

# RAPIDS

## Numba for CUDA Programmers Session 4 - Extending Numba

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

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:

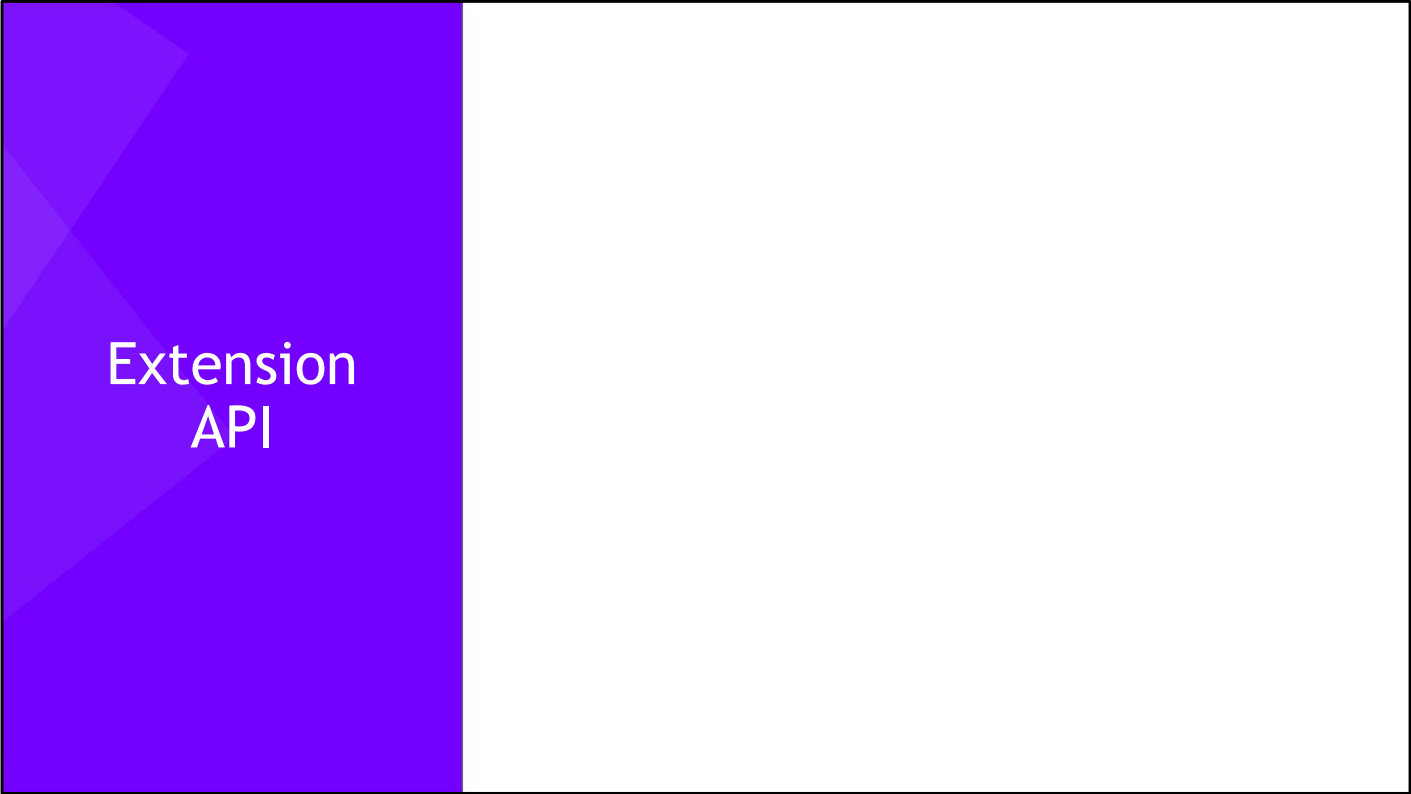
```
numba.core.errors.TypeError: Failed in nopython mode pipeline
Untyped global name 'Interval': cannot determine Numba type of
    <class 'type'>

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?

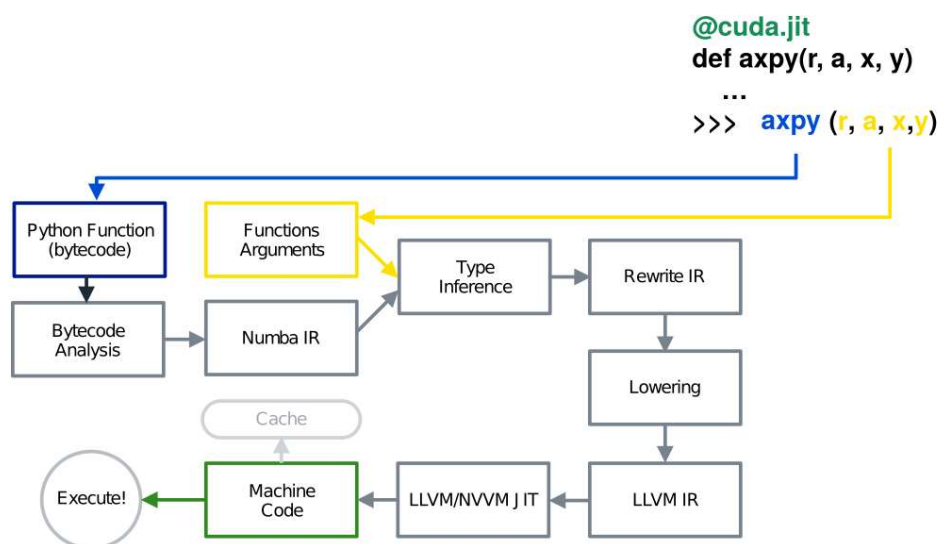
At the moment, we'll get a Typing error, because Numba doesn't know what to do with the Interval class.



Extension  
API

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

# Pipeline

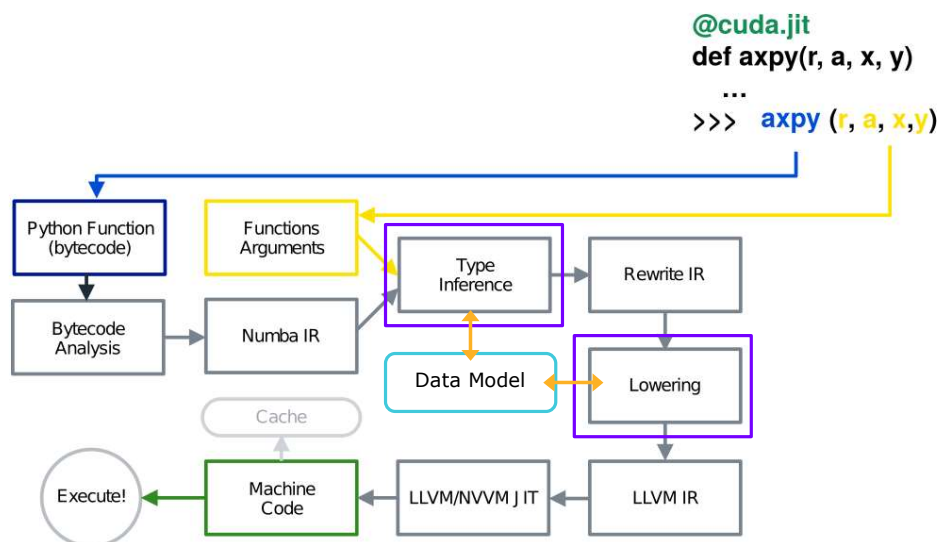


Let's just refresh our memory of Numba's pipeline, to remind us of what we're going to be working with.

- 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.cudadec1 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 1<sup>st</sup> (0) and 2<sup>nd</sup> (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**