NVIDIA

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

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

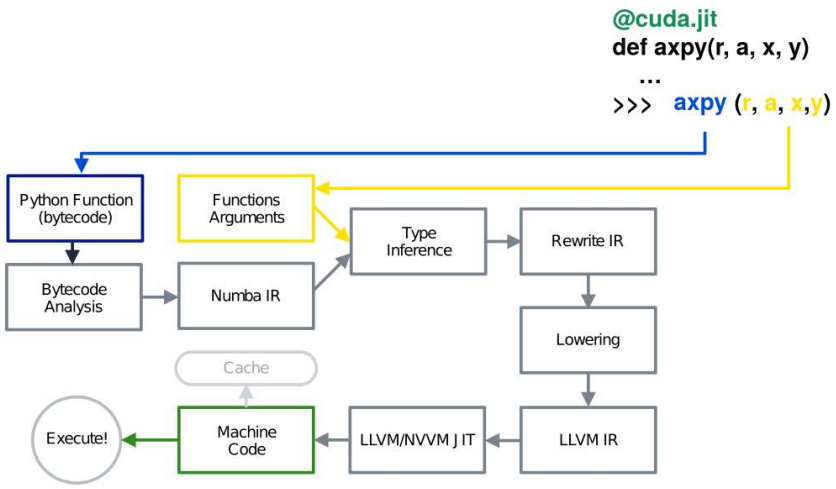
It's the biggest difference between C and Python with CUDA - to get to native code, you need to statically type everything at some point. However, Python has no static typing, unlike CUDA C.

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



RAPIDS 4

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
```


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
```


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
```


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
```


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 int32s 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 int32s and a float32.

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

Unification error

```
TypingError: 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
```


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

A TypingError 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:

```
TypingError: 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.
```


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)
```

```
TypingError: 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):
```


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
```


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)
```


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

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 elimiation 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)
```


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)
```


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;
```


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
```


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
```


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;
```


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
```


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

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")
```


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 cant 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

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

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
```


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(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

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

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