


Numba Cuda in Practice

To enable Cuda in Numba with conda just execute `conda install cudatoolkit` on the command line.

The Cuda extension supports almost all Cuda features with the exception of dynamic parallelism and texture memory. Dynamic parallelism allows to launch compute kernel from within other compute kernels. Texture memory has a caching pattern based on spatial locality. We will not go into detail of these here.

 Contents

[Finding out about Cuda devices](#)

[Launching kernels](#)

[Python features in Numba for Cuda](#)

[Memory management](#)

[Advanced features](#)

Finding out about Cuda devices

Let us first check what kind of Cuda device we have in the system.

```
from numba import cuda

cuda.detect()

Found 1 CUDA devices
id 0      b'Quadro RTX 3000'      [SUPPORTED]
        compute capability: 7.5
        pci device id: 0
        pci bus id: 1
Summary:
    1/1 devices are supported

True
```

Launching kernels

Launching a Cuda kernel from Numba is very easy. A kernel is defined by using the `@cuda.jit` decorator as

```
@cuda.jit
def an_empty_kernel():
    """A kernel that doesn't do anything."""
    # Get my current position in the global grid
    [pos_x, pos_y] = cuda.grid(2)
```

The type of the kernel is

```
an_empty_kernel

<numba.cuda.compiler.Dispatcher at 0x7f0dfc812d90>
```

In order to launch the kernel we need to specify the thread layout. The following commands define a two dimensional thread layout of 16×16 threads per block and 256×256 threads. In total this gives us 16, 777, 216 threads. This sounds huge. But GPUs are designed to launch large amounts of threads. The only restriction is that we are allowed to have at most 1024 threads in total (product of all dimensions) within a single thread block.

```
threadsperblock = (16, 16) # Should be a multiple of 32 if possible.
blockspergrid = (256, 256) # Blocks per grid
```

We can now launch all 16.8 million threads by calling

```
an_empty_kernel[blockspergrid, threadsperblock]()
```

Inside a kernel we can use the following commands to get the position of the thread.

```
@cuda.jit
def another_kernel():
    """Commands to get thread positions"""
    # Get the thread position in a thread block
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    tz = cuda.threadIdx.z

    # Get the id of the thread block
    block_x = cuda.blockIdx.x
    block_y = cuda.blockIdx.y
    block_z = cuda.blockIdx.z

    # Number of threads per block
    dim_x = cuda.blockDim.x
    dim_y = cuda.blockDim.y
    dim_z = cuda.blockDim.z

    # Global thread position
    pos_x = tx + block_x * dim_x
    pos_y = ty + block_y * dim_y
    pos_z = tz + block_z * dim_z

    # We can also use the grid function to get
    # the global position

    (pos_x, pos_y, pos_z) = cuda.grid(3)
    # For a 1-or 2-d grid use grid(1) or grid(2)
    # to return a scalar or a two tuple.

threadsperblock = (16, 16, 4) # Should be a multiple of 32 if possible.
blockspergrid = (256, 256, 256) # Blocks per grid

another_kernel[blockspergrid, threadsperblock]()
```

Python features in Numba for Cuda

Numba supports in Cuda kernels only a selected set of features that are supported by the Cuda standard. Not allowed are exceptions, context managers, list comprehensions and yield statements. Supported types are `int`, `float`, `complex`, `bool`, `None`, `tuple`. For a complete overview of supported features see <https://numba.pydata.org/numba-doc/dev/cuda/cudapysupported.html#>. Only a small set of Numpy functions are supported. Essentially, everything that does require dynamic memory management will not work due to the restrictions on kernels from the Cuda programming model.

Memory management

For simple kernels we can rely on Numba copying data to and from the device. For more complex code we need to manually manage buffers on the device.

Copy data to the device

```
import numpy as np

arr = np.arange(10)
device_arr = cuda.to_device(arr)
```

Copy data from the device back to the host

```
host_arr = device_arr.copy_to_host()
```

Copy into an existing array

```
host_array = np.empty(shape=device_arr.shape, dtype=device_arr.dtype)
device_arr.copy_to_host(host_array)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Generate a new array on the device

```
device_array = cuda.device_array((10,), dtype=np.float32)
```

Advanced features

Cuda has a number of advanced features that are supported by Numba. Some of them are:

- Pinned Memory is a form of memory allocation that allows much faster data transfer than standard buffers.
- Streams are a way to run multiple tasks on a GPU concurrently. By default, Cuda executes one command after another on the device. Streams allow us to create several concurrent queues for scheduling tasks onto the device. This allows for example to have a kernel stream that performs computations and a memory stream that does memory transfers, concurrently. One can use events to synchronize between different streams.
- Multiple devices are well supported by Numba. There exist helper routines to enumerate and select different devices.

For a full list of features check out the guide at <https://numba.pydata.org/numba-doc/latest/cuda/index.html>

By Timo Betcke

© Copyright 2020.