

RAPIDS

Numba for CUDA Programmers

Graham Markall - 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:  16.3k  209  5.2k  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?

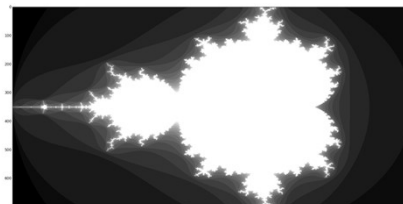
For the pure Python implementation, and the CPU-compiled version, the call site is the same. It's just a straightforward function call.

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?

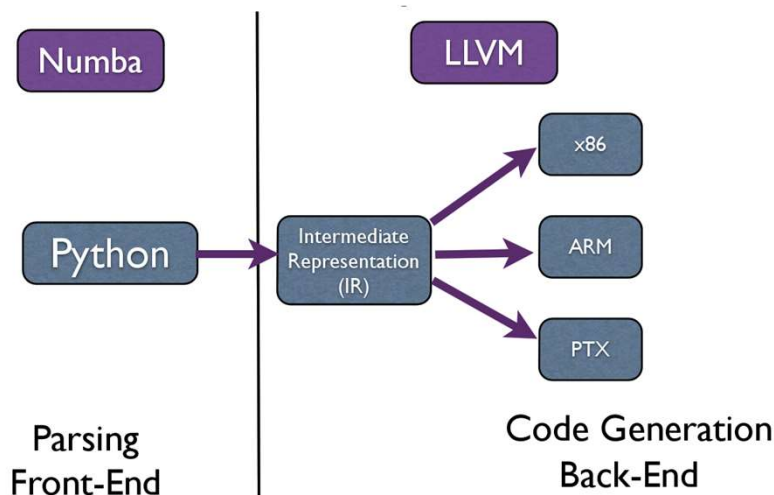
- **Just-in-time:** compilation at call time
- **Type-specialisation:** typing inferred from argument types

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.

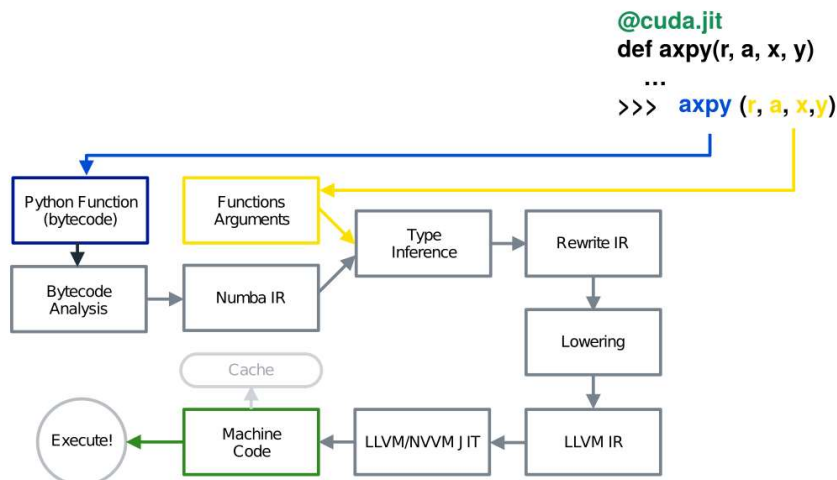
It takes the Python code of the function and transforms it into LLVM *Intermediate Representation (IR)*. Then, it uses LLVM to generate assembly code for the target hardware.

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)

RAPIDS

32

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

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

RAPIDS

Numba for CUDA Programmers Session 2 - Typing

Graham Markall - gmarkall@nvidia.com

Review from last week

- ▶ Numba: Python JIT Compiler that targets CPUs and GPUs
- ▶ Supported language / libraries
- ▶ `@cuda.jit`, `@vectorize`, `@guvectorize`
- ▶ Memory management `@cuda.to_device(...)` etc...
- ▶ Brief look at the compilation pipeline

Just before we begin, a quick reminder of what we looked at last week - Numba is a just-in-time compiler for Python that supports a subset of the python language in return for making code you compile with it go much faster.

You use it by annotating functions you want to compile with things like `cuda.jit`, or `vectorize` and `guvectorize`.

We looked at how to manage data between the host and the device and had a high-level look at its compilation pipeline.

This week: typing

- ▶ The most significant difference between CUDA C and CUDA Python
 - ▶ Native code is statically typed
 - ▶ Python is not
- ▶ Crucial to understanding and optimising performance
- ▶ How Numba's typing mechanism works
- ▶ Various examples and tricky cases

RAPIDS

3

This week we're going to look in depth at typing, which is one part of that pipeline.

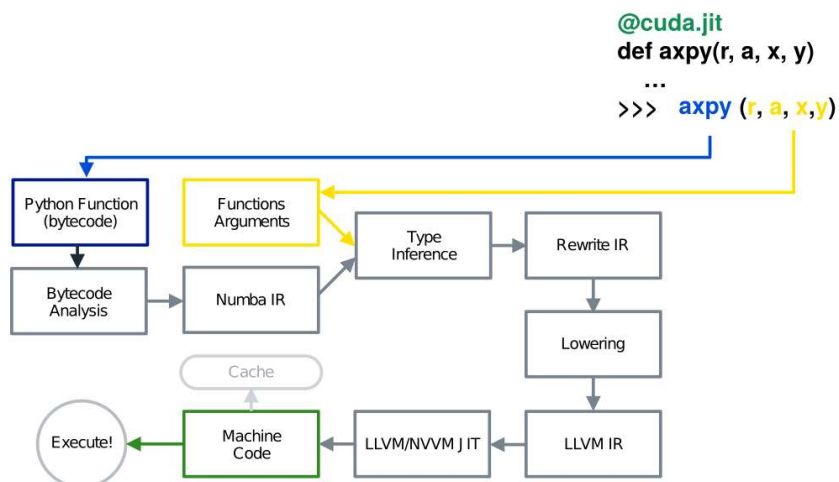
It's the biggest difference between C and Python with CUDA - to get to native code, you need to statically type everything at some point. However, Python has no static typing, unlike CUDA C.

One reason to look at typing is that you need to understand what it's doing to get the best performance out of Numba. If you don't keep an eye on what typings you get, then you can end up leaving performance on the table.

So to understand it, we'll first look at how typing works, then we'll go over some examples and tricky cases. In particular, some aspects of Numba's typing can make it tricky to optimise code for CUDA.

When we go through the examples, just try to follow and understand the general idea - don't try to memorise them, because you can always just refer to them later.

Pipeline in detail



This is Numba's pipeline, which we saw briefly last week.

In this session we're focusing only on the type inference stage.

Type inference works on the Numba IR, but mostly I'll talk about it with examples in terms of Python code, because that's a bit easier to think and talk about.

When you do start looking at the outputs of the type inference stage, you see the typing of the IR instead, and we do have some examples of that.

Type inference is really a key stage that connects the Python frontend to the LLVM backend - once you have a typing for a function, it's pretty much ready to be lowered into LLVM IR.

The Rewrite stage is where some optimisations happen, but it's not particularly crucial.

How typing works

- ▶ Numba has to determine types by propagating type information
- ▶ Type inference uses:
 - ▶ The data flow graph
 - ▶ Mappings of input to output types
- ▶ “No regrets” - it does not backtrack

```
def f(a, b):    # a:= float32, b:= float32
    c = a + b   # c:= float32
    return c    # return := float32
```

RAPIDS

5

Let's start looking at how type inference works.

The only type information Numba can start with is the types of the arguments passed into a function. It has to work out a type for every intermediate value and return value within the function. So, it has to propagate type information through the function, starting with the arguments, all the way through to each return statement.

It can do this by using two things.

- The first is the data flow graph, which it worked out very early on in the the pipeline. This tells it what the inputs and outputs of each expression in the function are.
- The second is a set of mappings from input types to output types, that it stores internally. These mappings are things like “adding two float64s together produces another float64”.

Propagation usually happens in a forward direction. If Numba's assigned a type to a variable, it generally doesn't go back and revisit that decision.

To make some of this concrete, we'll look at some examples. Here we'll assume the function is called where a and b are both float32.

- At the start of type inference, only the types of a and b are known
- So for the first propagation step, Numba has to determine the type of c
- This is an addition of two float32s, so that's going to result in another float32, according to Numba's internal mappings
- Then we can propagate again to the return statement.
- Because the type of c is float32, the return type is determined to be float32.

So, you can think of type inference as working through functions line-by-line.

Type unification (1)

Example typing 1:

```
def select(a, b, c): # a := float32, b := float32, c := bool
    if c:
        ret = a      # ret := float32
    else:
        ret = b      # ret := float32
    return ret       # return := {float32, float32}
                    #          => float32
```

RAPIDS

6

That first example was pretty straightforward, with propagation going through a straight line of code. But what happens when you have branches?

Let's have a look at another example, where we have an if condition used to select which value to return. First, we'll look at a case where the function is called with float32 values for a and b.

- In the if branch, the type of ret is float32
- In the else branch the type of ret is float32.
- Now when the control flow comes back together on the line with the return, we don't have a single type for the ret variable
 - Instead Numba constructs a set of types for the variable,
 - Then it tries to unify them into a single type that can represent values of all the types in the set.
 - Here the set is two float32s, which unifies quite straightforwardly to a float32.

Now, the typing is complete.

Type unification (2)

Example typing 2:

```
def select(a, b, c): # a := float32, b := float64, c := bool
    if c:
        ret = a      # ret := float32
    else:
        ret = b      # ret := float64
    return ret       # return := {float32, float64}
                    #          => float64
```

RAPIDS

7

Now let's have a look at another typing. This time we call the function with a float32 and a float64.

After propagation we end up with the return type set including a float32 and a float64.

These unify to a float64, because that's the type that can represent all the values of both a float32 and a float64.

Type unification (3)

Example typing 3:

```
def select(a, b, c): # a := tuple(int32, int32), b := float32, c := bool
    if c:
        ret = a      # ret := tuple(int32, int32)
    else:
        ret = b      # ret := float32
    return ret       # return := {tuple(int32, int32), float32}
                    # => XXX
```

RAPIDS

8

Sometimes unification can't succeed with the set of types it ends up with. This is an example of that happening.

If we call the function where *a* is a tuple of *int32*s and *b* is a *float32*, then we end up with a set containing a tuple and a *float32*.

Here the unification has to fail, because there's no single native type that can represent both a tuple of *int32*s and a *float32*.

You'll get a typing error and the compilation fails.

Unification error

```
TypeError: Failed in nopython mode pipeline
(step: nopython frontend)
Cannot unify array(int64, 1d, C) and Literal[int](10) for 'x.2'
```

Treating a variable as an array in one place and a scalar in another

```
@cuda.jit
def f(x, sel):
    for i in range(10):
        if i == 8:
            x = 10
        else:
            x[0] = 10
```

RAPIDS

9

When you get a typing error because of a unification problem, you'll get an error that looks something like this.

A `TypeError` is thrown, and the message makes some mention of what can't be unified.

In this particular example I'm treating the variable `x` as a scalar in one place and as an array in another. In the `if` branch I assigned `x` a literal integer, and in the `else` branch I tried to treat it like an array.

I find that this is a fairly common mistake that I make, especially when I'm getting a bit confused about what shape something is, which is why I've picked this example.

Interpreting type errors

Confusion with array dimensions / scalars:

```
x[0] = 10.0  
x[0, 0] = 2.0
```

Error:

```
TypeError: Failed in nopython mode pipeline  
          (step: nopython frontend)  
Invalid use of Function(<built-in function setitem>)  
  with argument(s) of type(s):  
    (array(int64, 1d, C),  
     UniTuple(Literal[int](0) x 2),  
     Literal[int](10))  
...  
This error is usually caused by passing an argument  
  of a type that is unsupported by the named function.
```

RAPIDS

10

Sometimes you can get other kinds of type error. Here is an example of a non-unification-related typing error - I'm treating x as a 1D array and then a 2D array in the same straight line of code.

Let's try and decipher the error message.

- The call to built in function setitem refers to a subscripting assignment.
- The arguments it gives are:
 - The thing being subscripted, which it thinks is a 1D array
 - The subscript, which is a tuple of two integers
 - and the value being assigned.

In other words, Numba is saying that it doesn't know of a way to index a 1D array with a 2D index. It couldn't find a mapping of the setitem function from those input types to any output type.

It's already set on the idea that x is 1D because we treated it like a 1D array first on the previous line.

In general, these "invalid use of function" messages can indicate an error in your code, but you might also see them if you try to do something not supported by Numba.

Unsupported functions

```
@cuda.jit
def sum_reduce(x):
    x[0] = x.sum()

x = np.ones(10)
sum_reduce(x)
```

```
TypeError: Failed in nopython mode pipeline
          (step: nopython mode backend)

Use of unsupported NumPy function 'numpy.nditer'
or unsupported use of the function.

File "numba/np/arraymath.py", line 167:
    def array_sum_impl(arr):
        <source elided>
        c = zero
        for v in np.nditer(arr):
```

RAPIDS

11

Numba does also try to tell you if you use an unsupported function. In this example the sum function of a NumPy array isn't supported in the CUDA target, so we get a typing error here too.

The only odd thing about it is that it talks about `numpy.nditer` being unsupported. The reason we see this is because some of Numba is implemented using Numba - there isn't a direct implementation of the sum function, but instead Numba is trying to compile some Python code that implements sum using `nditer`.

So if you see messages about unsupported functions you haven't used, it's most likely down to the internal implementation of a function that you have used.

Branch elimination

Some branches are removed if Numba can tell they are never taken:

```
@jit(nopython=True)
def branch_elim_example(a, b, cond):
    if cond is None:
        return a
    else:
        return b
```

RAPIDS

12

Another thing to be aware of is that you can sometimes not get a unification error when you would have been expecting one.

Sometimes, Numba can work out that a branch is never taken and then ignore the code of that branch, which avoids having to do unification on any types in it.

In this example, checking if `cond` is `None` is a test that Numba can see will always be true or always be false for a given compilation of the function.

Next, we'll look at examples branch-elimination in the working and non-working cases.

Branch elimination: working

```
@jit(nopython=True)
def branch_elim_example(a, b, cond):
    if cond is None:
        return a    # return type = int64
    #else:
    #    return b

# Works: else branch eliminated
branch_elim_example(1, (1, 2), None)
```

RAPIDS

13

If we call the function where *a* is a scalar, *b* is a tuple, and *cond* is *None*, then:

- After Numba's branch elimination analysis, it's as if we'd written code that only executes "return *a*"
- The other side of the branch returning *b* is deleted

So, the typing succeeds, even though it wouldn't be possible to unify a scalar with a tuple under normal circumstances.

Branch elimination: non-working

```
@jit(nopython=True)
def branch_elim_example(a, b, cond):
    if cond is None:
        return a      # return type = int64
    else:
        return b      # return type = UniTuple(int64 x 2)
    # return type = {int64, UniTuple(int64 x 2)}

# Typing error: else branch not eliminated
branch_elim_example(1, (1, 2), True)
```

General advice:

- ▶ If some calls fail to unify,
- ▶ then branch elimination may be involved

RAPIDS

14

Now what happens if we make the same call again with an int and a tuple, but this time cond is True?

- In this case Numba's branch elimination logic isn't quite clever enough to remove the if branch
- So it tries to unify an int with a tuple of ints, and then we're going to get a typing error.

As a general rule, if you have a function that works for some inputs that you pass into it, but not others, then it might be that branch elimination is able to delete part of the code of your function for the working calls.

If this happens, you might need to reorganise the function's code to avoid unification errors.

CUDA-specific issues

- Widening unification
- Widening constants
- Widening integer operations
- Register usage

We've just covered most of the general points and problems to do with typing.

There are some CUDA-specific issues to cover for the remainder of this session.

They're pretty much all focused on keeping the width of types under control because you typically want to do things with the smallest type possible in CUDA.

This is both because you have more arithmetic units available for smaller types, and also because you have to keep register usage down to keep occupancy high.

So, we'll look at the danger zones where you can end up with types wider than you need, and also into monitoring and controlling register usage.

Widening unification

```
@jit(nopython=True)
def select(a, b, threshold, value):
    if threshold < value:
        r = a    # r: float32
    else:
        r = b    # r: int32
    return r    # r: {float32, int32} unifies to float64

a = np.float32(1)
b = np.int32(2)
select(a, b, 10, 11) # Call with (float32, int32, int64, int64)
```

RAPIDS

16

First of all, sometimes unification can give you output types wider than any of the inputs. Here's one example of that happening.

It's fairly similar to the other unification examples, but in this case, the set of return values to unify is a float32 and int32.

It might be slightly surprising that this unifies to float64. The rationale for that is that integers can't be used to represent floats, and a float32 doesn't have as much precision as an int32 - only about 7 significant figures are accurate. So Numba's choice of float64 provides about 16 significant digits, and the ability to represent floats. This choice might not be perfect for all cases, but provides a reasonable tradeoff

Of course, in the context of cuda, if you don't want 64 bit types propagating everywhere - you might instead choose to modify your code so that unification doesn't produce 64-bit types.

Widening unification solution

```
@jit(nopython=True)
def select(a, b, threshold, value):
    if threshold < value:
        r = a          # r: float32
    else:
        r = float32(b)  # r: float32
    return r           # r: {float32, float32} unifies to float32

a = np.float32(1)
b = np.int32(2)
select(a, b, 10, 11)  # Call with (float32, int32, int64, int64)
```

RAPIDS

17

In this case, you could cast the int value to float32.

Then, the unification set ends up being two float32s, and you don't get any float64s appearing.

Width of constants

```
@cuda.jit
def assign_constant(x):
    x[0] = 2.0

x = np.zeros(1, dtype=np.float32)
assign_constant[1, 1](x)
```

The constant value:

```
# Numba IR: $const2.0 = const(float, 2.0) :: float64
# LLVM IR:  store float 2.000000e+00, float* %arg.x.4, align 4
# PTX:      mov.u32 %r1, 1073741824;
```

RAPIDS

18

Another problem is that constants are 64-bit by default. We'll look at the typing of constants over the next few examples. We're going to build up more complex sequences starting with a wide constant, so we can see how the propagation affects typing, and how to get things under control.

In this first example, we're assigning a constant to an array of float32s. The second code block shows how the constant is represented at different points of the pipeline.

- After typing, the Numba IR has the constant as a float64 value.
- In the optimized LLVM IR, it's been reduced down to a 32-bit float
- And in the corresponding PTX, it's also typed as a 32-bit value

So in some cases it doesn't matter if your constants are too wide - The LLVM optimizer could see that a 64-bit constant was being assigned to a 32-bit location, and optimised the constant down to 32-bits. That had a knock-on effect in the PTX as well, in that it only needed 32 bits.

Widening arithmetic

```
@cuda.jit
def assign_constant(x):
    x[0] = 2.0
```

(Simplified) Numba IR:

```
# $8subscr.4 = getitem(x, 0)  :: float32
# $const10.5 = const(float, 2.0)  :: float64
# $12add.6 = inplace_binop(fn=<add>, lhs=$8subscr.4, rhs=$const10.5)
#           :: float64
```

LLVM / PTX:

```
# LLVM IR: %.97 = fadd float %.4853, 2.000000e+00
# PTX: add.f32 %f2, %f1, 0f40000000;
```

RAPIDS

19

Now, let's add a little more complexity to the example. We'll change it so that instead of assigning a constant `x[0]`, we're adding a constant to it. So we've now got an arithmetic operator involved.

In the Numba IR, it sees that we've got:

- A float32
- A float64
- and we're adding them together, which produces a float64.

In this case we still get lucky:

- The LLVM optimizer sees a 64-bit float add being stored into a 32-bit float, and does all the arithmetic with float32s.
- So the PTX only uses float32s as well.

So far, so good.

Widening propagation

```
@cuda.jit
def add_constant_2(x, y):
    x[0] += y[0] + 2.0
```

Numba IR:

```
# $8subscr.4 = getitem(x, 0)  :: float32
# $14subscr.7 = getitem(y,0)  :: float32
# $const16.8 = const(float, 2.0)  :: float64
# $18add.9 = $14subscr.7 + $const16.8  :: float64
# $20add.10 = inplace_binop(fn=<add>, lhs=$8subscr.4, rhs=$18add.9)
#                               :: float64
```

RAPIDS

20

Now if we go a bit further and add a second value as well, from the y array, then the typing gets a bit more complex, and a bit worse.

In the Numba IR we've got:

- An add of a float32 and a float64 which produces another float64,
- and another add of a float32 and a float64

That's the typing. I can't fit everything on one slide, so let's look at the LLVM IR on the next slide.

Widening propagation (LLVM)

Numba IR:

```
# $8subscr.4 = getitem(x, 0)  :: float32
# $14subscr.7 = getitem(y,0)  :: float32
# $const16.8 = const(float, 2.0)  :: float64
# $18add.9 = $14subscr.7 + $const16.8  :: float64
# $20add.10 = inplace_binop(fn=<add>, lhs=$8subscr.4, rhs=$18add.9)
#           :: float64
```

LLVM IR:

```
# %.110 = fpext float %.97 to double
# %.111 = fadd double %.110, 2.000000e+00
# %.121 = fpext float %.60 to double
# %.122 = fadd double %.111, %.121
# %.161 = fptrunc double %.122 to float
```

RAPIDS

21

The Numba IR here is what we just saw on the previous slide.

The LLVM IR this time, contains:

- some casts from floats to double
- the actual arithmetic operations on doubles, instead of floats
- then a cast back down to float for storing the result

So this time the LLVM optimizer couldn't save us:

- Instead of ending up with everything being float32, which we probably wanted,
- We've got float64 operations and some casts.

Widening propagation (PTX)

LLVM IR:

```
# %.110 = fpext float %.97 to double
# %.111 = fadd double %.110, 2.000000e+00
# %.121 = fpext float %.60 to double
# %.122 = fadd double %.111, %.121
# %.161 = fptrunc double %.122 to float
```

PTX:

```
# cvt.f64.f32    %fd1, %f2;
# add.f64        %fd2, %fd1, 0d4000000000000000;
# cvt.f64.f32    %fd3, %f1;
# add.f64        %fd4, %fd3, %fd2;
# cvt.rn.f32.f64 %f3, %fd4;
```

RAPIDS

22

That goes through to the PTX too.

- Here's the LLVM IR we just saw
- and the PTX that NVVM generates from it

We see pretty similar things :

- conversions from f32 to f64
- the constant represented as a 64-bit value
- and then a conversion back down to f32.

Preventing widening

Explicitly type constants:

```
@cuda.jit
def add_constant_2(x, y):
    x[0] += y[0] + float32(2.0)
```

This cuts off the source of the problem:

```
# $8subscr.4 = getitem(x, 0)  :: float32
# $14subscr.7 = static_getitem(y, 0)  :: float32
# $const18.9 = const(float, 2.0)  :: float64
# $20call_function.10 = call (func=float32, args=[Var($const18.9)]
#                        :: (float64,) -> float32
# $22add.11 = $14subscr.7 + $20call_function.10  :: float32
# $24add.12 = inplace_binop(fn=<add> lhs=$8subscr.4, rhs=$22add.11)
#                        :: float32
```

RAPIDS

23

So now we've seen what the problem is with constants, and how their size can propagate, let's look at the solution.

If you explicitly type the constant by casting it at the point where it's declared, then that usually helps.

If we do that in this case, we see in the typing that the constant itself is still a float64, but we have this cast function call that immediately reduces it to float32.

That's just enough to help the LLVM optimizer keep the width of everything down.

Preventing widening - PTX, reg use

PTX:

```
# add.f32 %f3, %f2, 0f40000000;  
# add.f32 %f4, %f1, %f3;
```

Register usage:

```
for defn, kernel in add_constant_2.definitions.items():  
    print(defn)  
    reg_usage = kernel._func.get().attrs.regs  
    print(f'Register usage: {reg_usage}')
```



```
# Widening propagation:      8  
# No widening propagation: 6
```

RAPIDS

24

I'm not going to go through all the LLVM again, but if we look at the PTX that we get, we see the two additions done with f32s, instead of conversions and f64s.

If you want to see the register usage then there's a way to get at that- it's a bit convoluted, but if you iterate over all the definitions of the kernel you can access the regs attribute of the definition (When I say “definition” here, that's one definition per typing).

For the two variants of the add_constant kernel, with and without the cast of the constant, we see that when we stop the widening propagation from happening, it uses fewer registers.

So, it's really worth paying attention to this if you're trying to maximise occupancy.

Limiting register usage

```
@cuda.jit
def add_constant_2(x, y, a):
    a = y[0]
    b = 2.0
    c = y[1] / 6
    d = y[2] % 8
    e = y[3] * y[4]
    for i in range(a):
        a += 2
        b -= c
        e *= d
        x[0] += a * b + c * d - e
```

36 Registers used

RAPIDS

25

Sometimes there's a limit to how much you can reduce register usage by tinkering with the kernel.

It can be better to ask the optimizer to reduce the number of registers it uses instead.

We'll use this example to talk about it - it is a bit convoluted, but it's the simplest example I could think of where it makes an impact.

The code on the slide uses 36 registers by default.

Limiting register usage (2)

```
@cuda.jit(max_registers=24)
def add_constant_2(x, y, a):
    # ...
```

A request, not an instruction:

```
# Without max_registers: 36
# With max_registers=24: 24
# With max_registers=20: 24
```

Enforcing register limits:

```
if kernel._func.get().attrs.regs > MAX_REGS:
    raise RuntimeError("Kernel uses too many registers")
```

RAPIDS

26

You can pass a keyword argument to the `cuda.jit` decorator to tell it how many registers you'd like the kernel to use.

If the kernel is going to exceed that register usage, it will put some temporary values in global memory instead. Although global memory is a lot slower than registers, the occupancy increase could be a tradeoff that gives a net positive for performance.

Here we can get the register use down to 24 for this kernel with the keyword argument.

You can pass in any number of registers, but the optimiser will only do what it can - if it can't reduce register usage past a certain point, then it won't, and you won't get any error.

For this example, if I asked for 24 registers, then that's what I got. However, if I asked for 20 registers, I still got 24.

You might want to ensure that your code doesn't go above a certain register count in future, for example if you're evolving the code over time but don't want performance regressions.

There's nothing explicitly in Numba to enforce that, but you could check the register usage and raise an exception if it's higher than you want, perhaps as part of a unit test.

Remarks

- ▶ Typing is the root cause of operand width issues
- ▶ Sometimes LLVM / NVVM optimisers help
- ▶ For problems with operand width / register usage:
 - ▶ Focus on typing: use `inspect_types()`
 - ▶ The LLVM IR and PTX are a consequence of the typing
 - ▶ Don't try to evaluate changes by looking primarily at LLVM IR / PTX
 - ▶ Use `max_registers` for another way to control usage

RAPIDS

27

We've just covered quite a lot of detail, so this is a little recap over what we've discussed.

The key point is that typing is always at the root of issues to do with operand size and register usage. Sometimes the typing doesn't matter because the optimizers can shrink things back down.

- When you do have a problem, always look at the typing when you're trying to improve things.
- If you tweak something, look at the typing to see what changed - don't just look at the LLVM IR or PTX, because it doesn't provide good feedback about whether you're going in the right direction.
- Once everything falls into place, the LLVM IR and PTX will suddenly improve, but you can't easily see the effect of incremental changes without looking at the Numba IR.

If you've gone as far as you can with code changes, you can try the `max_registers` keyword argument, and see if you can find a sweet spot for performance with it.

Integer arithmetic behaviour

- ▶ Strong preference for int64 over int8, int16, int32, etc.
- ▶ Rule: *“Start big and keep the width unchanged”*
- ▶ Old rule: *“Start small and grow as required”*
- ▶ Further details: [Numba Enhancement Proposal 1 - Changes in integer typing](#)

RAPIDS

28

The final topic for this session is specific to integer arithmetic. Numba's integer arithmetic rules are geared towards doing something predictable for the user, rather than being particularly good for CUDA, which is a bit unfortunate for us.

The general rule Numba uses when typing integer operations, is to strongly prefer int64s for everything.

In other words, it starts big and keep the width unchanged. This doesn't make too much performance difference on a CPU target, but isn't great for register usage on CUDA.

It used to be that Numba tried to keep everything small and grow operands as necessary, but a lot of non-CUDA users found that a bit unintuitive.

Back in 2015, a Numba Enhancement Proposal changed it to what it is now. It did even discuss the performance impact, but concluded that it would be negligible, which is true for a CPU-centric perspective.

If you want to look into the background and rationale a bit more, you can follow the link to the proposal.

Integer arithmetic example

```
@cuda.jit
def index_computation(x):
    i = cuda.grid(1)                # int32

    if i < x.shape[0]:              # x.shape[0] :: int64
        for j in range(3):          # range_iter_int64
            x[i, j] = (i * 2) + (j * 3) # int64 computations

x = np.zeros((1024, 3), dtype=np.int32)
index_computation[32, 32](x)

# Register usage: 12
```

RAPIDS

29

We'll have a quick look at one example. This code computes some values stored into array, based on arithmetic using the loop indices.

We see that a lot of this gets typed as int64:

- `i` is an int32,
- but `j` is an int64
- and the arithmetic operations all produce float64s.

The code as-is uses 12 registers, which is probably more than optimal.

Integer arithmetic “solution”

```
@cuda.jit
def index_computation(x):
    i = cuda.grid(1)                # int32

    if i < int32(x.shape[0]):        # int32 compare
        for j in range(int32(3)):    # range_iter_int32
            x[i, j] = int32(int32(i) * int32(2))
                               + int32(int32(j) * int32(3)))
                               # int64 / 32 mix

# Register usage: 14!
```

- ▶ Can be better not to attempt to optimize!
- ▶ Proper solution requires a change in Numba

RAPIDS

30

So what if we go all out and try and force everything to be int32?

It looks quite an ugly mess, and it doesn't really help. In the end we get even more registers being used - when I tried this, 14 ended up being needed.

I haven't shown the PTX here, but the problem is that extra registers are needed for having some 32-bit values and some 64-bit values.

In the end, it's probably better just to try not to optimise the width of integer operations unless you're really desperate to improve things.

What we really need is a change in the behaviour of Numba for the CUDA target where we can make it prefer 32- or fewer bits for integers. That's something that's on the to-do list!

Summary / Review

- ▶ Typing uses the propagation / unification mechanism
 - ▶ Deciphering errors
- ▶ Controlling operand width important in CUDA:
 - ▶ Unification widening
 - ▶ Constant widening
 - ▶ Integer operations
- ▶ Limiting register usage

RAPIDS

31

To summarise what we've covered:

- Numba's main mechanism for typing is the propagation and unification process
- That can fail for user errors, or use of unsupported functions
- So we've looked at some examples of deciphering the errors.

For CUDA performance, we really need to keep the typing's understanding of operand width under control:

- Unification can widen things
- Constants and propagation can widen things,
- and integer operations especially can cause widening.

We've also looked at the `max_registers` keyword for another way of controlling register usage.

Thankyou! / Questions?



RAPIDS

RAPIDS

Numba for CUDA Programmers
Session 3 - Porting, performance,
interoperability, debugging

Graham Markall - gmarkall@nvidia.com

Review from last week

- ▶ Numba: Python JIT Compiler that targets CPUs and GPUs
- ▶ Typing / type inference:
 - ▶ Understanding errors
 - ▶ Understanding performance

A quick reminder of last week:

Pretty much all of last week was about Numba's typing and type inference.

It was a pretty heavy and in-depth topic, mostly focusing on understanding errors and getting better performance out of Numba.

This week

- ▶ Porting strategies
- ▶ Performance measurement
- ▶ Interoperability with other CUDA Python libraries
- ▶ Debugging

RAPIDS

3

This week is a little bit easier going. It's a bit of a mixed bag of various things that I think are useful when you're using Numba to port a Python code to CUDA.

I'll go over my approach to using Numba on a new codebase, going from a pure CPU-based Python code to running on a GPU with good performance.

I've got a little bit to say about performance measurement as well - a few things that can make it easier to understand what's happening.

Also we'll look at how to interoperate Numba code with other Python CUDA libraries, and how to debug things along the way.

Porting

First of all we'll look at porting.

General porting strategy

1. Profile code, identify hotspots
2. Compile hotspots with `@jit` for CPU
 - ▶ Possibly in *Object Mode*
3. Convert Object Mode code to Nopython mode
4. Identify functions to move to GPU
 - ▶ Large arrays/NumPy, lots of data parallelism, etc.
 - ▶ Closely-related functions to avoid data movement
5. Move functions to GPU
6. Manage data movement

RAPIDS

5

This slide outlines the steps of my approach. Over the next few, we'll get into the detail about these steps.

Maybe it seems a little obvious, but the first thing you want to do is know where the hotspots in the code are so you know what to apply Numba to. Usually profiling is a good way of finding that out, and I've got a little more to say on that later.

Once you've found the hotspots, the lowest friction thing to do is to compile them for the CPU with the `@jit` decorator. When you first do this you might find that Numba falls back to object mode, which is OK for a first pass.

Then when you've got everything you want compiling with Numba, start working through and fixing up all the object mode code so it compiles in nopython mode.

The CUDA target has no object mode, only nopython mode, so once you've done that you're ready to start identifying which functions you want to move to the GPU. You don't have to move absolutely everything over to the GPU - you can just move the ones that make the most sense to move there. These will be the ones that have lots of data parallelism, and lots of computational intensity, and maybe also some related functions so that you can keep the work on the GPU as much as possible without transferring data back-and-forth.

Once you've moved functions over to the GPU, you can look at optimising data movement and data management on the device a bit better.

Porting red flags

- ▶ Mainly serious CPU optimization attempts:
 - ▶ Heavy use of Cython
 - ▶ Python C extensions to native libraries
- ▶ Heavy use of unsupported constructs:
 - ▶ Classes, comprehensions, unsupported libraries
- ▶ Insufficient testing:
 - ▶ End-to-end tests, few unit tests in particular
- ▶ NumPy array operations:
 - ▶ Use CuPy where possible instead

RAPIDS

6

Before we look at some of those steps in detail, I want to highlight a few red flags for porting a Python code to use Numba, particularly for GPUs. These are mostly things that I've bumped up against in the past, that have made it difficult to implement a code on the GPU, or needed a lot of work to untangle and rewrite the implementation.

Usually they come from a lot of work having gone into optimizing code to run on the CPU. If there's existing native code through Cython or using Python extensions and C or C++, then you have to do a lot of work to get that implementation back into Python before you can really use Numba on it.

Even if the code is all Python, it might make use of a lot of constructs that aren't well supported by Numba. If you see a lot of classes, and object orientation, then that'll take a fair bit of rewriting to get something that can be compiled effectively.

Other dynamic features, and use of libraries or functions that Numba doesn't have support for compiling can also need a lot of work to get to a good starting point.

Another problem is when the test coverage isn't very good. I find that a lack of unit testing is particularly a problem with Python, more so than in a language like C or C++ because everything's so much more loose with typing.

If you've only got a few end-to-end tests, that's OK if you're working with pure Python because it's fairly easy to debug and explore. But for porting to Numba and CUDA, not having effective unit tests is a bit of a headache because it can be very hard to tell where things are going wrong.

Also, code that uses lots of NumPy array operations can be a bit problematic for Numba CUDA because a lot of NumPy array operations aren't supported by the CUDA target. In these cases, you'll have to re-implement things element-by-element in your kernels. Hopefully, if you're lucky you can use CuPy for a lot of these things instead.

CPU target Python support

In addition to CUDA Python-supported features

▶ Language:

- ▶ yield, try / except, with
- ▶ List comprehension
- ▶ Passing functions as arguments

▶ Types / libraries:

- ▶ *Typed List*, *Typed Dict*
- ▶ ctypes, cffi, random,
- ▶ Many more Numpy functions / array operations

RAPIDS

7

Another thing to bear in mind when you're porting is that the CPU target supports a lot more Python language features, types, and libraries than the

CUDA target. In a way that's a blessing early on because it makes it a lot easier for you to get started compiling lots of the code with Numba for the CPU.

But then when you get to moving over to the GPU you're going to need to reorganise the code to avoid using these features.

The major things you get on the CPU but not CUDA are some language features like generators, comprehensions, and being able to pass functions around, as well as better support for standard library types and modules, and a lot of the commonly-used Numpy array operations and functions.

None of these things are insurmountable, but you can feel like you've made a lot of progress early on using Numba, then find you've still got lots of work to do to get from the CPU target to the CUDA target.

Object mode

- ▶ Compiles code with unsupported types / libraries
- ▶ Generated code calls into Python interpreter C API
- ▶ Example:

```
@jit
def hash_computation(x):
    # Loop and body supported by Nopython mode
    for i in range(len(x)):
        x[i] = x[i] ** 2
    # Unsupported library / function call
    return hashlib.md5(x).digest()
```

RAPIDS

8

I mentioned earlier that when you first start decorating everything for the CPU with the @jit decorator, it might use object mode. What is that, and what's it for?

The idea is to allow Numba to compile code even if it's using some types or libraries that it doesn't support. It works by emitting code that calls back into the Python interpreter to deal with the unsupported bits. The tradeoff it makes is that the code can be compiled, but its execution could still be quite slow.

There's an example on the slide of code that'll force Numba to fall back to object mode. The hashlib library isn't presently supported by Numba, so that's going to need a call to the Python interpreter's C API to handle that line.

Loop lifting

- ▶ Loops supported by nopython mode outlined
- ▶ Potentially identifies how you should eventually refactor code to run bits on CUDA.

```
@jit(nopython=True)
def outline_1(x):
    for i in range(len(x)):
        x[i] = x[i] ** 2

@jit
def hash_computation(x):
    outline_1(x)
    # Unsupported library / function call
    return hashlib.md5(x).digest()
```

RAPIDS

9

One thing that Numba does when compiling in object mode to try to reduce its performance penalty is to identify loops that could be compiled in nopython mode, and outline them.

Then, it compiles the outlined loop in nopython mode, and inserts a call to it in the object mode function.

So in our previous example, with loop lifting, it's as if the code were written as shown on the slide, with the loop in the outlined function.

Loop lifting inspection

```
object_mode.py:12: NumbaWarning:
Compilation is falling back to object mode WITH looplifting enabled
because Function "hash_computation" failed type inference due to:
  Unknown attribute 'md5' of type Module(<module 'hashlib'...>)
```

numba --annotate-html output:

Function name: hash_computation
in file: object_mode.py
with signature: (array(int64, 1d, C),) -> pyobject

```
12: @jit
13: def hash_computation(x):
▶ 14:     for i in range(len(x)):
▶ 15:         x[i] = x[i] ** 2
▶ 16:     return hashlib.md5(x).digest()
```

RAPIDS

10

How do you know about it when object mode fallback happens, and when loop lifting happens?

Numba tells you with a warning like the one shown. It'll also tell you at least one of the reasons why it's doing that - in this case it doesn't know how to compile `hashlib.md5`.

Another way to get a better look at what's happening is to use the command line tool called `numba`. It has the `annotate-html` option that you can use to produce an HTML output showing the typing of the function.

It highlights lifted loops in green, and uses red for code that had to be compiled in object mode.

For our example the loop is nice and green, and the `hashlib` call is highlighted red.

Expanding view of types

Function name: hash computation
in file: object_mode.py
with signature: (array(int64, 1d, C),) -> pyobject

```
12: @jit
13: def hash_computation(x):
14:     for i in range(len(x)):
        label 13
        x = arg(0, name=x) :: array(int64, 1d, C)
        jump 14
        label 14
        $2load_global.0 = global(range: <class 'range'>) :: Function(<class 'range'>)
        $4load_global.1 = global(len: <built-in function len>) :: Function(<built-in function len>)
        $8call_function.3 = call $4load_global.1(x, func=$4load_global.1, args=[Var(x, object_mode.py:14)], kws=(),
        vararg=None) :: (array(int64, 1d, C),) -> int64
        del $4load_global.1
        $10call_function.4 = call $2load_global.0($8call_function.3, func=$2load_global.0, args=[Var($8call_function.3,
        object_mode.py:14)], kws=(), vararg=None) :: (int64,) -> range_state_int64
        del $8call_function.3
        del $2load_global.0
        $12get_iter.5 = getiter(value=$10call_function.4) :: range_iter_int64
        del $10call_function.4
        $phi14.0 = $12get_iter.5 :: range_iter_int64
        del $12get_iter.5
        jump 16
        label 16
        $14for_iter.1 = iternext(value=$phi14.0) :: pair(int64, bool)
        $14for_iter.2 = pair_first(value=$14for_iter.1) :: int64
        $14for_iter.3 = pair_second(value=$14for_iter.1) :: bool
        del $14for_iter.1
        $phi16.1 = $14for_iter.2 :: int64
        del $14for_iter.2
        branch $14for_iter.3, 36, 37
        label 36
        del $14for_iter.3
        i = $phi16.1 :: int64
        del $phi16.1
        label 37
        del x
        del $phi16.1
        del $phi14.0
        del $14for_iter.3
        $27 = build_tuple(items=[]) :: Tuple()
        return $27
```

RAPIDS 11

On each line there's also a little triangle you can click to expand the view.

When you do that, it shows you Numba's typing for the line of code - this is pretty much the same as what you would see from the `inspect_types` function.

For the lifted loop, the types are all things Numba knows about - like arrays, int64, bool, etc.

PyObject typing

```

16: return hashlib.md5(x).digest()
    label 37
    jump 38
    label 38
    $36load_global.0 = global(hashlib: <module 'hashlib' from '/home/gmarkall/miniconda3/envs/numba/lib/python3.8
    /hashlib.py'>) :: <missing>
    $38load_method.1 = getattr(value=$36load_global.0, attr=md5) :: <missing>
    del $36load_global.0
    $42call_method.3 = call $38load_method.1(x, func=$38load_method.1, args=[Var(x, object_mode.py:14)], kws=(),
    vararg=None) :: pyobject
    del x
    del $38load_method.1
    $44load_method.4 = getattr(value=$42call_method.3, attr=digest) :: <missing>
    del $42call_method.3
    $46call_method.5 = call $44load_method.4(func=$44load_method.4, args=[], kws=(), vararg=None) :: pyobject
    del $44load_method.4
    $48return_value.6 = cast(value=$46call_method.5) :: <missing>
    del $46call_method.5
    return $48return_value.6

```

RAPIDS

12

Now let's look at the expanded hashlib call - it gives us a bit more detail about exactly what the problem was.

We can see the IR that's given the pyobject type - if we can do something to change the code so that there's no more pyobject types, then it should be possible to compile in nopython mode.

In practice, that might mean writing your own python implementation of these functions that can be compiled in nopython mode, or finding another implementation or library that can be compiled in nopython mode, maybe in conjunction with making some algorithmic changes.

Sometimes you won't easily be able to get rid of these - in that case, you have to hope it's not too much of a performance hotspot. The problem code might well have been something that would have been hard to do well with on a GPU anyway.

Moving to Nopython mode

- ▶ Start converting `@jit` to `@jit(nopython=True)` (or `@njit`)
- ▶ Rewrite / avoid unsupported functions
 - ▶ E.g. `hashlib.md5` from previous example
- ▶ If hotspots in lifted loops, manually outline
 - ▶ In preparation for moving them to the GPU
- ▶ Start with hotspots
 - ▶ May not be necessary to convert all functions

RAPIDS

13

Once you've got things going with object mode, you want to start moving towards making everything nopython mode so the code's in a form that's going to be compilable on the CUDA target.

This mostly consists of systematically trying to do the things I've just mentioned across the codebase - generally to try and get rid of unsupported functions and libraries somehow. You might want to manually outline the nopython mode loops so you can turn them into cuda-jitted functions too.

You probably want to focus on where the code spends most of its time executing to get the best return on your efforts.

Moving from Nopython to CUDA

- ▶ Replace @njit with @cuda.jit, add launch parameters
- ▶ Distribute loops across threads

```
@jit
def f(x):
    for i in range(len(x)):
        do_something(x)
```

Becomes:

```
@cuda.jit
def f(x):
    for i in range(cuda.grid(1), len(x), cuda.gridsize(1)):
        do_something(x)
```

RAPIDS

14

Once you've got an acceptable amount of code in nopython mode, you can start to move it over to the GPU.

You need to replace your njit decorators with cuda.jit decorators, and at the call sites add launch configurations.

For simplicity, I might start with just one thread and one block to get things compiling and running on a GPU. It'll be quite slow but at least you can validate that things are working correctly.

Nextm you can go on to distribute the work across threads. For loops in your kernel functions will need to be transformed so that threads do the work together. The example on the slide makes a sequential loop into a thread strided loop.

The parallelisation strategies at this stage are pretty much the same for going to CUDA Python as they are for going to CUDA C++ from a plain C or C++ code.

Rewriting NumPy array operations

- ▶ `np.sum`, `np.mean`, etc. not supported in kernels
- ▶ Rewrite array ops, or use CuPy
 - ▶ In-kernel: thread-private data / small sub-arrays
 - ▶ CuPy: large data, shared (e.g. mean of all elements)

```
@jit
def normalize(x):
    for i in range(x.shape[0]):
        x[i, :] = x[i, :] / x[i, :].mean()
```

RAPIDS

15

One thing you will probably bump into is the use of NumPy array operations and functions.

Lots of numpy functions like `sum`, and `mean`, and other things that operate on a whole array and not just individually elementwise won't work on the CUDA target.

When you've got operations like this, there's two options for getting around them:

- One is to modify code in your kernel so that you manually implement these operations using sequential loops, or with some cooperation across threads,
- and the other is to use CuPy for them.

Rewriting them yourself is likely to work best with operations on small chunks of data, especially where it makes sense for them to be thread-private.

Using CuPy can make more sense when the operations are on much larger data, and where the results are shared between all threads.

There's an example on the slide where we'd bump into a couple of instances of this problem.

- First, it uses the `mean` function to compute the mean of rows of a matrix, which isn't supported
- Secondly, it uses an array operation to normalise each row using its mean.

We'll look at both ways of dealing with both of these problems.

Rewrite in-kernel

```
@jit
def normalize(x):
    for i in range(x.shape[0]):
        x[i, :] = x[i, :] / x[i, :].mean()
```

```
@cuda.jit
def normalize(x):
    i = cuda.grid(1)

    # Compute mean manually
    mean = float32(0.0)
    for j in range(x.shape[1]):
        mean += x[i, j]
    mean /= x.shape[1]

    # Array op division rewritten as a loop
    for j in range(x.shape[1]):
        x[i, j] = x[i, j] / mean
```

RAPIDS

16

First of all, if we decide to use a cuda kernel and implement the operation ourselves, this is what it would look like.

On top is the original CPU-jitted function, and on the bottom is after transforming it to a CUDA kernel.

First we separately compute the mean for each column, with one thread assigned to each column. Then we rewrite the array operation doing the divide using a loop.

Now this code will compile and run on the CUDA target.

Rewrite using CuPy

```
@jit
def normalize(x):
    for i in range(x.shape[0]):
        x[i, :] = x[i, :] / x[i, :].mean()
```

```
x_cp = cp.asarray(x)
x_cp = x_cp / x_cp.mean(axis=1).reshape((x_cp.shape[0], 1))
```

Notes:

- ▶ No kernel call in this case
- ▶ Can't call CuPy functions from `@cuda.jit` kernels
- ▶ Can pass CuPy arrays to Numba kernels

RAPIDS

17

Now let's look at the other way of doing it.

If we're going to use CuPy and we've got a Numba or NumPy array, we first need to convert it to a CuPy array. This isn't a performance problem - Numba and CuPy arrays can share data, and we'll look at how that works a bit later on.

With a little bit of fiddling to implement the same thing as the for loop with one array operation, we end up with the code in the second block, which gives the exact same result as the original code and the Numba kernel.

There's a couple of things to note:

- Here we're not doing these operations inside a Numba kernel - these are all calls to CuPy functionality from the Python interpreter.
- In fact it's not even possible to call CuPy functions from Numba kernels.
- However, you can pass CuPy arrays to Numba kernels if you want to.

So, we've seen the two ways of dealing with NumPy array operations for the CUDA target.

Manage data movement

- ▶ Using `to_device`, `device_array_like`, etc.
- ▶ Use streams for async:
 - ▶ `cuda.stream()` - construct a new stream
 - ▶ `cuda.default_stream()` - get the default stream
 - ▶ `cuda.external_stream(ptr)` - wrap external stream

```
# Create a new stream
stream = cuda.stream()
# Create a pinned array on the host for async transfers
a = cuda.pinned_array(n, dtype=np.int32)
# Create an array with a "default stream"
d_a = cuda.device_array_like(a, stream)
# Numba automatically uses async transfers when streams involved
d_a.copy_to_device(a, stream=stream)
# Kernel launch on stream
kernel[nblocks, nthreads, stream]
```

RAPIDS

18

The last step in my porting approach is to start to think about data movement. Partly this is just about doing the same things we already covered in the first session, like using the device array constructors and methods to put data on the device, and only copy data when necessary.

One additional thing to mention here was that if you want to overlap communication and compute then you can do that with Numba - it supports streams and you can use them for asynchronous operations.

There's a couple of different ways of making stream objects:

- You can use `cuda.stream()` to get a new non-default stream,
- or `default_stream()` to get the default stream
- You can also use a stream from elsewhere, maybe another Python or another language's library - as long as you've got a pointer to the stream object, you can call `external_stream()` to get back an object that Numba understands that wraps that stream.

You can use Numba stream objects to construct arrays that have a default stream - that is, any transfer operations on them use that stream by default

You can also specify a stream when you do a transfer. Whenever you do specify a stream for one of these operations, Numba automatically uses the async versions of the underlying functions in the driver API.

There's a short example of creating and using a stream on the slide, then transferring asynchronously from pinned memory and launching a kernel on the same stream.

CUDA target useful tools

- ▶ **Random Numbers:** `cuda.random`
 - ▶ xoroshiro128+ algorithm c.f. `cuRAND XORWOW` / `MTGP32` / `Philox`
 - ▶ Uniform and normal distributions
 - ▶ `numpy.random`: many other distributions
- ▶ **Reductions:**
 - ▶ `@reduce` decorator
- ▶ **Foralls:**
 - ▶ Call a kernel on every element of array
 - ▶ Configuration, occupancy, etc., worked out by Numba
 - ▶ `forall()` method of kernels

RAPIDS

19

There are a few other bits and pieces in Numba that can be useful when you're moving to CUDA. We'll go over them and have a look at an example of each.

One is the random number generator - Numba doesn't use `cuRAND` because of some issue with integrating it, but it does include its own random number generator.

It's not a secure random number generator, but it's good enough for things like Monte Carlo simulations. It uses a different algorithm to `cuRAND`. You can use it to get numbers from normal and uniform distributions.

If the Python you're porting uses the `numpy.random` module, that's got a lot of other distributions, so you might need to take that into account as well.

Another useful tool is the reduce decorator. You can use it to generate reduction kernels by writing the reduction operation in terms of a single pair of elements. The decorator then it makes a kernel that applies the reduction to an array - in a sense it's a bit like vectorize and `guvectorize` in the way that it accepts a function for one operation, and the application of that function is taken care of for the user.

Finally, there's the `forall` method of kernels. This gives you back a function that calls the kernel applied to every element of an array without you having to think about the kernel configuration or occupancy.

Random generator usage

```
@cuda.jit
def compute_pi(rng_states, iterations, out):
    """Find the maximum value in values and store in result[0]"""
    tid = 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, tid)
        y = xoroshiro128p_uniform_float32(rng_states, tid)
        if x**2 + y**2 <= 1.0:
            inside += 1

    out[tid] = 4.0 * inside / iterations

state_size = nblocks * nthreads
rng_states = create_xoroshiro128p_states(state_size, seed=1)
compute_pi[nblocks, nthreads](rng_states, 10000, out)
```

RAPIDS 20

This example shows how to use the random number generator. The code computes the value of pi. Without going into the mathematical details, let's focus on how it uses the RNG.

In the kernel to draw a number from the distribution, you can call `xoroshiro128p_uniform_float32` to get a float32.

There's a few other functions that follow this naming convention. If you want a float64 you can get that instead with the float64 variant, and if you want the normal distribution you can change uniform to normal.

So that's how to get random numbers in the kernel. Before you call the kernel, you need to initialize the state first. You can do that with `create_xoroshiro128p_states`:

- The size of the state that you pass in to initialize needs to have a size equal to the total number of threads.
- You should also pass in an appropriate seed as well.

Then, when you launch the kernel, you can pass in your random states.

Reduction generator usage

```
@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (np.arange(1234, dtype=np.float64)) + 1

# Transfers result to host
sum_reduce(A)

# Keep result on device
r = cuda.device_array(1, dtype=np.float64)
sum_reduce(A, res=r)

# Use device array, keep result on device
A_d = cuda.to_device(A)
sum_reduce(A_d, res=r)
```

RAPIDS

21

This is a simple example of the reduction generator. This sums all the elements in the array.

We've applied the `cuda.reduce` decorator, so now we can call the `sum_reduce` function to reduce an array.

In the first call, the result of the reduction gets transferred back to the host. That might be useful in some cases, like if its going to be part of the termination condition for a loop, but mostly you probably want to keep the result on the device.

So, in the second call we can do that by creating a place for the result to be stored, and passing it in through the `res` keyword argument.

As with normal kernels, we can avoid any implicit data transfer by ensuring that the input data is already on the device. The third call demonstrates this with input and the output on the device.

ForAll usage

(Example borrowed from cuDF)

```
@cuda.jit
def gpu_round(in_col, out_col, decimal):
    i = cuda.grid(1)
    f = 10 ** decimal

    if i < in_col.size:
        ret = in_col[i] * f
        ret = rint(ret)
        tmp = ret / f
        out_col[i] = tmp

# in_data and out_data are device arrays
nelems = len(in_data)
# Round all elements to 3 d.p.
gpu_round.forall(nelems)(in_data, out_data, 3)
```

RAPIDS

22

This is an example of forall, borrowed from the cuDF source.

gpu_round is a kernel that rounds every element of an array to a given number of decimal places. It's a normal kernel, and we could launch it on an array. If we did that, we'd have to work out sensible grid dimensions for the launch.

Here we're saving the hassle of doing that by calling gpu_round's forall method, and telling it how many elements are in the array. The function it returns can be called on the data without worrying about the launch configuration.

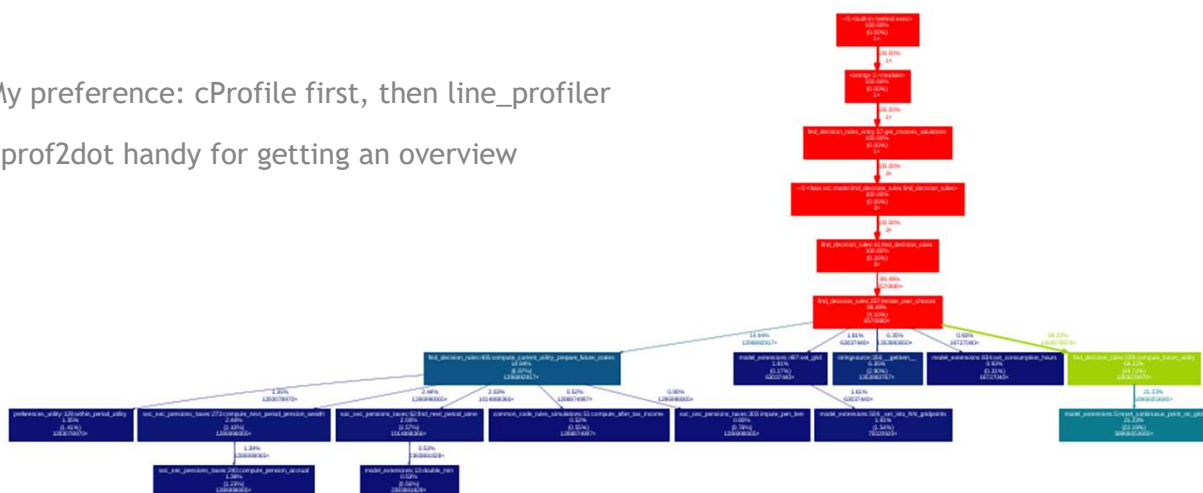
Internally, Numba uses the occupancy calculation of the kernel and the number of elements to work out a good launch configuration.

Measuring Performance

This next section covers some helpful points on measuring performance.

Profiling

- ▶ My preference: cProfile first, then line_profiler
- ▶ gprof2dot handy for getting an overview



RAPIDS 24

This is my approach to profiling - I might be a little out of date with the latest and greatest Python performance measuring tools, but I find that using cProfile to get an overview of the code usually works quite well and is quite easy to do.

Then if I want to see how much time is spent on individual lines for any reason, then line_profiler works pretty well for doing that, but you have to annotate which functions you want to profile. So line_profiler can be a bit cumbersome, and it doesn't work with numba-compiled code.

Another tool I quite like is gprof2dot, which is quite good for taking the profile from cProfile and turning it into a nice graph so I can see an overview of the hotspots.

Timing - general points

- ▶ First call to a jitted function requires compilation
- ▶ Time and report subsequent executions
- ▶ Notes on `@cuda.jit`-ed functions:
 - ▶ Make sure data is already on the device, or transfer will be counted
 - ▶ Run multiple times and calculate average time to amortize overheads
 - ▶ Can use events
 - ▶ Can use NSight Systems, NSight Compute, etc.

RAPIDS

25

The next couple of slides are about timing code compiled with Numba.

You should keep in mind when you're writing code to measure performance that the first call to a jitted function for a given set of argument types includes the compilation time, which can be quite lengthy. So, it's better to time and report subsequent executions.

Another thing to keep in mind is to put the data on the device before you're timing it, so that you don't time the copy and all the overhead that that entails.

I usually prefer to run the kernel in a loop many times and then synchronize, then work out the time of a run from the total time to try and get a clearer picture of the time taken in a representative situation.

You can also record CUDA events for timing, if you prefer to do that - it's particularly helpful if you're doing anything asynchronously.

And, Nsight compute and Nsight systems work on Numba compiled code, so if they're your preferred tools for investigating performance then you can use them.

Timing - caching example

```
@jit(nopython=True)
def go_fast(a):
    # ...

# Compilation time included in the first execution
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (with compilation) = %sms" % ((end - start) * 1000))

# Re-time executing from cache to get execution time only
start = time.time()
go_fast(x)
end = time.time()
print("Elapsed (after compilation) = %sms" % ((end - start) * 1000))
```

```
Elapsed (with compilation) = 214.68210220336914ms
Elapsed (after compilation) = 0.005245208740234375ms
```

RAPIDS 26

Here's a quick example showing how compilation and caching affects the measured time.

We have a jitted function - on the first call that we time, the compilation occurs. Then on the second call, the cached version is executed.

When we run this, we get the output shown. The first call seems very expensive - orders of magnitude slower than the second call.

So, always make sure your kernels are compiled before you time them.

Timing - CUDA events example

```
# Create event handles
start = cuda.event()
stop = cuda.event()

# Stage 1: Asynchronously issue work
start_time = time.time()
start.record(stream=stream)
d_a.copy_to_device(a, stream=stream)
increment_kernel[nblocks, nthreads, stream](d_a, value)
d_a.copy_to_host(a)
stop.record(stream=stream)
stop_time = time.time()

# Stage 2: Have CPU do some work while waiting for stage 1 to finish
counter = 0
while not stop.query():
    counter += 1
```

RAPIDS

27

Here's an example of timing using cuda events, in conjunction with asynchronous work on a stream. We're recording time on the CPU using the time function as well as recording the events to highlight the difference.

- First we create a couple of events,
- Then we'll record the start event,
- Fire off our work on a stream,
- Record the stop event,
- and the end time,
- then we can loop on the CPU until the stop event is reached.

(the example continues on the next slide)

Timing - CUDA events example (2)

```
gpu_time = start.elapsed_time(stop)
cpu_time = (stop_time - start_time) * 1000

print("time spent executing by the GPU: %.2fms" % gpu_time)
print("time spent by CPU in CUDA calls: %.2fms" % cpu_time)
```

```
time spent executing by the GPU: 10.89ms
time spent by CPU in CUDA calls: 0.46ms
```

RAPIDS

28

So now we've done everything and hit the stop event, we can work out how much time the CPU and GPU spent.

We can see the CPU time was very short (0.46ms), because all the async calls returned pretty much immediately, and we have the true time spent executing on the GPU from the events (10.89ms).

Interoperability

It's rare that an entire application would be implemented just using Numba - you'll probably want to draw on some other libraries too.

The next section is about interoperability of Numba with other Python libraries that use CUDA.

CUDA Array Interface

- ▶ A standard for different libraries to exchange / use each others' on-device data.
- ▶ Objects implement `__cuda_array_interface__`. Returns a dict:
 - ▶ pointer, shape, strides, etc.
- ▶ Implemented by:
 - ▶ Numba, CuPy, PyTorch, PyArrow, mpi4py, ArrayViews, JAX
 - ▶ RAPIDS: cuDF, cuML, cuSignal, RMM

RAPIDS

30

The CUDA array interface is a standard protocol that libraries can follow to exchange their CUDA data with each other without copying the data on the device, maintaining an understanding of what the underlying data represents.

Each library that implements it provides a special property in its objects, called `__cuda_array_interface__`. When that property is accessed, it gives back a dict with a pointer to the data, and its shape and strides, and some other properties.

The libraries that I know of that implement it are all listed on the slide - some general numerical libraries, some machine learning and AI libraries, and the RAPIDS libraries.

So you can use any of these libraries with Numba and pass their objects straight into Numba kernels and it should just do the right thing.

You can also pass Numba device arrays to the functions of these libraries (in most cases).

Example - Numba on CuPy data

Calling a Numba Kernel on a CuPy array:

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

add[1, 32](a, b, out)
```

RAPIDS

31

We'll look at a couple of examples with CuPy.

Here we've got a Numba kernel and we create a CuPy array.

Now we can pass that array straight into our Numba kernel as if it were a Numba device array.

Example - CuPy on Numba data

Zero-copy conversion using CUDA Array Interface:

```
import numpy
import numba
import cupy

# type: numpy.ndarray
x = numpy.arange(10)

# type: numba.cuda.cudadrv.devicearray.DeviceNDArray
x_numba = numba.cuda.to_device(x)

# type: cupy.ndarray - a view of x_numba's data
x_cupy = cupy.asarray(x_numba)
```

RAPIDS

32

Let's look at an example going the other way. Supposing you've got a Numba device array, and you want to be able to call CuPy array operations on it.

Here we create an array on the host, then move it onto the device.

Then we use cupy's `asarray` function to convert our device array to a CuPy array. There's no new allocation here - the CuPy array is just viewing the data in the Numba device array.

So, if we were to call CuPy functions on `x_cupy` that update it, then we'd see the results reflected in both the `x_numba` and `x_cupy` objects

CUDA Array Interface implementation

► Implement object with `__cuda_array_interface__`.

► Simple example for contiguous read-write array:

```
class MyArray:
    def __init__(self, shape, typestr, data):
        self._shape = shape
        self._data = data
        self._typestr = typestr

    @property
    def __cuda_array_interface__(self):
        return {
            'shape': self._shape
            'typestr': self._typestr
            'data': (self._data, False)
            'version': 2,
        }
```

RAPIDS

33

Now, what if we want to interoperate with a library that doesn't provide the CUDA Array Interface?

We can get around that by providing our own wrapper that implements it. This can wrap a device pointer from anywhere - it doesn't have to be from a Python library, as long as you've got the pointer to the data and some knowledge about its type and shape you can use this wrapper.

On the slide is the simplest example of a wrapper object. It's sufficient for a contiguous array that's readable and writable.

We need to initialize the class with the shape and type, and the pointer to the data. The class defines the `__cuda_array_interface__` property that returns them correctly populated in the dict.

Using our CUDA Array Interface wrapper

```
# Use ctypes to get the cudaMalloc function from Python
cudart = CDLL('libcudart.so')
cudaMalloc = cudart.cudaMalloc
cudaMalloc.argtypes = [POINTER(c_void_p), c_size_t]

# Allocate some Numba-external memory with cudaMalloc
ptr = c_void_p()
float32_size = 4
nelems = 32
alloc_size = float32_size * nelems
cudaMalloc(byref(ptr), alloc_size)

# Wrap our memory in a CUDA Array Interface object
arr = MyArray(nelems, 'f4', ptr.value)
```

RAPIDS

34

Now let's have a look at how we can use the wrapper. In this example, we've got a pointer to a device array that came from `cudaMalloc`, so it's outside the Numba and Python ecosystem.

Don't worry too much about the mechanism by which we do that - the main thing is that after we call `cudaMalloc` here, `ptr` contains a valid device pointer.

Now assuming we think this is a pointer to float data, we can then create a `MyArray` object with:

- the number of elements,
- the type string, which is a NumPy type string,
- and the data pointer.

(the example continues on the next slide)

Using our CUDA Array Interface wrapper (2)

```
# Wrap our memory in a CUDA Array Interface object
arr = MyArray(nelems, 'f4', ptr.value)

@cuda.jit
def initialize(x):
    i = cuda.grid(1)
    if i < len(x):
        x[i] = 3.14

initialize[1, 32](arr)
```

Having created our wrapper around the cudaMalloc'd device pointer, we can then pass the wrapper straight into a Numba kernel, and it'll work as expected and use the data.

Debugging

This section contains a few pointers on different ways of debugging CUDA jitted functions.

Generating debug info

- ▶ Pass `debug=True` to the `@cuda.jit` decorator
- ▶ Add checks for raised exceptions
- ▶ Adds line number information
- ▶ Helps interpret `cuda-memcheck` output
- ▶ Not much change to `gdb` / `cuda-gdb` usage

RAPIDS

37

One thing you can do to make it easier to debug a kernel is to pass the `debug=True` keyword argument.

This does two things:

- One is that if something raises an exception in your kernel, then the exception will be reported. That also includes assertions, so when you pass `debug=True` you can also use the `assert` keyword in kernels to catch bad states.
- The other is that it adds line number information to the generated code. Having line number information makes it a little bit easier using `gdb` or `cuda-gdb` to debug the code, and it can make it a little bit easier to see where a problem is with `cuda-memcheck`.

There's some examples of these things over the next few slides.

Off-by-one error example

```
@cuda.jit(debug=True)
def reciprocal(x):
    i = cuda.grid(1)
    # Off-by-one - accesses beyond end of array
    if i > x.shape[0]:
        return
    x[i] = 1 / x[i]

x = np.zeros(10)
reciprocal[1, 32](x)
```

- ▶ Without debug=True: wrong answer, no exception
- ▶ With debug=True: exception raised

In this first example we go slightly beyond the end of an array.

The condition in the if statement is incorrect, and we get one rogue thread accessing just past the last element.

If we run without debug=True, then the kernel will just run and there won't be any error come up, but we could end up corrupting some other item in memory.

When we compile with debug=True, we should see an exception raised.

Division by zero exception

```
Traceback (most recent call last):
  File "blackscholes_cuda.py", line 126, in <module>
    main(*sys.argv[1:])
  File "blackscholes_cuda.py", line 104, in main
    black_scholes_cuda[griddim, blockdim, stream](
  File ".../numba/cuda/compiler.py", line 817, in __call__
    cfg(*args)
  File ".../numba/cuda/compiler.py", line 576, in __call__
    self._kernel_call(args=args,
  File ".../numba/cuda/compiler.py", line 688, in _kernel_call
    raise exccls(*exc_args)

ZeroDivisionError: tid=[0, 0, 256] ctaid=[0, 0, 3906]:
    division by zero
```

So if we run this kernel, then we'll now get a traceback like this.

The ZeroDivision error gets raised, and it even tells us which thread ID and block ID caused it - this can help pinpoint the source of the issue.

Off-by-one error: cuda-memcheck

```
@cuda.jit(debug=True)
def add_1(x):
    i = cuda.grid(1)
    # Off-by-one - accesses beyond end of array
    if i > x.shape[0]:
        return
    x[i] += 1

x = np.zeros(10)
add_1[1, 32](x)
```

RAPIDS

40

Next we'll look at the use of cuda-memcheck. We'll use some similar code to that we've just seen, this time without a divide by zero error.

It does still have an invalid read and write outside the bounds of the array.

cuda-memcheck output

► Without debug=True:

```
Invalid __global__ read of size 8
  at 0x000000f0 in cudapy::__main__::add_1$241(
    Array<double, int=1, C, mutable, aligned>)
  by thread (10,0,0) in block (0,0,0)
  Address 0x7f4a37800050 is out of bounds
  Device Frame:cudapy::__main__::add_1$241(
    Array<double, int=1, C, mutable, aligned>)
    (cudapy::__main__::add_1$241(
      Array<double, int=1, C, mutable, aligned>)
      : 0xf0)
```

RAPIDS

41

First of all, if we run cuda-memcheck without the debug=True keyword argument, then it does spot the error and report it.

When you try to see where it came from, we can at least tell the mangled name of the function that caused it.

However, it's not that easy to get any more detailed information from the output.

cuda-memcheck output

► With debug=True:

```
Invalid __global__ read of size 8
  at 0x00000360 in ../examples/debug_memcheck.py:11:
    cudapy::__main__::add_1$241(
      Array<double, int=1, C, mutable, aligned>)
  by thread (10,0,0) in block (0,0,0)
Address 0x7f4f75800050 is out of bounds
Device Frame: ../examples/debug_memcheck.py:11:
  cudapy::__main__::add_1$241(
    Array<double, int=1, C, mutable, aligned>)
  (cudapy::__main__::add_1$241(
    Array<double, int=1, C, mutable, aligned>)
    : 0x360)
```

RAPIDS

42

Now if we run again with debug=True then we also get the file name and the line number in which the illegal access occurred, along with the rest of the information that we had before, including the thread ID and block ID.

So that makes it a bit easier to narrow down what the problem is.

cuda-gdb

- ▶ Limited: `debug=True` doesn't give much extra help.
- ▶ Functionality available:
 - ▶ `cuda-memcheck`: set `cuda memcheck on` to break on memcheck error.
 - ▶ Set breakpoints on kernels
 - ▶ Step by instruction
- ▶ Limitations:
 - ▶ Backtraces don't work
 - ▶ No `list` command to view source
 - ▶ No line numbers
 - ▶ No stepping by source line

RAPIDS

43

You can also use `cuda-gdb` with Numba kernels, but my experience is that it's of limited help.

You can use `cuda-memcheck` in conjunction with `cuda-gdb`, so that you can get it to break on a memory error.

You can also set a breakpoint on a kernel launch, and then step through it instruction by instruction.

Presently you can't get a backtrace, or view the source - `cuda-gdb` seems not to have any concept of the line numbers or the source, so you also can't step by source line.

It's a little bit inconvenient but it can be a bit helpful if you're trying to pinpoint why something isn't working.

I'm hopeful that Numba's interoperability with `cuda-gdb` can be improved somewhat.

Example cuda-gdb session

```
$ cuda-gdb --args python debug_memcheck.py
(cuda-gdb) set cuda memcheck on
(cuda-gdb) run
Starting program: .../envs/numba/bin/python debug_memcheck.py
Illegal access to address (@global)0x7ffffb7800050 detected.
Thread 1 "python" received signal CUDA_EXCEPTION_1,
  Lane Illegal Address.
[Switching focus to CUDA block (0,0,0), thread (10,0,0), ...]
0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>) ()

(cuda-gdb) bt
#0  0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>) ()
#1  0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>)
    <<<(1,1,1),(32,1,1)>>> ()

(cuda-gdb) list
2 in /tmp/build/80754af9/python_1588880935370/work/Programs/python.c
```

RAPIDS

44

Here's an example of one short session with cuda-gdb.

I'm starting up my example code, turning on memcheck, and then running the program.

It soon stops because of the illegal access, but if I ask for a backtrace I only really get the current frame with the mangled name, and no source line information.

If we try to list the code, cuda-gdb looks for something in python.c instead, which is as far in the stack as cuda-gdb can see with this Numba code.

Example cuda-gdb session (2)

```
(cuda-gdb) disas
Dump of assembler code for function _ZN6cudapy8__main__9add_...:
=> 0x00005555574c6980 <+0>:  MOV R1, c[0x0][0x28]
    0x00005555574c6990 <+16>:  MOV R2, 0x180
    0x00005555574c69a0 <+32>:  LDC.64 R2, c[0x0][R2]
    ...

(cuda-gdb) stepi
0x00005555574c6990 in cudapy::__main__::add_1$241(Array<...>)
    <<<(1,1,1),(32,1,1)>>> ()

(cuda-gdb) stepi
0x00005555574c69a0 in cudapy::__main__::add_1$241(Array<...>)
    <<<(1,1,1),(32,1,1)>>> ()

(cuda-gdb) disas
Dump of assembler code for function _ZN6cudapy8__main__9add_...:
    0x00005555574c6980 <+0>:  MOV R1, c[0x0][0x28]
    0x00005555574c6990 <+16>:  MOV R2, 0x180
=> 0x00005555574c69a0 <+32>:  LDC.64 R2, c[0x0][R2]
```

RAPIDS

45

We can disassemble the code to see where we are.

We can then step a couple of instructions.

Then if we disassemble again, we can see that the instruction pointer's moved on a little.

CUDA simulator

► Features:

- Emulates CUDA execution model in Python
- Regular Python exceptions occur (e.g. OOB access)
- Break in kernels with Python debugger
- Step through kernels, view all variables, print out, etc.

► Limitations:

- Slow!
- Only one GPU simulated
- Threaded access / kernel calls not supported
- Most driver API unimplemented

RAPIDS

46

An alternative way to debug things is to use the CUDA Simulator, which gives you a lot more insight into the program, but it's much slower and more limited in functionality.

It emulates the CUDA execution model in Python, so you can raise exceptions in kernels, and break and step through things with the normal Python debugger.

You can also view all variables, and print things out, so you can see a lot more about what's going on.

The downside to it is that it's pretty slow and has some limitations. It only simulates one GPU, you can only access the simulated GPU from a single host thread, and most of the driver API features are unimplemented.

Using the CUDA simulator

Getting started:

- ▶ Set `NUMBA_ENABLE_CUDASIM=1` in your environment
- ▶ May need to run with small data set
- ▶ Or, strip down to minimal reproducer

Then:

- ▶ Run and see if exceptions occur, and/or
- ▶ Use e.g. `from pdb import set_trace; set_trace()`

Starting under debugger doesn't work well:

- ▶ OS threads implement CUDA threads - confuses debugger

RAPIDS

47

To enable the simulator, you can set an environment variable before you start up. Then, Numba ignores CUDA devices and launches everything with the simulator.

Because it's so slow you might need to configure your program to run with a small data set or fewer threads, or maybe strip it down to a minimal reproducer of your issue - otherwise the simulator might take a long time to reach the point you want to debug.

Once you're set up, you can just run and see if any Python exceptions get thrown, or you can modify your kernels to start the debugger at the point at which you're interested.

You can't usually start the program under the debugger because the threading used by the simulator seems to confuse the Python debugging infrastructure.

CUDA simulator debug example

- ▶ Run with simulator:

```
$ NUMBA_ENABLE_CUDASIM=1 python debug_check.py
...
File "debug_check.py", line 11, in reciprocal
    x[i] = 1 / x[i]
...
IndexError: tid=[10, 0, 0] ctaid=[0, 0, 0]:
    index 10 is out of bounds for axis 0 with size 10
```

- ▶ Add debug break to kernel:

```
if i == 10:
    from pdb import set_trace; set_trace()
```

RAPIDS

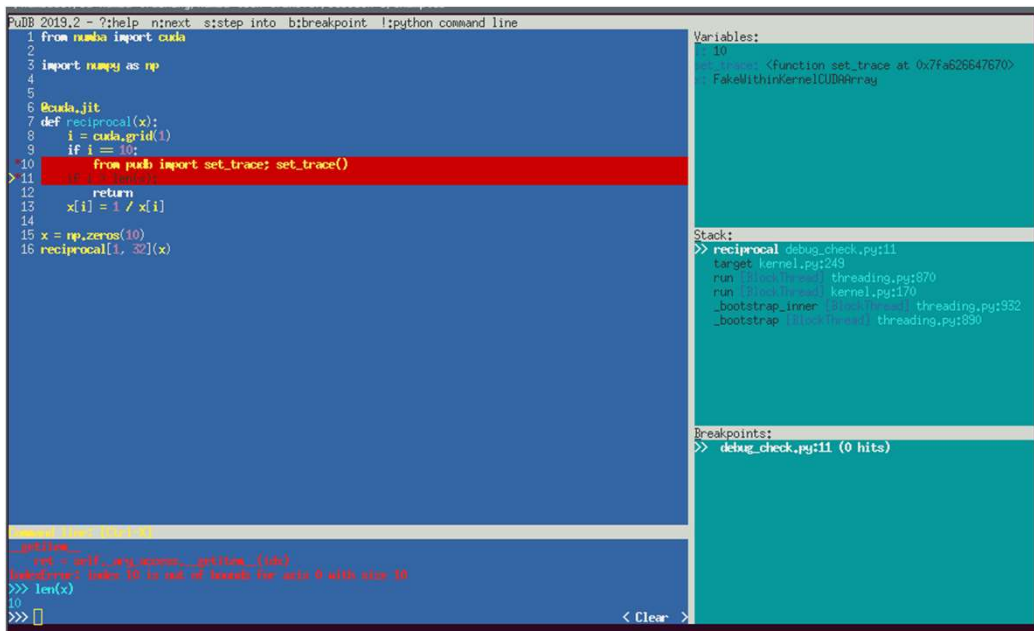
48

If we run the earlier example with the simulator, then we see an exception pop up.

If we want to look into the state leading up to that exception, then we can modify the kernel. The exception told us that thread 10 was responsible, so I'll specifically add `set_trace()` for thread 10.

The standard python debugger is called `pdb` - I'm using one called `PuDB`, which I like using because it has a bit of a nicer interface, but you can use any python debugger.

CUDA simulator debug example (2)



```

PuDB 2019.2 - ?help n:next s:step into b:breakpoint l:python command line
1 from numba import cuda
2
3 import numpy as np
4
5
6 @cuda.jit
7 def reciprocal(x):
8     i = cuda.grid(1)
9     if i == 10:
10        from puDB import set_trace; set_trace()
11        return
12    x[i] = 1 / x[i]
13
14 x = np.zeros(10)
15 reciprocal[1, 32](x)

```

Variables:

```

10
set_trace: <function set_trace at 0x7fa626647670>
FakeWithKernelCUDAMemory

```

Stack:

```

>> reciprocal debug_check.py:11
target kernel.py:249
run [1000 threads] threading.py:870
run [1000 threads] kernel.py:170
_bootstrap_inner [1000 threads] threading.py:932
_bootstrap [1000 threads] threading.py:990

```

Breakpoints:

```

>> debug_check.py:11 (0 hits)

```

Command Line (Ctrl+Q)

```

>>> len(x)
10
>>>

```

Now when I run again, I end up in PuDB.

- I can see what the values of variables are in the right
- I can print out the length of x
- And I can then see that indexing with i is going to overrun it.

From this point I can step through the kernel line by line, and generally debug it as if it were Python code.

Summary

- ▶ Porting strategies
- ▶ Measuring performance
- ▶ Interoperability
- ▶ Debugging

This session dealt with topics relevant to moving a pure Python code over to the GPU

- I've outlined my strategy for gradually migrating the code to the GPU through object mode, nopython mode, CUDA-jitting, and then managing data movement,
- We've looked at some approaches to and pitfalls with measuring performance to ensure that the port is providing speedups in the right places and to guide its optimization,
- We've seen how Numba can interoperate with other CUDA libraries that might provide better replacements for other CPU-based libraries,
- And we've covered the different debugging strategies using the hardware (cuda-gdb and cuda-memcheck), and the simulator.

Thankyou! / Questions?



RAPIDS

RAPIDS

Numba for CUDA Programmers Session 4 - Extending Numba

Graham Markall - gmarkall@nvidia.com

Reminder of previous sessions

- ▶ Using Numba
- ▶ Numba internals:
 - ▶ Type system
 - ▶ Performance optimisation
 - ▶ Debugging

Previous sessions were focused on how to use Numba: what it does, and what you can do with it.

We did look at some Numba internals. In particular we spent a fair bit of time looking at the type system, but that was mainly to help understand performance optimisation and debugging.

We haven't yet looked into extending or modifying Numba itself.

This week - extending Numba

- ▶ Why?
- ▶ Extension API:
 - ▶ Adding new types and functions
- ▶ Upstreaming new types and features
- ▶ My workflow:
 - ▶ Extension API first,
 - ▶ then consider internal modification / upstreaming

RAPIDS

3

So, this week focuses on extending Numba. Why would you, as a user, want to do this?

The reason is that Numba only compiles code using types and functions that it recognises. So, you have to rewrite your code to avoid using anything that it doesn't recognise. Alternatively, a more powerful solution is to add support for the types and functions that you want to use.

There's two ways you can go about adding that support. One is through the extension API, where you can write external code that augments Numba. The other way is to modify Numba itself.

Usually I would start adding support for new types and functions using the extension API because it's a bit simpler to do - you can keep your additions self-contained and bundled with your application code.

Subsequently, I'd look at modifying Numba and upstreaming the additions if it makes sense to do so.

Interval - running example class

```
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
```

► Source: <https://numba.pydata.org/numba-doc/latest/extending/interval-example.html>

RAPIDS

4

Just before we get started talking about these, we'll introduce this example class that most of the code we'll look at will be based around. It represents an interval using two floating point attributes, and it's got a property and an `__init__` method.

This example is from the Numba documentation, which does do a walkthrough of using the extension API.

However, that documentation focuses on the CPU target, whereas we'll be looking at extending the CUDA target.

Interval - attempt to use

```
@cuda.jit(void(float32[:,1], float32[:,1]))
def f(x1, x2):
    i = cuda.grid(1)

    region = Interval(x1[i], x2[i])
```

► Error:

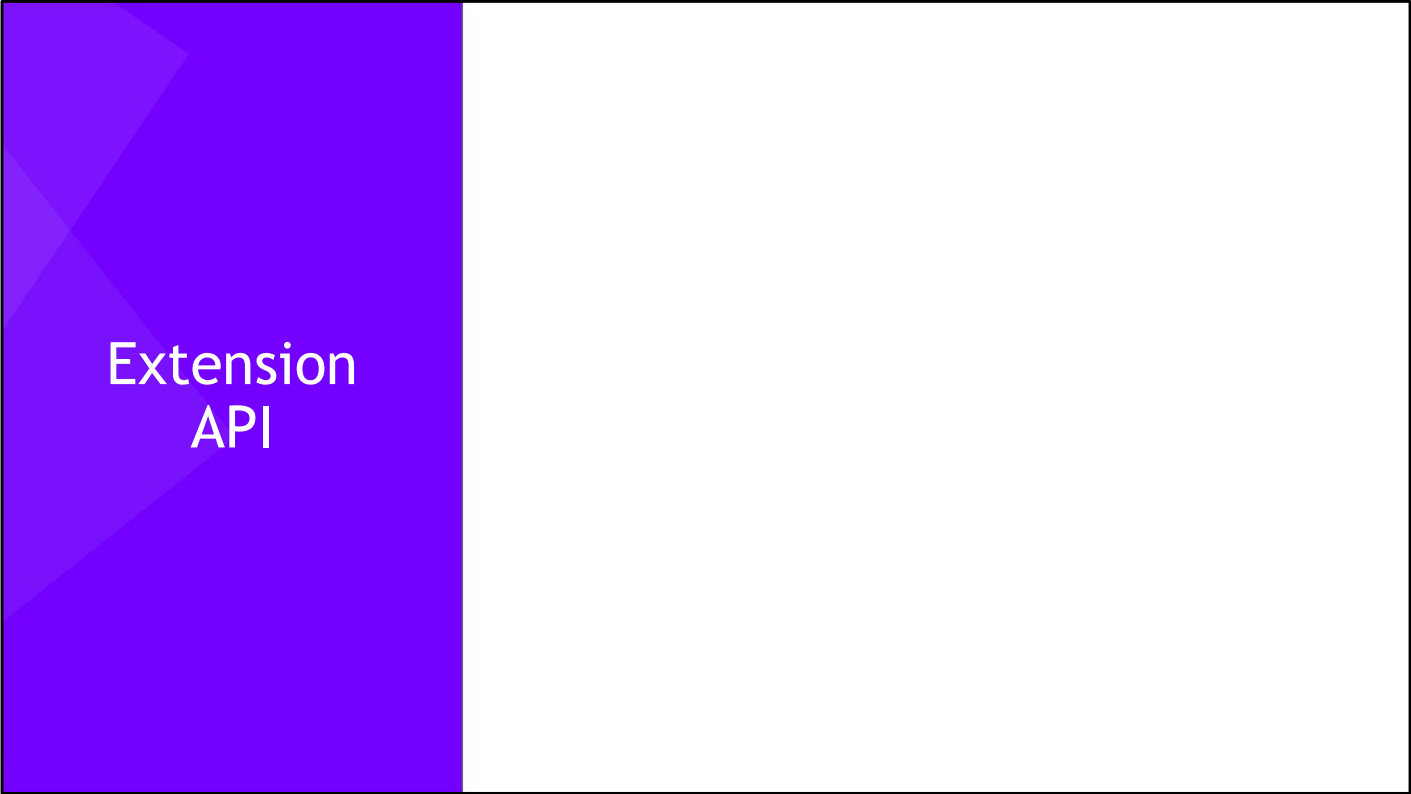
```
numba.core.errors.TypeError: Failed in nopython mode pipeline
Untyped global name 'Interval': cannot determine Numba type of
    <class 'type'>

File "internal_no_extension.py", line 24:
def f(x1, x2):
    <source elided>

    region = Interval(x1[i], x2[i])
```

So, let's try and use that class in a CUDA kernel right now - what happens?

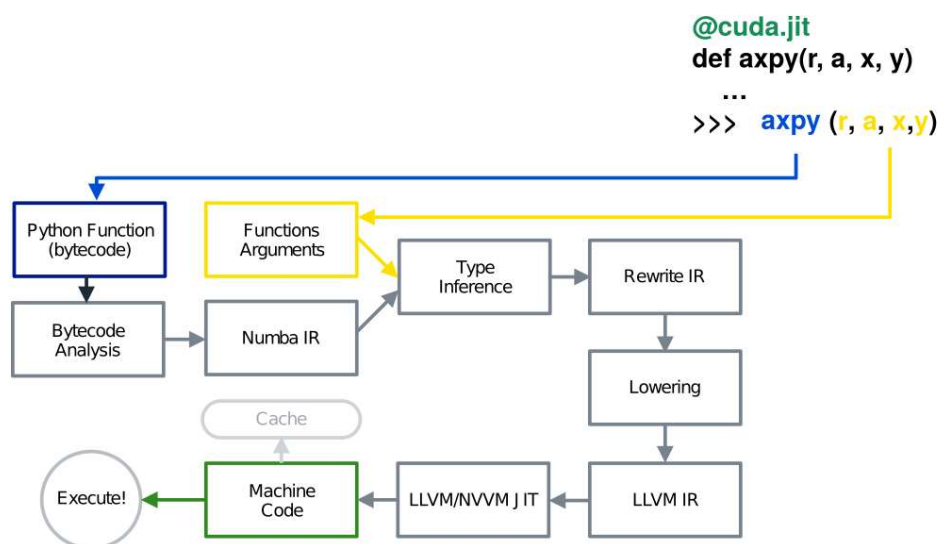
At the moment, we'll get a Typing error, because Numba doesn't know what to do with the Interval class.



Extension
API

Let's look at fixing things up with the Extension API!

Pipeline



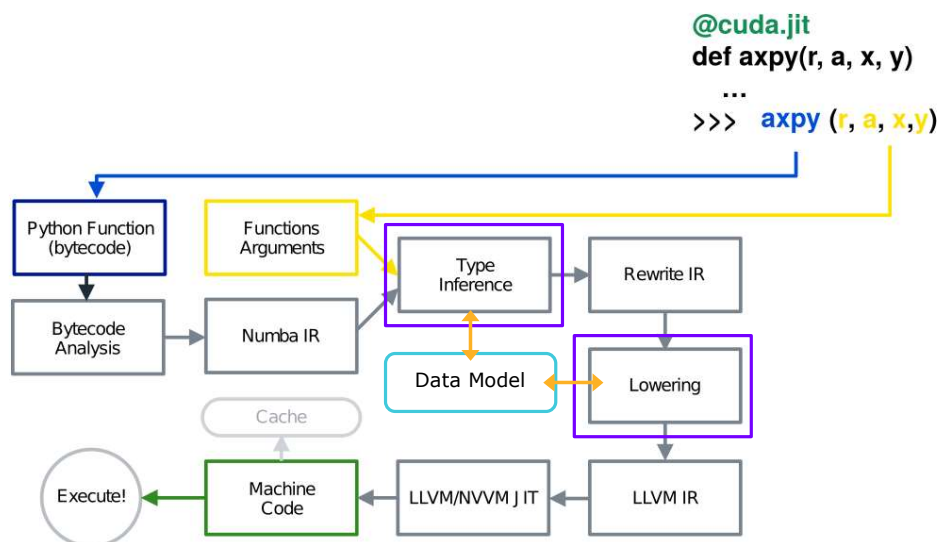
RAPIDS 7

Let's just refresh our memory of Numba's pipeline, to remind us of what we're going to be working with.

- The Python interpreter's bytecode compiler is used to generate bytecode
- Bytecode analysis translates the bytecode into the Numba Intermediate Representation (Numba IR).
- Type inference assigns a type to every input, intermediate value, and output in the Numba IR.
- There are some rewrite passes, which we can gloss over.
- Lowering, generates the LLVM Intermediate Representation (LLVM IR) from the Typed Numba IR.
- Then, NVVM is used to generate PTX code,
- After that, the PTX code is linked and loaded with the CUDA driver API, and the resulting module is cached and executed.

Stages affected by extensions

1. Type Inference
2. Lowering
3. Data model



We only need to work with specific parts of the pipeline to create an Extension. These are:

- Type inference,
- Lowering,
- And the Data Model.

The Data Model isn't really a pipeline stage - it's the component of Numba that provides some coupling between the Typing and the Lowering of Numba IR.

Type inference (1)

- ▶ Recognising argument types:

```
class Interval:
    # implementation omitted

@njit
def f(x):
    # implementation omitted

i = Interval(1.0, 3.0)

# How does Numba know what to do with `i`?
f(i)
```

- ▶ Note: passing in extension types presently only supported on CPU target

RAPIDS

9

So let's look at typing first. We'll use the couple of slides to show examples of what adding typing support enables Numba to do.

One thing it does is it allows Numba to recognise the types of arguments. So in this example, we have our Interval class, and some jitted function, and we pass an interval object to it in a call.

Numba needs to be able to recognise the type of what we've just passed in, to be able to do something meaningful with it. So, one of the problems our typing implementation has to do is to tell Numba how to do that.

At the moment you can only pass in extension types to functions compiled for the CPU - this is something I'm working on fixing for CUDA at some point in the near future.

Type inference (2)

- ▶ Propagating type information through operations on extension types:

```
@njit
def compute_width(x):
    # What are the types of lo and hi?
    lo = x.lo
    hi = x.hi
    return hi - lo

@njit
def interval_factory(lo, hi):
    # How does Numba know what type Interval() returns?
    new = Interval(lo, hi)
    return new
```

RAPIDS

10

The other thing that our typing does is to help Numba propagate the type information through operations on our extension type.

If we have an interval and we access its attributes, how does Numba know that their types are? Ideally we'd like Numba to know that lo and hi are both floats, so we need to somehow implement that typing.

Another example is the Interval constructor. That's going to return an Interval object, so we need to specify that.

The typing for the constructor also has to check that the types of things passed in (hi and lo in this case), are appropriate types to use for constructing an interval. For example, we'd like this call to the constructor to type correctly if we're passed lo and hi as float variables, but not if they're strings.

Defining a new Numba type

```
from numba import types

# Type class
class IntervalType(types.Type):
    def __init__(self):
        super().__init__(name='Interval')

# Type instance
interval_type = IntervalType()
```

RAPIDS

11

To be able to express the things we've just covered, we need to create a new Numba type that we can refer to.

We can do that by creating a type class that derives from `types.Type`. Generally all type class definitions look a bit like this boiler plate. For our interval type class, all we need to do is call the `Type.__init__` method in our `__init__` method with an appropriate name.

In most places in Numba, we don't usually pass type classes around or refer to them, but instead we use type instances. So, we'll create an instance of the type as well.

Type classes vs. type instances

In general:

- ▶ Instances are *specialized*, classes are *generic*
- ▶ Type instances used in most places
- ▶ Type classes sometimes used to match all instances

Parameterization example (numba.core.types.scalars):

```
class Integer(Number): # Derives from types.Type
    def __init__(self, name, bitwidth=None, signed=None):
        super().__init__(name='Integer')
        if bitwidth is None:
            bitwidth = parse_integer_bitwidth(name)
        if signed is None:
            signed = parse_integer_signed(name)
        self.bitwidth = bitwidth
        self.signed = signed
```

RAPIDS

12

Our interval type was pretty simple, but before we move on to discussing what we do with it, we'll spend a moment on the difference between classes and instances, and type parameterization.

All of our intervals are exactly the same from a typing perspective. Sometimes you need things that are of similar, but not exactly the same type, and this is where using instances of a parameterized type class comes in.

One reason for using type instances is that the instances are specialized versions of generic type classes. Generally an instance only matches other instances when Numba's looking for an implementation of something, but the type class matches all instances of the class. This is a bit abstract for now, but we'll see a bit more of the matching later when we talk about lowering.

To make some aspects of this concept more concrete, let's look at the Integer class from inside Numba.

Our interval type didn't have any attributes, but the Integer class has both a bitwidth and a signedness. So when we construct an instance of it, these are both set and fixed for that instance.

Rather than studying the code too hard, let's look at the instances on the next slide.

Parameterizations of Integer

```
# Type class
class Integer(Number):
    def __init__(self, name, bitwidth=None, signed=None):
        # ... impl omitted ..

# Specialized instances
byte = uint8 = Integer('uint8')
int16 = Integer('int16')
uint32 = Integer('uint32')
uint64 = Integer('uint64')

int8 = Integer('int8')
int16 = Integer('int16')
int32 = Integer('int32')
int64 = Integer('int64')
```

RAPIDS

13

Here's most of the parameterizations, taken from the Numba code base.

They should be familiar as the Numba types that we discussed using in previous sessions, like when you write a signature on the jit decorator for eager compilation.

These are all instances of the integer type class, and will all match the type class - but they're all different from each other, in bitwidth and signedness.

So Numba considers all these instances distinct from each other, and doesn't match two different instances as being equal.

Typing arguments and constants

```
global_interval = Interval(1.0, 2.0)
```

```
@njit
def lo_difference(x):
    return x.lo - global_interval.lo
```

```
param_interval = Interval(3.0, 4.0)
lo_difference(param_interval)
```

```
from numba.core.extending import typeof_impl
```

```
@typeof_impl.register(Interval)
def typeof_interval(val, c):
    return interval_type
```

- ▶ val: The instance being typed
- ▶ c: typeof context. c.Purpose is argument or constant

RAPIDS

14

Now that we've created a Numba type instance for Interval, we can start to define where it gets assigned to variables.

The example illustrates the two different circumstances that the type could be assigned:

- For things that are passed in as function arguments,
- and for global variables that are not passed in, but are referred to in a jitted function

Here we've got a function that refers to the global interval and it also gets passed in another interval through the x parameter.

To handle these cases is to register a typeof function. The typeof function is registered for a specific Python type, in our case the Interval class. Then, Numba will call this typeof function when it sees an instance of the Interval class.

The job of the typeof function is to return a Numba type. To do that, it gets passed two arguments - one is val, which will be the instance being typed and the other is c, called the typeof context.

You can use the typeof context to tell whether you're typing an argument or a global - but it almost never makes any difference, so I usually don't worry about that.

In this implementation we don't need to do much because our interval type isn't parameterized - we can just return the interval type.

If the type were parameterized, then we might inspect the value val to work out what kind of interval it is, and then return the appropriate parameterization.

Typing functions

- ▶ What type does `Interval(lo, hi)` return?

```
from numba.core.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
```

- ▶ Successful typing: return a Numba type (`interval_type`)
- ▶ Failed typing: return `None`
 - ▶ Failure is not an error - Numba tries other typers
 - ▶ Typing error occurs if all candidates exhausted

RAPIDS

15

Next, we need to define the typing associated with functions of the interval class. The interval constructor is a function, so we'll use that for this example.

To define a function typing, we register it using the `type_callable` decorator. This gets dispatched on the function being called - in this case it's `Interval`, for the interval constructor.

Our typing function should return another function that accepts the argument types, and returns a Numba type. For our example, the typer function checks that the numba types of `lo` and `hi` are `float32`, and if so, it returns the interval type. If not, then the typing failed, and it won't return anything.

It's not an error for a typing function to return nothing - although this particular typing can fail, Numba might try other available typing functions. Once it's exhausted all the possible typing functions, if they've all returned `None`, then the user does get a typing error.

In my experience this behaviour can make your typings a bit difficult to debug - the idea that Numba keeps on trying until it finds something or gives up can make it hard to catch when something goes wrong in one of your typing functions. So, this is something to be aware of when you're trying to see why a typing function isn't working.

The Data Model

Next we'll look at Data Models, and adding one for our Interval type.

Frontend to Backend mapping

- ▶ The Data Model maps between Numba and LLVM types
- ▶ Numba (frontend) types:
 - ▶ `int32`, `int64`, `uint8`, `slice2_type`, `range_iter32_type`, etc.
- ▶ LLVM (backend) types:
 - ▶ `i32`, `void`, `[40x i32]`, `{ i32, float, i8* }`
- ▶ For every new type, we need to add a Data model

RAPIDS

17

As mentioned before, the Data Model connects the front end types to the back end types. To illustrate the difference between them, we'll look at some of the Numba and LLVM types.

The front end types are more closely aligned with object types in Python, some examples of which are on the slide. You can see they're handling concepts like slices, and iteration over ranges. That's good for the semantics of python, but LLVM needs more primitive, lower-level types.

LLVM types are things like a 32-bit integer, an array of integers, or a struct with a 32-bit integer, a float, and a pointer to 8-bit integers.

So the LLVM types are much more like what you'd see in C, and you can envisage a straightforward mapping from them to register types or a memory layout.

For each Numba type, we need to add a data model that implements the mapping.

Interval Data Model

► StructModel: a read-only struct type.

```
from numba.core.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)
```

RAPIDS

18

There's quite a few models in Numba that make a good basis for extension type data models. I won't go through them here, but one that's appropriate for the Interval model is the StructModel - it makes sense to represent an Interval at a low level as a struct containing two floats.

This is what we do to use the struct model:

- First we have to register the model, noting that we register the data model for the type class, not for the type instance.
- Then in the `__init__` function we define the members of the struct, and their Numba type,
- Then initialize the struct model itself.

It might seem a bit odd that the members are defined in terms of Numba types - this works because the StructModel parent class handles the mapping from Numba types to the LLVM types of each member.

Float Data Model

► In `numba.core.datamodel.models`:

```
@register_default(types.Float)
class FloatModel(PrimitiveModel):
    def __init__(self, dmm, fe_type):
        if fe_type == types.float32:
            be_type = ir.FloatType()
        elif fe_type == types.float64:
            be_type = ir.DoubleType()
        else:
            raise NotImplementedError(fe_type)
        super(FloatModel, self).__init__(dmm, fe_type, be_type)
```

RAPIDS

19

Here's an example of another data model from Numba's internals - the Float Data Model.

What I'd point to here is that in general data models have these two attributes, `fe_type`, and `be_type`.

- The `fe_type` is the Numba type,
- and the `be_type` is an LLVM type. This ir module is the LLVM IR builder, and it's used here to provide either LLVM Float or Double type instances.

Lowering

Now we have our typing and our data model, we can move on to lowering.

This section on lowering has got a lot of information in a small space. The accompanying exercise notebook covers what's explained here.

So just follow along with what I'm saying to try and get some familiarity with the concepts, rather than attempting to commit all the details to memory now - then you can refer back to the notebook and the notes for the details later on.

Attribute Access

```
from numba.core.extending import make_attribute_wrapper

# make_attribute_wrapper(type, struct_attr, python_attr)
make_attribute_wrapper(IntervalType, 'lo', 'lo')
make_attribute_wrapper(IntervalType, 'hi', 'hi')
```

► Lower-level struct member access:

```
from numba.core.extending import lower_getattr_generic

@lower_getattr_generic(IntervalType)
def lower_interval_getattr(context, builder, sig, args):
    proxy_cls = cgutils.create_struct_proxy(interval_type)
    struct = proxy_cls(context, builder, value=args[0])
    return getattr(struct, args[1])
```

RAPIDS

21

Let's first look at lowering the access of attributes, which for our Interval class are lo and hi.

Numba provides a convenience function for read-only access to attributes in a struct model that we can use here. It's called `make_attribute_wrapper`, and you pass it:

- The type class you're lowering for, and
- the name of the struct model member to provide for a given python member name.

So with these two calls we've implemented the lowering for the Interval's attributes.

But, let's consider for the sake of argument that you want to lower something that needs to access struct members without returning them directly. The second example shows how to do this.

For now we'll focus on the content of this function rather than its parameters. Inside it we have a call to `create_struct_proxy`, which comes from a library in numba called `cgutils`.

`cgutils` has lots of little utility functions that save you from having to directly write code to generate LLVM IR. In this case the struct proxy gives you an object that generates the code to access struct members.

So first of all we create the struct proxy class for our interval type, then we can instantiate it with the struct that we've been passed in. Now, when we call `getattr` on the proxy object, it doesn't do the `getattr` in Python, but instead it inserts code into the output to access that struct, and what we're actually returning here is a bit of LLVM IR.

Now let's look outside the body of the lowering function. In general lowering functions get registered with a lowering decorator. In this case it's `lower_getattr_generic` because we're lowering a `getattr` - again, we register it for the relevant type class.

It doesn't matter what the function name is - i've chosen `lower_getattr_generic` because I think that's quite descriptive.

All lowering functions get passed four arguments:

- `context` is the typing context. You would usually use that to instantiate constants, and casts, and things that might be specific to a target architecture.
- `builder` is next, which is an llvm IR builder. When you call builder methods, it returns a piece of LLVM IR. You would often stitch together the instructions that you need by making calls to the builder's functions.
- `sig` is the signature of the function. This tells you the return type of the function, and its argument types, if you need them.
- `args` is the arguments to the function. This is a list of previously-built LLVM IR that comes from other lowering functions that ran earlier than the current one.

The lowering function is then expected to return LLVM IR.

So, that was a lot, but it covers the general structure of pretty much all lowering.

Function lowering

```
from numba.core.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()
```

RAPIDS

22

Next we'll have a look at a lowering of a function. This is an example lowering the call to Interval with two floats, which is the interval constructor.

This time we need to use `lower_builtin` - the name is a bit odd, but it basically means to lower a function. The arguments to the decorator are the function and then the types of the arguments we're lowering for. This example will match a call to Interval with two floats.

We've got the same arguments to the function as before the typing context, the IR builder, the signature, and the IR for the arguments.

Inside the body we use the signature to get the return type, which we can use to create a struct proxy for the Interval.

Because we didn't supply a value for the struct proxy, this creates a new struct object. Then we assign to the members of the struct using the struct proxy, setting hi and lo.

Now when we return, we want to return some LLVM IR for the struct we created, rather than the struct proxy itself. We get at the underlying struct to return by calling the `_getvalue()` method.

That's all for the implementation of the constructor function.

Inspecting IR

► Dump LLVM IR with NUMBA_DUMP_LLVM=1:

```
interval = cgutils.create_struct_proxy(typ)(context, builder)
```

```
# %"x" = alloca {double, double}
# store {double, double} zeroinitializer, {double, double}* %"x"
```

```
interval.lo = lo # for Interval(1.0, 3.0)
```

```
# %"$const4.1" = alloca double
# store double 0x3ff0000000000000, double* %"$const4.1"
# %".12" = load double, double* %"$const4.1"
# %".17" = getelementptr inbounds {double, double},
#           {double, double}* %".14", i32 0, i32 0
# store double %".12", double* %".17"
```

RAPIDS 23

- What we've written in terms of code generation is all a bit decoupled from the LLVM IR,
- so we'll look at a bit of that now and see what it corresponds with.

In general if you want to inspect LLVM IR, you can do it by setting the NUMBA_DUMP_LLVM environment variable.

On the slide we have some abridged examples of dumped IR.

First, what did the struct proxy object generate? When we created it, we got an allocation of a struct containing two doubles, and then that was initialized to zero

When we set the lo member of the struct, this is what we see:

- First the IR allocates a space for a single constant, then stores the constant value in it,
- Then it loads that value into another variable,
- Then looks up a pointer to the lo member of the struct,
- and stores the value to the right location in the struct.

That IR is a bit verbose, but it's not generally a problem because it usually gets shortened and simplified by LLVM's optimizer.

Implementing a property / attribute

- ▶ For the CPU target:

```
@overload_attribute(IntervalType, "width")
def get_width(interval):
    def getter(interval):
        return interval.hi - interval.lo
    return getter
```

- ▶ Implementations written in Python and JITted
- ▶ Not supported on CUDA yet work-in-progress:
 - ▶ Pull Request: [Extending the extension API for hardware targets](#)

RAPIDS

24

Now that we've implemented attribute access and a function call, the other thing we didn't implement was property access. Property of the interval class called width.

On the CPU target, you've got this nice decorator called `overload_attribute`, where you just declare the type you're overloading and the name of the attribute, and then you can write the code that implements that property in Python, and Numba will JIT compile it to use as the implementation, so it saves you from having to manually build LLVM IR for the implementation.

This isn't yet supported on the CUDA target, but it will be added soon.

CUDA attribute implementation

► Need to augment CUDA-specific typing and lowering

► Typing:

```
from numba.core.typing.templates import AttributeTemplate
from numba.cuda.cudadecl import registry as cuda_registry

@cuda_registry.register_attr
class Interval_attrs(AttributeTemplate):
    key = IntervalType

    def resolve_width(self, mod):
        return types.float64
```

RAPIDS

25

Instead what we have to do is to directly add to Numba's CUDA-specific typing and lowering code.

First we'll look at the typing code. To do this we need to import the CUDA typing registry to register a new attribute with it. For properties, we'd tend to use this `AttributeTemplate` class and register our class for a particular type by storing it in the `key` member of the class. Then, we can add as many functions to this class as there are properties we want to register - we begin the method name with `resolve_`, and concatenate the name of the property it types.

So, if it sees us accessing `i.width` where `I` is an instance of an `Interval`, it knows to use this class because of the `IntervalType` key, and will call `resolve_width` because of the property name.

What the `resolve` method has to do is return a Numba type for the property. Width is a float, so we just return a `float64` type.

So that gives the CUDA target enough information about the type of the property.

CUDA attribute implementation

► Need to augment CUDA-specific typing and lowering

► Lowering:

```
from numba.cuda.cudaimpl import lower_attr as cuda_lower_attr

@cuda_lower_attr(IntervalType, 'width')
def cuda_Interval_width(context, builder, sig, arg):
    lo = builder.extract_value(arg, 0)
    hi = builder.extract_value(arg, 1)
    return builder.fsub(hi, lo)
```

► IR (Abridged):

```
# "%.49" = load {double, double}, {double, double}* %"x"
# "%.50" = extractvalue {double, double} "%.49", 0
# "%.51" = extractvalue {double, double} "%.49", 1
# "%.52" = fsub double "%.51", "%.50"
```

RAPIDS

26

Next, we also need to add to the lowering of the property as well. To do this we import `lower_attr` from the CUDA implementations. This is a bit like the lowering functions we've seen before - we register the lowering function for the type of the property and its name.

Then we have the familiar signature of a lowering function that receives a context, a builder, the signature, and the arguments.

Using the LLVM IR builder this implementation pulls out the first and second members of the struct and stores them in `lo` and `hi`. Then it calls `builder.fsub` to construct the LLVM IR for a floating point subtraction. Because `fsub` generates the `fsub` IR, this is ready to be returned from the lowering function.

What you would see in the dumped LLVM IR for this is:

- The first line loads the interval argument into a local variable
- The `extractvalue` instruction is used to get out the 1st (0) and 2nd (1) values into local variables
- And then the `fsub` operates on these extracted values.

So there is a close correspondence when you use the IR builder between what you write in the Python code and the IR that you see dumped.

Now we've added the typing and lowering, the implementation of the attribute is complete.

Trying it out

► The accompanying notebook contains some examples using the Interval class on CUDA:

```
@njit
def inside_interval(interval, x):
    """Tests attribute access"""
    return interval.lo <= x < interval.hi
```

```
@njit
def interval_width(interval):
    """Tests property access"""
    return interval.width
```

```
@njit
def sum_intervals(i, j):
    """Tests the Interval constructor"""
    return Interval(i.lo + j.lo, i.hi + j.hi)
```

```
@cuda.jit
def kernel(arr):
    x = Interval(1.0, 3.0)
    arr[0] = x.hi + x.lo
    arr[1] = x.width
    arr[2] = inside_interval(x, 2.5)
    arr[3] = inside_interval(x, 3.5)
    arr[4] = interval_width(x)
```

```
y = Interval(7.5, 9.0)
z = sum_intervals(x, y)
arr[5] = z.lo
arr[6] = z.hi
```

RAPIDS

27

The accompanying notebook contains some examples using the Interval class in CUDA - this slide contains some excerpts from that notebook.

On the left are some functions that test various parts of the Interval implementation. They're decorated njit, but that's OK - you can call njit functions from CUDA kernels and they'll be treated as if they're device functions.

On the right is a kernel that constructs an interval and calls some of these functions - this tests the functionality in them, and also tests passing Intervals to functions.

I'd recommend trying this out in the notebook to see the output that the kernel produces.

Modifying Numba, Upstreaming

Once you're happy with an extension that you've made, or when you've run into some limitation with the Numba extension API, you might want to look into modifying Numba itself.

Or, you may want to implement your extension within Numba with a view to upstreaming the support you've added.

The principles for modifying Numba are the same as those for writing an extension - it's just a little different in practice.

Why, and what to modify

- ▶ Why?
 - ▶ Standard library functionality - e.g. math module functions.
 - ▶ Equivalents to CUDA C/C++ features - e.g. Cooperative Grids
 - ▶ Support for commonly-used PyData libraries (e.g. NumPy)
- ▶ What? CUDA target-specific files in numba/cuda:
 - ▶ Typing: [cudadecl.py](#)
 - ▶ Data Models: [models.py](#)
 - ▶ Lowering: [cudaimpl.py](#)
 - ▶ Separate modules, c.f. libdevice implementations ([libdevice*.py in numba/cuda](#))

RAPIDS

29

Why would you modify Numba and upstream the changes, instead of making an extension?

One reason could be that you've added support for something that's in the standard library, for example something in the math module. Although the CUDA target supports some of the math module, there's probably some functions there that are yet to be implemented (or there might be functions in other libraries to implement).

Another reason you might extend Numba is to provide equivalent implementations of CUDA C/C++ features that haven't yet been added to Numba. Examples of these features would be things like thread groups, grid groups, asynchronous barriers, etc. - it would be great to have all of these in Numba, and there's still plenty of opportunities to add these features.

You might also want to add support for features from commonly-used PyData libraries - for example NumPy or other libraries.

The four main files that you'd expect to touch to extend Numba itself are:

- Typing would typically get added to `cudadecl.py`, for CUDA features.
- Data models are kept in `models.py`
- Lowering implementations for CUDA features go into `cudaimpl.py`

If you're going to be adding a large body of functionality, it might make more sense for you to create new modules in the CUDA target for your typing and lowering implementations - this is done for the libdevice function implementations.

Example contributions

- ▶ Implementing `isinf(x)` and `isnan(x)` where `x` is an integer:
 - ▶ <https://github.com/numba/numba/pull/5761/files>
- ▶ Implementing `math.modf(x)` for CUDA:
 - ▶ <https://github.com/numba/numba/pull/5578/files>
- ▶ Implementing `math.frexp(x)` and `math.ldexp(x, y)` for CUDA:
 - ▶ <https://github.com/numba/numba/pull/6064/files>

Some examples of past contributions to Numba can also be a useful guide to follow for extending Numba.

There are some links on the slide to PRs that made relatively small additions to the CUDA target, that should be easy enough to follow and use as a guide for your own extensions.

Documentation links

- ▶ [llvmlite](#): llvmlite is used to build IR
 - ▶ See [IRBuilder](#) for builder functions
- ▶ [Numba developer documentation](#)
- ▶ [Extending Numba](#)
- ▶ [LLVM 7.0.0 IR reference manual](#) - matches the IR version used by NVVM
- ▶ [NVVM IR specification](#) - Reference for CUDA-specific IR

General approach:

- ▶ Find something similar to what you're trying to do in numba/cuda
- ▶ Copy / modify the code until it does what you want

RAPIDS

31

There's no one comprehensive source of information that you might draw on when writing a Numba extension. This slide collects some links to useful documentation that will help you understand how to build LLVM IR using llvmlite, some general Numba documentation, and references for the LLVM and NVVM IRs that you'll be working with.

A good strategy is to find an implementation of something similar to what you're trying to do in the CUDA target source (or look at one of the PRs linked in the previous slide) and then copy or augment it with modifications until it does what you want.

Summary

- ▶ **Typing:** Recognising instances and propagating type info
- ▶ **Data Model:** Coupling front-end with back-end types
- ▶ **Lowering:** Converting Numba IR to LLVM IR
- ▶ **Upstreaming:** Adding to internal Typing, Lowering, and Models

This week we've looked at extending Numba. Typically, extensions consist of three components:

- The typing, which helps Numba's type inference work out the types of instances, function return values, members, and properties.
- The data model, which couples Numba's front-end Python-like types to low-level LLVM types
- And the lowering, that translates Numba IR code into LLVM IR.

We also looked at moving extension code into Numba, which you might do for the purpose of upstreaming your additions.

Thankyou! / Questions?



RAPIDS

RAPIDS

Numba for CUDA Programmers Session 5 - Memory Management

Graham Markall - gmarkall@nvidia.com

This week - memory management

- ▶ Numba Memory Management Internals
- ▶ Using External Memory Management (EMM) Plugins
- ▶ EMM Plugin Development

RAPIDS

2

This week we'll look in more detail at memory management with Numba. There's a reasonable amount of complexity in Numba's memory management, but I don't think it will take a whole hour to cover like the other sessions, so today is a little shorter.

There's a couple of areas we'll look at:

- First we'll cover the mechanism that Numba uses internally, that you get by default.
- And it's also got a way to replace that, so we'll look at why and how you'd replace it with some other mechanism as well.

Memory Management Internals

First of all we'll look at the internal memory management mechanisms.

Numba Memory Management

- ▶ Direct calls to Driver API `cuMemAlloc`, `cuMemFree`, etc.

- ▶ Allocations made when object constructed. E.g.:

```
# Results in immediate call to cuMemAlloc
cuda.device_array()
```

- ▶ Deallocations queued up, completed when object garbage collected:

```
# cuMemAlloc called
x = cuda.device_array(5)
# Deallocation queued, cuMemFree not called yet
del x
```

RAPIDS

4

We did touch on memory management a bit in the first session, where we mentioned that when Numba needs to allocate or free memory, it does it directly using the driver API.

Whenever you call `device_array`, or one of the other functions that copies data to the device, it immediately makes a call to `cuMemAlloc`.

As the user you're not responsible for deleting memory when you've finished it - you just need to wait for the object to get garbage collected, and then Numba will eventually call `cuMemFree` on it.

In the example I'm forcing the `x` object to get garbage collected, but this isn't the point at which `cuMemFree` gets called. Instead, Numba adds the memory underlying `x` to a queue of pending deallocations. At some point later it goes through the deallocations queue and calls `cuMemFree` on each item in it in a batch.

Queue processing

► Deallocations processed when:

- Queue reaches maximum entry count, or
- Queue reaches pending size limit, or
- An allocation needs more memory than is available.

► Controlled by config variables. Defaults:

- `config.CUDA_MAX_PENDING_DEALLOCS_COUNT` : 10 deallocations
- `config.CUDA_MAX_PENDING_DEALLOCS_RATIO` : 0.2 (20% of device memory)

RAPIDS

5

So when, does it actually decide to run through the deallocations? There are a couple of different policies governing that behaviour.

- The first is the maximum queue length - when the queue fills up with entries, Numba empties everything out of it.
- The second is the pending size limit - that is, the queue will only hold up to a certain number of bytes' worth of deallocations before it gets forced to empty. This stops you having a small number of very large deallocations holding on to a substantial amount of device memory for too long.

If neither of these limits have been reached and you try to allocate some new memory but the allocation fails then Numba will empty the queue and try the allocation again. That way, an allocation that should succeed never gets blocked by a lack of resources caused by the pending deallocations.

You do have some control over the first policies - you can set some configuration variables to override their values.

- The default queue length is 10, which is not a lot of allocations. If you have loops that do a lot of allocation and deallocation, you'll quickly be filling the queue repeatedly so you might well want to increase the maximum queue length.
- The default size limit, which is referred to as the ratio, is 20% of device memory. If you want to change that, then you can specify it as a fraction. If you experience a lot of memory pressure, then you might want to tune it down a bit.

Deferring deallocation

- ▶ Deallocation can break asynchronous execution:

```
async_func1(...) # Allocates some arrays, does async work
# async_func1's allocations are out of scope now
async_func2(...) # Does more async work
# Deferred cleanup could happen in the middle of async_func2
```

- ▶ Use `defer_cleanup()` context manager to temporarily prevent deallocations:

```
with cuda.defer_cleanup():
    async_func1(...) # Allocates some arrays, does async work
    async_func2(...) # Does more async work
# Deallocations can only happen here, outside the with block
```

- ▶ Caveat: easier to run out of memory.

RAPIDS

6

Another thing to think about is that although you've got some control over the deallocation policies, you don't have explicit control over their effects. Changing the values will affect when the queue gets processed, but doesn't specify exactly when the queue will be cleared.

So, if you want to make sure that you avoid queue processing at a certain time then you need to do something else. You might want to do this if you're doing lots of asynchronous operations. When you create a sequence of async operations like kernel launches and memcopies, your python code proceeds onwards, but if your stream of operations gets interrupted by Numba clearing the queue then it breaks up the stream of async operations and forces synchronization with the device.

You can stop any queue processing and deallocation temporarily using the `defer_cleanup` context manager, so that any code inside its `with` block can run asynchronously in an unbroken stream.

In the example on the slide, the first function allocates some memory that goes out of scope and gets garbage collected, leaving some pending deallocations. Then, inside the second function, Numba goes through the pending deallocation list, causing a synchronization.

If we put both those calls inside the `with` block, then the deallocations can't happen until after `async_func2` has returned.

This can be handy, but you do need to be aware when you're using it that it makes it easier to run out of memory because you're effectively preventing the freeing of any device memory at all.

External Memory Management Plugins

That's covered the internal memory management mechanisms - now we'll continue by looking at using external plugins for memory management.

Recap - CUDA Array Interface

- ▶ A standard for different libraries to exchange / use each others' on-device data.
- ▶ Objects implement `__cuda_array_interface__`. Returns a dict:

```
class MyArray:
    @property
    def __cuda_array_interface__(self):
        return {
            'shape': self._shape,
            'typestr': self._typestr,
            'data': (self._data, False),
            'version': 2
        }
```

- ▶ Supported in Numba, CuPy, PyTorch, PyArrow, mpi4py, ArrayViews, JAX, RAPIDS.

RAPIDS

8

Before we get into the detail of that, this is a recap of the CUDA Array Interface that we talked about in session 3.

If you have a library that allocates CUDA memory, you can share your objects with other cuda libraries by implementing the interface. That interface provides a dict with a pointer to the data and some other information, like its type and shape.

There's quite a few libraries that implement it now - these include Numba, CuPy, PyTorch, cuDF, and some other examples on the slide.

Example - Numba on CuPy data

Calling a Numba Kernel on a CuPy array:

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

add[1, 32](a, b, out)
```

RAPIDS

9

This is one of the examples from before, as a reminder that the interface is transparent to the user.

We create a Numba kernel and a CuPy array, and then we can call the kernel on the array without having to do anything manually to exchange data between the two libraries.

Numba sees an object that's not one of its own device arrays, and it checks for the CUDA Array Interface. It then uses that information to prepare the kernel arguments for the launch.

Library memory management

Problem: Combining libraries oversubscribes GPU memory

- ▶ Numba: [Deferred Deallocation](#)
- ▶ RAPIDS: [Rapids Memory Manager device pool](#)
- ▶ CuPy: [Device and Pinned Memory Pools](#)
- ▶ PyTorch: [Caching memory allocator](#)

RAPIDS

10

This transparent sharing of memory between different libraries is nice and convenient, and it works well as all the libraries you're using have fairly basic memory management strategies like Numba does.

In more complex scenarios, it can be problematic. Several libraries implement pool or caching allocators, that allocate a very large chunk of memory. Then when the library needs some more device memory, it doesn't use a CUDA driver or runtime function to allocate memory, but instead it just slices up its pool or cache to provide chunks of memory to the application.

That makes for a much more efficient implementation because you can effectively allocate and deallocate memory without synchronizing with the device and without using any CUDA calls, but there's a downside when you combine two libraries that both do this.

To start up, a library with a memory pool might allocate a really substantial chunk of memory. For example, RMM might start up by creating a pool that uses half the device memory. So if you use that, and you use a device memory pool from CuPy, you can find that before any application code is running, nearly all your device memory is allocated.

That doesn't leave you any for your own purposes outside the libraries, and it can also mean that the libraries can run out of memory in their pools, then be unable to grow the pool because another library is taking up the rest of the device memory with its own pool.

So you end up with this situation where you've got a lot of memory allocated, but not a lot in use, and not being able to allocate more.

There's some links on the slide to some of the memory allocation strategies used in some libraries - apart from Numba they're all using some kind of memory pool.

Using one strategy

► **Solution:** Configure all libraries to use one implementation

► Example:

```
import cupy
import rmm

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)

# Allocated using RMM
a = cupy.arange(5)
```

RAPIDS

11

We can overcome this problem by getting all the libraries to use the allocator from just one of them. Several of the libraries we've just mentioned support this.

There's an example on the slide, where you might want to use the RAPIDS Memory Manager with CuPy. You can do that by importing them both, then before you do any allocations, you can call `set_allocator` in CuPy to tell it to use RMM. RMM includes a function called `rmm_cupy_allocator` that CuPy can call on for allocations.

Now when you create CuPy arrays, the underlying memory comes from the RMM pool.

Using RMM with Numba

```
from numba import cuda
import rmm

cuda.set_memory_manager(rmm.RMMNumbaManager)

# Allocated using RMM
a = cuda.device_array(5)
```

► CuPy + Numba using RMM pool:

```
from numba import cuda
import rmm
import cupy

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)
cuda.set_memory_manager(rmm.RMMNumbaManager)
```

RAPIDS

12

You can do a similar thing to use RMM with Numba. The interface is a tiny little bit different, but the idea is still the same. In this case we call `cuda.set_memory_manager` with the `RMMNumbaManager` class, then all Numba allocations get made using RMM. We'll look in a bit more detail at the `RMMNumba` manager class later.

If you're going to be using CuPy and Numba both together and want to use RMM for everything, then you can just import everything and set the allocator for both.

So, that's covered the API for using external memory management plugins. If you're using a library that implements an external memory management plugin for Numba then you can just set Numba up to use it by calling `cuda.set_memory_manager` with the class.

At the moment I'm only aware of RMM including a Numba memory management plugin within its source.

EMM Plugin Development

You might want to use another library's memory manager, and to do that you can write a plugin that lives separately from the library that you want to use for allocation.

So that's what we'll look at next - we'll see how the EMM plugins like the RMM Numba Manager work, and how we can implement one.

Why write an EMM plugin?

- ▶ Interfacing with a library that has internal memory management
 - ▶ For which no plugin is currently implemented.
- ▶ Want to use that rather than Numba's direct approach.

Why write an EMM Plugin?

- Perhaps you're working with some library that has some complex memory management implementation that you'd like to use.
- Maybe implementing your own memory allocation strategy would be more efficient for your use case.

Implementing a plugin

- Implement the BaseCUDAMemoryManager interface.

```
class BaseCUDAMemoryManager:
    def memalloc(self, size):
        """Allocate on-device memory in the current context."""

    def memhostalloc(self, size, mapped, portable, wc):
        """Allocate pinned host memory."""

    def mempin(self, owner, pointer, size, mapped):
        """Pin a region of host memory that is already allocated."""

    def get_ipc_handle(self, memory):
        """Return an IPC handle from a GPU allocation."""

    def defer_cleanup(self):
        """Context manager that ensures that cleanup is deferred
        while it is active."""
```

RAPIDS

15

So how do you go about implementing an EMM plugin? You need to write a class that implements the BaseCUDAMemoryManager interface. We'll have a look at the methods in that interface, and what they need to do. Then, we'll have a look at a couple of examples of implementations.

On this slide are some of the methods of the interface that are directly related to memory management.

When the plugin is in use, Numba calls the methods as needed.

- If it needs an allocation of device memory, it'll call memalloc.
- If it needs pinned host memory, it'll call memhostalloc.
- mempin is used for pinning already allocated memory.
- get_ipc_handle is needed in a multi-gpu setup when numba needs an IPC handle for one of your allocations.
- Finally, in defer_cleanup you should make sure that your plugin won't do any deallocations or operations that would synchronize with the device.

BaseCUDAMemoryManager continued

► Housekeeping methods:

```
def initialize(self):
    """Do anything needed for the EMM plugin instance to be ready."""

def get_memory_info(self):
    """Returns ``(free, total)`` memory in bytes in the context.
    Can throw NotImplementedError."""

def reset(self):
    """Clears up all memory allocated in this context."""

@property
def interface_version(self):
    """Version the plugin implements. Should be 1."""
```

RAPIDS

16

There's also some housekeeping methods you might need to implement.

- In `initialize`, you can do anything you need to get your memory management ready to use in the current context.
- `get_memory_info` should return the free and total memory in bytes. It is possible that that's not an easy thing to do if you've got a complex memory management strategy and it might not make sense to just reduce the state down to a simple count like that, so you can throw `NotImplementedError` instead if you want.
- `reset` is called when Numba resets the current context. If this method gets called, then you should clean up everything in the current context.
- Also, the interface is versioned just in case it changes in future. There's only one version for now, so you should always return 1.

Device-only plugins

- ▶ Use the `HostOnlyCUDAMemoryManager` class as a base.

```
class MyEMMPlugin(GetIpcHandleMixin, HostOnlyCUDAMemoryManager):
    def memalloc(self, size):
        # Return MemoryPointer to device memory of `size`.

    def initialize(self):
        pass

    def get_memory_info(self):
        raise NotImplementedError

    @property
    def interface_version(self):
        return 1
```

- ▶ Using the plugin:

```
cuda.set_memory_manager(MyEMMPlugin)
```

RAPIDS

17

There is a fair bit to implement for the `BaseCUDAMemoryManager` that you might not want to have to do. For example, you might only want to implement a device memory allocation strategy, if you're trying to use a library that doesn't manage host memory.

So, there is a base class that can simplify things a bit, which is the `HostOnlyCUDAMemoryManager`. It takes care of the host-side memory management, so your derived class needs to only look after the device memory.

That way, you can have a fairly minimal implementation that looks something like this on the slide. Here I'm also using the `GetIpcHandleMixin` to save implementing `get_ipc_handle` - that should work in most cases, so in general you probably want to use it for implementing a plugin.

So for a minimal implementation you only need to implement `memalloc`, provide a stub implementation of `get_memory_info`, and give the interface version. Then, you should be able to use it with `cuda.set_memory_manager`.

Return values

```
class MyEMMPlugin:
    def memalloc(self, size):
        ptr = my_alloc(size) # Returns e.g. a ctypes c_uint64
        ctx = self.context
        finalizer = make_finalizer(ptr.value)
        return MemoryPointer(ctx, ptr, size, finalizer=finalizer)

def make_finalizer(ptr):
    def finalizer():
        my_free(ptr)

    return finalizer
```

RAPIDS

18

So far we've glossed over what goes in the implementation of the memalloc function, so let's have a look at it now.

In this example, we'll assume there's some function called my_alloc that somehow allocates device memory of a given size, and returns a pointer to that device memory.

Then we need to put together arguments for constructing a MemoryPointer object. We'll look at the MemoryPointer and similar classes shortly. For now, we'll look at what's needed to construct one:

- We need to get the current context. That's always available in self.context in an EMM Plugin implementation.
- We also need a finalizer. The finalizer is a function that gets called when the MemoryPointer is garbage collected. The finalizer should do whatever's necessary to clean up the allocation. In this example, our finalizer function calls a function called my_free, that we'll assume just frees the pointer immediately.

So with all the above, we can construct the MemoryPointer and return it.

MemoryPointer class

- ▶ Used by Numba internally to represent allocations.

- ▶ Construct with:

```
MemoryPointer(context, pointer, size, owner=None, finalizer=None)
```

- ▶ context: The context in which the memory is allocated

- ▶ pointer: The address of the memory.

- ▶ size: The size of the allocation in bytes.

- ▶ owner: Don't set this in an EMM Plugin.

- ▶ finalizer: Function to be called when the object is GC'd

We've just constructed a `MemoryPointer`. This is the class that's used internally in Numba to represent allocations of device memory.

We've already seen most of the arguments and how to use them. The only other one is `owner`, which Numba sometimes sets, but it's not something you need to set from an EMM plugin, so you can just leave it out the argument list.

Other memory classes

- ▶ `MemoryPointer`: Returned from `memalloc`.
- ▶ `MappedMemory`: Return from `memhostalloc` or `mempin` when host memory mapped into device space.
- ▶ `PinnedMemory`: Return from `memhostalloc` or `mempin` when host memory NOT mapped into device space

RAPIDS

20

The `MemoryPointer` class is for device memory only. If you're implementing a plugin that manages host memory too, then there's a couple of other classes to be aware of:

- The `MappedMemory` class is used for host memory mapped into device space,
- The `PinnedMemory` class is for host memory that's not mapped into the device space.

All of these objects are constructed in the same way as the `MemoryPointer` class, so we'll not go over the arguments for constructing them again.

Examples

- ▶ Link to [RMMNumbaManager](#)
- ▶ Link to [NumbaCUDAMemoryManager](#)

We've covered quite a lot of the specifications and details for an EMM Plugin.

This slide links to two EMM Plugin implementations that you can read to understand the implementation of an EMM Plugin in a real-world context.

Summary

Key points:

- ▶ Numba uses immediate alloc / deferred dealloc
- ▶ Use EMM plugins to replace internal with third-party allocator
- ▶ Write EMM plugins to interact with other allocators

Doc links:

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

This session we've:

- Examined Numba's memory allocation strategy that defers deallocations to increase performance, and looked at how to modify its cleanup policies as well as suspending deallocation to support asynchronous use cases,
- Looked at how to replace Numba's memory allocation strategy with an external mechanism,
- And how to implement an External Memory Management plugin to provide our own memory management mechanism.

Documentation that is useful to follow when looking at Numba's memory management or implementing your own memory management is linked on the slide.

Thankyou! / Questions?



RAPIDS