Fundamentals of Accelerated Computing with CUDA Python

<https://numba.pydata.org/numba-doc/dev/reference/pysupported.html>

<https://numpy.org/doc/stable/user/quickstart.html>

<https://docs.scipy.org/doc/numpy-1.15.1/reference/ufuncs.html>

<https://docs.scipy.org/doc/numpy-1.15.0/user/basics.broadcasting.html>

<https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.vectorize.html>

<https://numba.pydata.org/numba-doc/dev/reference/types.html>

<https://numba.pydata.org/numba-doc/dev/user/vectorize.html>

<http://numba.pydata.org/numba-doc/latest/cuda/cudapysupported.html>

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>

<https://numba.pydata.org/numba-doc/dev/cuda/memory.html>

<http://numba.pydata.org/numba-doc/latest/user/vectorize.html#the-guvectorize-decorator>

<http://numba.pydata.org/numba-doc/latest/cuda/ufunc.html#generalized-cuda-ufuncs>

<http://numba.pydata.org/numba-doc/dev/cuda/intrinsics.html#supported-atomic-operations>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#memory-hierarchy>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>

<https://numba.pydata.org/numba-doc/dev/cuda/memory.html#shared-memory-and-thread-synchronization>

<https://numba.pydata.org/numba-doc/dev/reference/types.html#numba-types>

Wow, the GPU is *a lot slower* than the CPU?? For the time being this is to be expected because we have (deliberately) misused the GPU in several ways in this example. How we have misused the GPU will help clarify what kinds of problems are well-suited for GPU computing, and which are best left to be performed on the CPU:

* **Our inputs are too small**: the GPU achieves performance through parallelism, operating on thousands of values at once. Our test inputs have only 4 and 16 integers, respectively. We need a much larger array to even keep the GPU busy.
* **Our calculation is too simple**: Sending a calculation to the GPU involves quite a bit of overhead compared to calling a function on the CPU. If our calculation does not involve enough math operations (often called "arithmetic intensity"), then the GPU will spend most of its time waiting for data to move around.
* **We copy the data to and from the GPU**: While in some scenarios, paying the cost of copying data to and from the GPU can be worth it for a single function, often it will be preferred to to run several GPU operations in sequence. In those cases, it makes sense to send data to the GPU and keep it there until all of our processing is complete.
* **Our data types are larger than necessary**: Our example uses int64 when we probably don't need it. Scalar code using data types that are 32 and 64-bit run basically the same speed on the CPU, and for integer types the difference may not be drastic, but 64-bit floating point data types may have a significant performance cost on the GPU, depending on the GPU type. Basic arithmetic on 64-bit floats can be anywhere from 2x (Pascal-architecture Tesla) to 24x (Maxwell-architecture GeForce) slower than 32-bit floats. If you are using more modern GPUs (Volta, Turing, Ampere), then this could be far less of a concern. NumPy defaults to 64-bit data types when creating arrays, so it is important to set the [dtype](https://docs.scipy.org/doc/numpy-1.14.0/reference/arrays.dtypes.html) attribute or use the [ndarray.astype()](https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.ndarray.astype.html) method to pick 32-bit types when you need them.

Given the above, let's try an example that is faster on the GPU by performing an operation with much greater arithmetic intensity, on a much larger input, and using a 32-bit data type.

**Please note:** Not all NumPy code will work on the GPU, and, as in the following example, we will need to use the math library's pi and exp instead of NumPy's. Please see [the Numba docs](https://numba.pydata.org/numba-doc/latest/reference/numpysupported.html) for extensive coverage of NumPy support on the GPU.

from numpy import exp

# Modify these 3 function calls to run on the GPU.

@vectorize(['float32(float32)'], target='cuda')

def normalize(grayscales):

return grayscales / 255

@vectorize(['float32(float32, float32)'], target='cuda')

def weigh(values, weights):

return values \* weights

@vectorize(['float32(float32)'], target='cuda')

def activate(values):

return ( math.exp(values) - math.exp(-values) ) / ( math.exp(values) + math.exp(-values) )

def create\_hidden\_layer(n, greyscales, weights, exp, normalize, weigh, activate):

device\_greyscales = cuda.to\_device(greyscales)

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

device\_weights = cuda.to\_device(weights)

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

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

normalize(device\_greyscales, out = out\_normalized)

weigh(out\_normalized, device\_weights, out = out\_weighted)

activate(out\_weighted, out = out\_activated)

activated = out\_activated.copy\_to\_host()

# The assessment mechanism will expect `activated` to be a host array, so,

# even after you refactor this code to run on the GPU, make sure to explicitly copy

# `activated` back to the host.

return activated

cuda-memcheck python debug/ex3a.py

cat -n debug/ex3a.py | grep -C 2 "17"

ERRATA:

The word NVIDIA in the first paragraph;

Not working link: The NumPy docs on gufuncs: <https://numpy.org/doc/stable/reference/c-api.generalized-ufuncs.html>