

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.

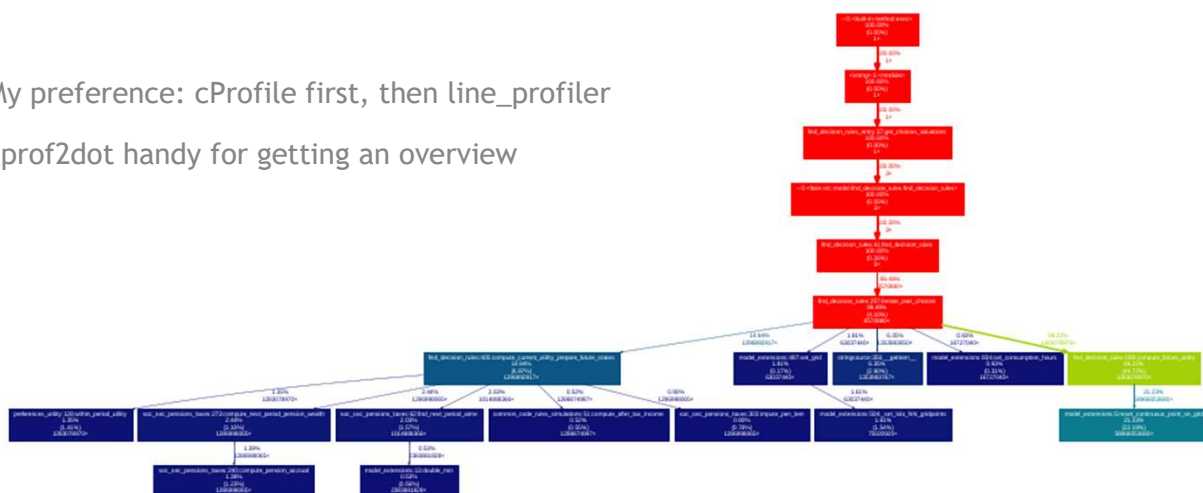
Internally, Numba uses the occupancy calculation of the kernel and the number of elements to work out a good launch configuration.

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
```

RAPIDS

39

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)0x7fffb7800050 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

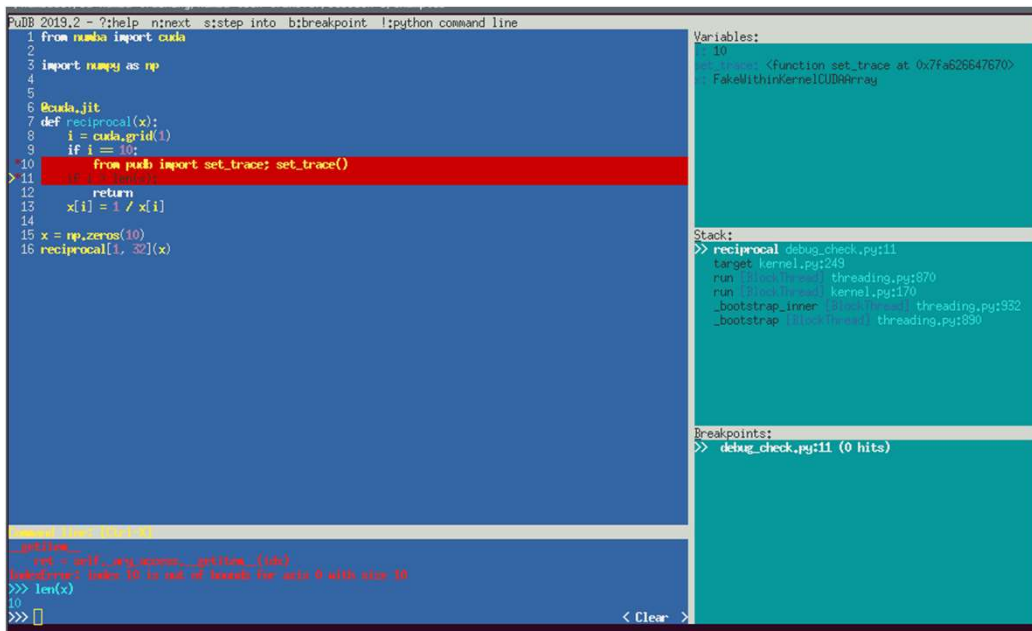
48

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
12    return
13    x[i] = 1 / x[i]
14
15 x = np.zeros(10)
16 reciprocal[1, 32](x)

```

Variables:

```

10
set_trace: <function set_trace at 0x7fa626647670>
FakeWithKernelCUDAArray

```

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
>>>

```

< Clear >

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