an example of a ufunc. To write one, you apply the `@vectorize` decorator to a function that accepts scalar inputs, and returns a scalar output. When the ufunc is called on array inputs, NumPy takes care of applying the compiled code to each element of the array and assigning the scalar results into an output array.

The example on the slide calculates the relative difference of two scalars. In the final line of the example, it's called on the arrays `a` and `b`.

This example compiles and executes the ufunc on the CPU.

## Example 2 continued - CUDA vectorize

- ▶ Add target='cuda' to generate CUDA implementation

- ▶ **Note:** CUDA target needs type specification

```
@vectorize([float32(float32, float32)], target='cuda')
def rel_diff(x, y):
    return 2 * (x - y) / (x + y)
```

Call (no change):

```
a = np.arange(1000, dtype = float32)
b = a * 2 + 1
rel_diff(a, b)
```

RAPIDS

36

To compile and execute the ufunc on a GPU, we don't have to change much - just add the target=cuda kwarg to the vectorize decorator, and a specification of the types of the inputs and outputs that the function will be called with.

The inputs and outputs are given as a list of signatures, so you can compile the same ufunc for multiple types on the GPU.

Calling the function remains the same - here we're calling the ufunc with two NumPy arrays, so Numba takes care of transferring the data to and from the device around the call - later we'll see how to manage data movement more efficiently.

## Example 2 continued - prioritizing types

► Multiple signatures:

```
@vectorize([float32(float32, float32),  
            float64(float64, float64)],  
            target='cuda')
```

► Always uses float64 signature, even for float32 arguments:

```
@vectorize([float64(float64, float64),  
            float32(float32, float32)],  
            target='cuda')
```

RAPIDS

37

When a ufunc is called, the list of signatures is searched for one that matches the types of the arguments, and the first one that will apply to the given arguments is used.

It's therefore important to think about the ordering of the signatures in the list - in the second example, passing float32 arrays will always result in the float64 version being used, because it came first in the list, and float32 can be cast to float64 with no loss of precision.

However, this casting results in much more overhead due to the creation of temporary casted arrays, as well as executing a compiled version of the function that needs more resources.

Ordering signatures from the narrowest type to the widest type, and placing signatures with integer types before floating point ones, should generally result in an optimal choice of compiled function for any given arguments.

## Example 3 - guvectorize

Operate on an arbitrary number of elements. Example:

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

- ▶ No return value: output is passed in
- ▶ Input and output layouts: (n),()->(n)
- ▶ Before ->: Inputs, not allocated. After: outputs, allocated
- ▶ Don't rely on in-place modification!

See guvectorize notebook in exercises.

RAPIDS 38

Conceptually, *Generalized UFuncs* (*gufuncs*) are like ufuncs, that operate on an arbitrary number of elements at a time.

To build a gufunc, we use the guvectorize decorator. This decorator needs several things:

- A list of signatures. Signatures are similar to ufunc signatures, but the dimension of each argument also needs to be given using a comma-separated list of colons.
- A layout specification. This is a string that gives the relationships between the shapes of the inputs and outputs. Input shapes are given before the ->, and outputs after it.
- The target kwarg, if the gufunc is to run on a CUDA GPU.

Instead of returning an output, the output for a gufunc is passed in.

The example g function accepts two arguments, a vector and a scalar, and has one vector output.

- In the signature, both the vector and scalar input have shape [:] - the need to make scalar inputs 1-dimensional is a slight idiosyncrasy in the interface.
- In the layout specification, the inputs are (n),() - an n-length vector and a scalar. The output is (n), so the output length is equal to the first input length.

You can use the notebook on gufuncs to execute them and get a feel for how the shapes of the inputs to the gufunc get applied to actual input arguments.

## Layout examples

### ► Matrix-vector products:

```
@guvectorize([(float64[:, :], float64[:, :], float64[:])],
              '(m,n),(n)->(m)')
def batch_matmul(M, v, y):
    pass # ...
```

### ► Fixed outputs (e.g. max and min):

```
@guvectorize([(float64[:, :], float64[:, :], float64[:])],
              '(n)->(),()')
def max_min(arr, largest, smallest):
    pass # ...
```

RAPIDS

39

Here are two more examples of signatures and layout specifications to help make the idea clearer.

The `batch_matmul` gufunc (implementation omitted) performs matrix-vector multiplication for a set of matrices and vectors. It has:

- One 2D input, `M`, with type `float64[:,:]` and shape `(m,n)`.
- One 1D input, `v`, with type `float64[:]`, and shape `(n)` - the same length as the number of columns in `M`.
- One 1D output, `y`, with type `float64[:]` and shape `(m)` - the same length as the number of rows in `M`.

Another gufunc limitation is that only symbolic dimensions can be specified - for example, we can't specify an output shape of `(2)`. To get around this, we can specify two scalar outputs instead. This is done for the `max_min` gufunc, which finds the maximum and minimum of a batch of arrays. It has:

- One 1D input, `arr`, with type `float64[:]`, and shape `(n)`.
- Two scalar outputs, with type `float64[:]`, and shape `()`.

## Summary / conclusions

- ▶ Numba is a JIT compiler focused on compiling type-specialised versions of numerically-focused code.
- ▶ Supports a restricted subset of Python
- ▶ CUDA Python enables writing CUDA kernels
- ▶ Memory management necessary for best performance
- ▶ UFuncs and GUFuncs can offer a quick route to performance for simple algorithms

RAPIDS

40

We've used this session to take a quick tour of Numba and its main capabilities - compiling a subset of numerically-focused Python code for CPUs and GPUs.

We've scratched the surface of memory management for performance optimisation, and will examine it in more detail in later sessions.

We've also seen how ufuncs and gufuncs can provide a quick route to writing GPU-accelerated functions that operate elementwise on arrays.



# Exercises

Accompanying notebooks:

- ▶ `kernels.ipynb`: Writing CUDA kernels with the `@jit` decorator.
- ▶ `vectorize.ipynb`: The `@vectorize` decorator.
- ▶ `guvectorize.ipynb`: The `@guvectorize` decorator.
- ▶ `cuda-ufuncs-and-memory-management.ipynb`: More ufunc examples and memory management.

RAPIDS

41

The accompanying notebooks provide some opportunity to experiment with the concepts we've introduced in this session.

Some of the examples go beyond the concepts introduced here (for example, introducing atomic operations) - they will be covered more thoroughly in a later session.

Thankyou! / Questions?



**RAPIDS**