PyCuda exercise

Setting up PyCuda on ARC3

All CUDA code must run on a GPU node as with the previous exercises. You will need to load the following modules as there are currently some incompatibilities with later versions of the GNU compilers and CUDA libraries.

```
module load cuda/8.0.61
module load gnu/native
module load python
module load python-libs
```

It is possible, although not recommended, to get an interactive session on a GPU node. This command line snippet will request an interactive session on a P100 node for one hour.

```
qrsh -l ports=1,coproc_p100=1,h_rt=1:0:0 -pty y /bin/bash -i
```

Before you can use PyCuda, you have to import and initialize it:

```
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
```

Note that you do not have to use **pycuda.autoinit**. Initialization, context creation, and cleanup can also be performed manually, if desired.

Transferring Data

The next step in most programs is to transfer data onto the device. In PyCuda, you will mostly transfer data from numpy arrays on the host. (But indeed, everything that satisfies the Python buffer interface will work, even a str.) Let's make a 4x4 array of random numbers:

```
import numpy
a = numpy.random.randn(4,4)
```

a consists of double precision numbers (64 bit), but there is differing support for single precision and double precision floating point numbers across NVidia devices (see https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/ for a more detailed discussion of performance across sp and dp data). In essence, GPUs perform operations faster on sp (32 bit) numbers than on dp (64 bit) numbers. So let's convert to an appropropriate data type.

```
a = a.astype(numpy.float32)
```

Finally, we need somewhere to transfer data to, so we need to allocate memory on the device:

```
a_gpu = cuda.mem_alloc(a.nbytes)
```

As a last step, we need to transfer the data to the GPU:

```
cuda.memcpy_htod(a_gpu, a)
```

Executing a Kernel
For this exercise, we'll use something simple:

We will write code to double each entry in **a_gpu**. To this end, we write the corresponding CUDA C code, and feed it into the constructor of a **pycuda.compiler.SourceModule**. Although we can write the CPU code and GPU handler code in Python using PyCuda, the kernel must still be written in C/C++ with the CUDA extensions.

```
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
```

If there aren't any errors, the code is now compiled and loaded onto the device. We find a reference to our pycuda.driver. 'func' and call it, specifying **a_gpu** as the argument, and a block size of **4x4**:

```
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))
```

(We're assigning the function to the variable name **func** and then using that to call it and pass into it the arguments)

Finally, we retrieve the data back from the GPU and display it, together with the original a:

```
a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print (a_doubled)
print (a)
```

This will print something like this:

```
[[ 0.51360393  1.40589952  2.25009012  3.02563429]
 [-0.75841576 -1.18757617  2.72269917  3.12156057]
 [ 0.28826082 -2.92448163  1.21624792  2.86353827]
 [ 1.57651746  0.63500965  2.21570683 -0.44537592]]
[[ 0.25680196  0.70294976  1.12504506  1.51281714]
 [-0.37920788 -0.59378809  1.36134958  1.56078029]
 [ 0.14413041 -1.46224082  0.60812396  1.43176913]
 [ 0.78825873  0.31750482  1.10785341 -0.22268796]]
```

To try:

1. Execute this as a single Python script submitted through the compiler.
2. Try to convert one of the earlier exercises from today to a PyCUDA version.
    a. Do you get similar results?
    b. How could you profile this Python code?