Assignment 3 - GPU acceleration

Start Assignment

- Due No due date
- Points 20
- Submitting a text entry box or a file upload

Exercise 1:

Let's GPU accelerate a "zero suppression" function. A common operation when working with waveforms is to force all sample values below a certain absolute magnitude to be zero, as a way to eliminate low amplitude noise. Let's make some sample data:

```
# This allows us to plot right here in the notebook
%matplotlib inline
```

```
# Hacking up a noisy pulse train
from matplotlib import pyplot as plt

n = 100000
noise = np.random.normal(size=n) * 3
pulses = np.maximum(np.sin(np.arange(n) / (n / 23)) - 0.3, 0.0)
waveform = ((pulses * 300) + noise).astype(np.int16)
plt.plot(waveform)
```

```
def zero_suppress(waveform_value, threshold):
    if waveform_value < threshold:
        result = 0
    else:
        result = waveform_value
    return result
```

```
# This will throw an error until you successfully vectorize the `zero_suppress` function above.
# The noise on the baseline should disappear when zero_suppress is implemented
plt.plot(zero_suppress(waveform, 15))
```

Exercise 2: Optimize Memory Movement

Given these ufuncs:

```
import math
```

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

def make_pulses(i, period, amplitude):

    return max(math.sin(i / period) - 0.3, 0.0) * amplitude


n = 100000

noise = (np.random.normal(size=n) * 3).astype(np.float32)

t = np.arange(n, dtype=np.float32)

period = n / 23
```

As it currently stands in the cell below, there is an unnecessary data roundtrip back to the host and then back again to the device in between the calls to make_pulses and add_ufunc.

Update the cell below to use device allocations so that there is only one copy to device before the call to make_pulses and one copy back to host after the call to add_ufunc.

```
pulses = make_pulses(t, period, 100.0)

waveform = add_ufunc(pulses, noise)

%matplotlib inline

from matplotlib import pyplot as plt

plt.plot(waveform)
```

Exercise 3:

The following exercise will require you to utilize everything you've learned so far to GPU-accelerate neural network calculations. Unlike previous exercises, there will not be any solution code available to you. Just like in this section, the other 2 notebooks in this course also have assessment problems. For those of you who successfully complete all 3, you will receive a certificate of competency in the course.

- Please read the directions carefully before beginning your work to ensure the best chance at successfully completing the assessment.

Accelerate Neural Network Calculations

- You will be refactoring a simple version of some code that performs work needed to create a hidden layer in a neural network. It normalizes grayscale values, weighs them, and applies an activation function.
- Your task is to move this work to the GPU using the techniques you've learned while retaining the correctness of the calculations.

Load Imports and Initialize Values

You should not modify this code, it contains imports and initial values needed to do work on either the CPU or the GPU.

```
import numpy as np
from numba import cuda, vectorize

# Our hidden layer will contain 1M neurons.
# When you assess your work below, this value will be automatically set to 100M.
n = 1000000
```

```
greyscales = np.floor(np.random.uniform(0, 255, n).astype(np.float32))
weights = np.random.normal(.5, .1, n).astype(np.float32)


# As you will recall, `numpy.exp` works on the CPU, but, cannot be used in GPU implmentations.
# This import will work for the CPU-only boilerplate code provided below, but
# you will need to modify this import before your GPU implementation will work.
from numpy import exp


# Modify these 3 function calls to run on the GPU.
def normalize(grayscales):
    return grayscales / 255

def weigh(values, weights):
    return values * weights

def activate(values):
    return ( exp(values) - exp(-values) ) / ( exp(values) + exp(-values) )


# Modify the body of this function to optimize data transfers and therefore speed up performance.
# As a constraint, even after you move work to the GPU, make this function return a host array.
def create_hidden_layer(n, greyscales, weights, exp, normalize, weigh, activate):

    normalized = normalize(greyscales)
    weighted = weigh(normalized, weights)
    activated = activate(weighted)

  # 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
```