**“Introduction to GPU programming with Numba”**

"Numba is a just-in-time compiler for Python that works best on code that uses NumPy arrays and functions, and loops. For our purposes today, Numba is a python package that allows you to write python code for GPUs.

**%matplotlib inline**

**from matplotlib import pyplot as plt**

**import numpy as np**

**import math**

**from numba import jit, njit, vectorize, cuda**

**Accessing the GPU**

1) In order to run Numba functions using google's free GPUs, we have to do a couple of things. First, go to the Runtime menu, click on 'Change Runtime Type', and in the pop-up box, under 'Hardware Accelerator', select 'GPU'. Save the Runtime.

2) Ideally, that's all we should have to do. But in practice, even though the CUDA libararies are installed, as of this writing Colab can't find them automatically. So, we'll figure out where they are, and then point Colab to them.

**!find / -iname 'libdevice'**

**!find / -iname 'libnvvm.so'**

Paste the location of the libraries into the following code box (if it's different, otherwise you can just run the code):

**import os**

**os.environ['NUMBAPRO\_LIBDEVICE'] = "/usr/local/cuda-10.0/nvvm/libdevice"**

**os.environ['NUMBAPRO\_NVVM'] = "/usr/local/cuda-10.0/nvvm/lib64/libnvvm.so"**

**Vector Addition on GPUs**

The simplest way to access the GPU via Numba is to use a vectorized ufunc. A Numpy ufunc, or Universal Function, is a function that operates on vectors, or arrays. If we use Numba's vectorize decorator and specify the cuda target, Numba will automatically write a CUDA kernel for us and run the function on the GPU! Let's try it out:

**@vectorize(['int64(int64, int64)'], target='cuda')**

**def add\_ufunc\_gpu(x, y):**

**return x + y**

**x = np.arange(10)**

**y = 2 \* x**

**add\_ufunc\_gpu(x, y)**

Cool, it worked! But what actually just happened? Well, a lot of things. Numba automatically:

-Compiled a CUDA kernel to execute the ufunc operation in parallel over all the input elements.

-Allocated GPU memory for the inputs and the output.

-Copied the input data to the GPU.

-Executed the CUDA kernel with the correct kernel dimensions given the input sizes.

-Copied the result back from the GPU to the CPU.

-Returned the result as a NumPy array on the host.

Using the %timeit magic function, we can determine how fast the CUDA function is:

**%timeit add\_ufunc\_gpu(x, y)**

And compare it to a version compiled for the CPU:

**@vectorize(['int64(int64, int64)'], target='cpu')**

**def add\_ufunc\_cpu(x, y):**

**return x + y**

**%timeit add\_ufunc\_cpu(x, y)**

Writing Cuda Kernels While targeting ufuncs with the cuda syntax is the most straightforward way to access the GPU with Numba, it may not be flexible enough for your needs. If you want to write a more detailed GPU program, at some point you are probably going to need to write CUDA kernels.

As discussed in the lecture, the CUDA programming model allows you to abstract the GPU hardware into a software model composed of a grid containing blocks of threads. These threads are the smallest individual unit in the programming model, and they execute together in groups (traditionally called warps, consisting of 32 threads each). Determiming the best size for your grid of thread blocks is a complicated problem that often depends on the specific algorithm and hardware you're using, but here a few good rules of thumb:

-the size of a block should be a multiple of 32 threads, with typical block sizes between 128 and 512 threads per block.

-the size of the grid should ensure the full GPU is utilized where possible. Launching a grid where the number of blocks is 2x-4x the number of streaming multiprocessors on the GPU is a good starting place. (The Tesla K80 GPUs provided by Colaboratory have 15 SMs - more modern GPUs like the P100s on TigerGPU have 60+.)

-The CUDA kernel launch overhead does depend on the number of blocks, so it may not be best to launch a grid where the number of threads equals the number of input elements when the input size is very big. We'll show a pattern for dealing with large inputs below.

As a first example, let's return to our vector addition function, but this time, we'll target it with the cuda.jit decorator:

**@cuda.jit**

**def add\_kernel(x, y, out):**

**tidx = cuda.threadIdx.x # this is the unique thread ID within a 1D block**

**bidx = cuda.blockIdx.x  # Similarly, this is the unique block ID within the 1D grid**

**block\_dimx = cuda.blockDim.x  # number of threads per block**

**grid\_dimx = cuda.gridDim.x    # number of blocks in the grid**

**start = tidx + bidx \* block\_dimx**

**stride = block\_dimx \* grid\_dimx**

**# assuming x and y inputs are same length**

**for i in range(start, x.shape[0], stride):**

**out[i] = x[i] + y[i]**

That's a lot more typing than our ufunc example, and it is much more limited: it only works on 1D arrays, it doesn't verify input sizes match, etc. Most of the function is spent figuring out how to turn the block and grid indices and dimensions into unique offsets in the input arrays. The pattern of computing a starting index and a stride is a common way to ensure that your grid size is independent of the input size. The striding will maximize bandwidth by ensuring that threads with consecuitive indices are accessing consecutive memory locations as much as possible. Thread indices beyond the length of the input (x.shape[0], since x is a NumPy array) automatically skip over the for loop.

Let's call the function now on some data:

**n = 100000**

**x = np.arange(n).astype(np.float32)**

**y = 2 \* x**

**out = np.empty\_like(x)**

**threads\_per\_block = 128**

**blocks\_per\_grid = 30**

**add\_kernel[blocks\_per\_grid, threads\_per\_block](x, y, out)**

**print(out[:10])**

The calling syntax is designed to mimic the way CUDA kernels are launched in C, where the number of blocks per grid and threads per block are specified in the square brackets, and the arguments to the function are specified afterwards in parentheses.

Note that, unlike the ufunc, the arguments are passed to the kernel as full NumPy arrays. A thread within the kernel can access any element in the array it wants, regardless of its position in the thread grid. This is why CUDA kernels are significantly more powerful than ufuncs. (But with great power, comes a greater amount of typing...)

Numba has created some helper functions to cut down on the typing. We can write the previous kernel much more simply as:

**@cuda.jit**

**def add\_kernel(x, y, o ut):**

**start = cuda.grid(1)**

**stride = cuda.gridsize(1) # ditto**

**for i in range(start, x.shape[0], stride):**

**out[i] = x[i] + y[i]**

As before, using NumPy arrays forces Numba to allocate GPU memory, copy the arguments to the GPU, run the kernel, then copy the argument arrays back to the host. This not very efficient, so you will often want to pre-allocate device arrays.

**Memory Management**

Numba can automatically handle transferring data to and from the GPU for us. However, that's not always what we want. Sometimes we will want to perform several functions in a row on the GPU without transferring the data back to the CPU in between. To address this, Numba provides the to\_device function in the cuda module to allocate and copy arrays to the GPU:

**x\_device = cuda.to\_device(x)**

**y\_device = cuda.to\_device(y)**

**print(x\_device)**

**print(x\_device.shape)**

**print(x\_device.dtype)**

x\_device and y\_device are now Numba "device arrays" that are in many ways equivalent to Numpy ndarrays except that they live in the GPU's global memory, rather than on the CPU. These device arrays can be passed to Numba cuda functions just the way Numpy arrays can, but without the memory copying overhead.

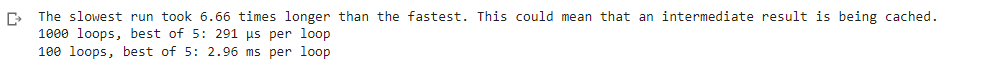
We can also create an output buffer on the GPU with the numba.cuda.device\_array() function:

out\_device = cuda.device\_array(shape=(n,), dtype=np.float32)

Now we can try out our Cuda kernel using the pre-copied device arrays, and compare the time to a version without moving the data first.

**%timeit add\_kernel[blocks\_per\_grid, threads\_per\_block](x\_device, y\_device, out\_device)**

**%timeit add\_kernel[blocks\_per\_grid, threads\_per\_block](x, y, out)**



As you can see, moving data back and forth is expensive. In general, you will want to keep your data on the GPU as long as possible and do as many calculations as you can before moving it back. Of course, at some point you will have to move the data back to the CPU, whether to output to a file, perform some other CPU functions, etc.. Numba provides the copy\_to\_host function for that:

out = out\_device.copy\_to\_host()

After all of this extra work, you may be wondering what the point of doing calculations on the GPU is. After all, if we compare the time it takes to run our fancy Cuda kernel with the pre-allocated GPU arrays, to the CPU version provided by python, we find that the CPU version is still faster:

**%timeit x + y**

But that's because we still haven't given the GPU enough work to do. If we go back and try the same problem, but with arrays that are 10 times larger...

**n = 1000000**

**x = np.arange(n).astype(np.float32)**

**y = 2 \* x**

**x\_device = cuda.to\_device(x)**

**y\_device = cuda.to\_device(y)**

**out\_device = cuda.device\_array(shape=(n,), dtype=np.float32)**

**%timeit add\_kernel[blocks\_per\_grid, threads\_per\_block](x\_device, y\_device, out\_device)**



**%timeit x + y**



...we start to see the power of the GPU. As you may have noticed, the GPU function called with a million array elements took approximately the same amount of time to run as the version with 100 thousand elements. That's because most of the GPU was not busy when we only had 100 thousand elements, so we still hadn't overcome the overhead of launching the kernel.