Author: Drew Afromsky

Key Resources:

- Programming Massively Parallel Processors, A Hands on Approach, Third Edition, David B. Kirk, Wen-mei W. Hewu;
- https://www.quora.com/Why-do-we-transpose-matrices

Email: daa2162@columbia.edu

Parallel Computing for Converting an RGB Image to Grayscale and Matrix Transpose Operation

RGB to Grayscale Conversion of an Image

- RGB to Grayscale conversion for an image is a common image processing technique, used in order to reduce the size of data, since colour information is sometimes not required. An efficient algorithm for this conversion is very useful, and it is a good demonstrator of the topic of accessing the elements of a matrix using 2-dimensional threads.

To convert the rgb value of a pixel to a grayscale value:

```
grayscale = 0.21*r + 0.71*g + 0.07*b
```

where *r* is the red intensity, *g* is the green intensity, and *b* is the blue intensity.

Introduction and Background Part I: In an RGB image representation, each pixel in an image is stored as a tuple of (r,g,b) values. The format of an image's row is (r g b) (r g b)… (r g b). each tuple specifies a mixture of R (red), G (green), and B (blue). For each pixel, the r, g, and b values represent the intensity (0 dark, 1 bright/full intensity) of the R, G, and B light sources when the pixel is rendered. To convert an RGB image to grayscale, we need to compute the luminance value for each pixel by applying the equation below (*grayscale*) to each r, g, b element of each pixel. None of these per pixel computations depends on each other and all of them can be performed independently. For the problem at hand, our input image is an RGB image of dimensions (384,512,3), where the 3 represents the color channels; the image can be considered a 3D tensor. After converting to grayscale, our new image has dimensions (384,512).

Process/Methods (PyCUDA): I began with writing the serial (Python) code for creating the grayscale image. This is comprised of reading in the color image, typecasting to float32, and implementing FOR-loops to apply the grayscale equation to each r, g, and b element of each pixel of the input image. I runt this 3 times so that I can average the run-times and get a more accurate estimation. I then transfer the resulting output image from the server and to my local, which can be found below. The input image is of size 384x512x3 and after grayscale conversion, becomes 384x512. The code is shown below:

```
rgb = plt.imread('/home/daa2162/color.png') # path to color input image on the server
rgb = rgb.astype(np.float32)


# Create the output array
```

```
py_output = np.zeros(shape=(rgb.shape[0],rgb.shape[1]))


# Serial (Python)
times = []
for e in range(3):
    start = time.time()
    for i in range(len(rgb)): # 0 to 383
        for j in range(len(rgb[i])): # 0 to 511
            grayscale = 0.21 * rgb[i][j][0] + 0.71 * rgb[i][j][1] + 0.07 * rgb[i][j][2]
            py_output[i][j] = grayscale
    times.append(time.time() - start)
py_times.append(np.average(times))



# Plotting on the server

MAKE_PLOT = True
if MAKE_PLOT:
    plt.figure()
    plt.imsave("./py_gray_scale_2.png",py_output, cmap="gray")
```

Input Color Image

Output Grayscale Image
(Python Serial Code)

Next, for the parallel processing task I uploaded the input color image to the server. Looking at our input image, we see that we will have 384*512 computations to perform (196,608, since we will be computing gray intensity for each R, G, B element of each pixel at once). To accomplish this, the code begins with creating a class and a function within that class with the input image as the input argument of that function. I first declare all of my host variables: (1.) the input image after it has been read, (2.) the dimensions of the input image, (3.) a temporary zeros array that will be filled with the intensity values after applying the grayscale equation, (4.) contiguous array of all R values for each pixel, (5.) contiguous array of G values for each pixel, and (6.) contiguous array of B values for each pixel. (4.-6.) arrays come from the input image. Next, I allocate memory on the device for the expected kernel function inputs – the grayscale values resulting from the R,G,B to grayscale conversion equation, as well as for the R,G,B arrays. Once memory has been allocated, I can copy or transfer the data from the host (CPU) to the device (GPU), using PyCuda's function: **cuda.memcpy_htod(*dest*, *src*).** Next, I write the kernel code using CUDA-C language. The inputs of the kernel code are the following: an array of all of the resulting grayscale intensity values after applying the grayscale conversion equation, the arrays (4.-6.) mentioned previously, as well as the first two dimensions of the color image (M X N = 384 X 512). More specifically, the inputs are float *gray, float *r, float *g, float *b, int M, and int N. Now, since we will be working in 2-dimensions (due to the input image dimensions) and that we must use more than one block (since each block can have a maximum of 1024 threads and we require ~200,000 threads), we must also have indices for both the x- and y-direction. Since we know the dimensions of our input data, we can calculate the block size and grid size. Our blocks will be 2-dimensional and we wish to maximize our resources/efficiency, so we will use the maximum amount of threads per block (1024). Therefore, the dimensions of our block are 32x32 which is equivalent to 1024 threads. Now, for each dimension of our input data we need to allocate our blocks/threads. We can obtain the grid size by taking our input dimension sizes and dividing by the size of the blocks (32). Therefore, to meet our # of threads requirements to compute in parallel we will need a grid size of 12 X 16 and block size of 32 X 32, which results in 1024 threads per block and 196,608 threads used in total, which is what we want since our input image is of size (384, 512, 3) and each thread is performing the grayscale equation on the arrays of R,G,B values for all the pixels in parallel. This is counter-intuitive since our input image is a 3D tensor (384,512,3), but since we are applying the grayscale equation to each R,G,B value for each pixel it makes since that after creating the arrays for R,G, and B values, each element of the input image will have a unique ID corresponding to that location of pixel, and we will have to index into that, where we find the intensity of R,G,B at that pixel location. Therefore, in the kernel code we create the two indices (in the x- and y-; rows and columns), and define the particular location of a pixel according to the # of columns (N) of the input image, and the x- and y- indices (pixel "ID" = [index in the x-]*N + [index in the y-]). This location value is used to index into each of the R,G,B arrays. After the kernel code has been created, I create a cuda event to measure time and call the kernel function, start a timer, and then run the function. Afterwards, I need to transfer those outputs to the CPU using **cuda.memcpydtoh(*dest*, *src*)** and synchronize my CUDA events. Finally, we return our grayscale array with all the intensity values and the run times for this computation.

Outside of the class, I define our input image and read it, using this as input for the class' function rgb2g. Beneath this is the code equality and verification code. After saving the python file, I use SCP terminal command to transfer the file to the server and sbatch with nvprof for profiling the execution of the code on the server. The figures below illustrate the output of the PyCuda program on the server.

Output Grayscale Image (PyCuda)

```
==8335== NVPROF is profiling process 8335, command: python pycuda_rgb2g.py
('Code Equality:', True)
('CUDA Times:', [0.000534570664167404])
('Serial Times:', [4.28216767311096])
('Speed-Up CUDA:', 8010.480110764129)
==8335== Profiling application: python pycuda_rgb2g.py
==8335== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 64.84%  630.44us         9  70.048us  68.864us  77.312us  [CUDA memcpy HtoD]
 19.33%  188.00us         3  62.666us  62.496us  62.784us  [CUDA memcpy DtoH]
 15.83%  153.92us         3  51.307us  51.104us  51.585us  rgb_2_g

==8335== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 64.66%  211.38ms         1  211.38ms  211.38ms  211.38ms  cuCtxCreate
 30.86%  100.90ms         1  100.90ms  100.90ms  100.90ms  cuCtxDetach
  1.17%  3.8300ms        12  319.16us  217.15us  553.04us  cuMemAlloc
  1.08%  3.5417ms         9  393.52us  365.26us  427.11us  cuMemcpyHtoD
  0.84%  2.7465ms        12  228.88us  179.11us  413.91us  cuMemFree
  0.65%  2.1177ms         3  705.89us  612.58us  788.11us  cuMemcpyDtoH
  0.49%  1.6045ms         3  534.82us  480.55us  615.66us  cuModuleLoadDataEx
  0.11%  371.34us         3  123.78us  115.82us  137.75us  cuModuleUnload
  0.07%  214.52us         3  71.505us  66.924us  76.556us  cuLaunchKernel
  0.03%  82.907us         6  13.817us  8.9500us  17.878us  cuEventRecord
  0.01%  38.012us         6  6.3350us  1.8780us  11.125us  cuEventCreate
  0.00%  16.275us         6  2.7120us     772ns  10.008us  cuCtxPopCurrent
  0.00%  15.580us         6  2.5960us     960ns  4.5580us  cuEventDestroy
  0.00%  13.852us         3  4.6170us  3.9340us  5.2000us  cuEventSynchronize
  0.00%  12.301us         6  2.0500us     489ns  9.3560us  cuCtxPushCurrent
  0.00%  10.068us         3  3.3560us  2.9670us  3.8430us  cuEventElapsedTime
  0.00%  6.8430us         3  2.2810us  2.0690us  2.5400us  cuFuncSetBlockShape
  0.00%  6.0340us         4  1.5080us  1.2050us  1.7750us  cuCtxGetDevice
  0.00%  5.8490us         3  1.9490us  1.8380us  2.0460us  cuModuleGetFunction
  0.00%  5.1690us         3  1.7230us     311ns  3.3780us  cuDeviceGetCount
  0.00%  4.8590us         3  1.6190us  1.5460us  1.7370us  cuDeviceComputeCapability
  0.00%  2.8610us         3     953ns     657ns  1.1820us  cuDeviceGet
  0.00%  2.1120us         4     528ns     366ns     805ns  cuDeviceGetAttribute
```

Slurm output of the PyCuda program showing code equality is true, the run-time in CUDA, and the serial code run-time, as well as the amount of speed up using CUDA versus Python.

PyCuda Code:

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

#################################
# author = Drew Afromsky        #
# email = daa2162@columbia.edu  #
#################################

import numpy as np
import time
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import pycuda.driver as cuda
from pycuda.compiler import SourceModule
import pycuda.autoinit

class RGB2GRAY:
    def rgb2gray(self, rgb_cpu):
        # rgb_cpu: an RGB image

        # Host variables
        self.color = rgb_cpu
        m,n,k = self.color.shape
        self.gray = np.zeros(shape=(self.color.shape[0],self.color.shape[1]), dtype=np.float32)
        self.r = np.ascontiguousarray(self.color[:,:,0]) # already dtype=np.float32 since rgb_cpu is rgb from below
        self.g = np.ascontiguousarray(self.color[:,:,1])
        self.b = np.ascontiguousarray(self.color[:,:,2])

        # Device memory allocation
        self.gray_d = cuda.mem_alloc(self.gray.nbytes)
        self.r_d = cuda.mem_alloc(self.r.nbytes)
        self.g_d = cuda.mem_alloc(self.g.nbytes)
        self.b_d = cuda.mem_alloc(self.b.nbytes)
```

```
# copy data to device
cuda.memcpy_htod(self.r_d, self.r)
cuda.memcpy_htod(self.g_d, self.g)
cuda.memcpy_htod(self.b_d, self.b)

# kernel
self.kernel_code_template = """
  #include <stdio.h>

  __global__ void rgb_2_g(float *gray, float *r, float *g, float *b, int M, int N)
  {
    // 2-D thread ID assuming more than one block will be executed
    int index_x = threadIdx.x + blockIdx.x * blockDim.x; // ROWS
    int index_y = threadIdx.y + blockIdx.y * blockDim.y; // COLUMNS

    // M, N = first 2 dimensions of the color image; index of each r,g,b pixel from color image
    int pixel = index_x * N + index_y;

    float r_element = r[pixel]; // intensity of red of that pixel
    float g_element = g[pixel];
    float b_element = b[pixel];

    gray[pixel] = 0.21 * r_element + 0.71 * g_element + 0.07 * b_element;

  }
"""
self.kernel_code = self.kernel_code_template % {
}
self.mod = SourceModule(self.kernel_code)

# create CUDA Event to measure time
start = cuda.Event() #pay attention here: this is the recommended method to record cuda running time
end = cuda.Event()

# function call
func = self.mod.get_function('rgb_2_g')
```

```
        start.record()
        start_ = time.time()
        # in CUDA block=(x,y,z), grid=(x,y,z)
        # maximum number of threads single block can have
        func(self.gray_d, self.r_d, self.g_d, self.b_d, np.int32(m), np.int32(n), block=(32, 32, 1), grid =
(np.int(np.ceil(float(m)/32)), np.int(np.ceil(float(n)/32)),1)) # In CUDA block=(x,y,z), grid=(x,y,z)
        end_ = time.time()
        end.record()

        # memory copy to host
        cuda.memcpy_dtoh(self.gray, self.gray_d)

        # CUDA Event synchronize
        end.synchronize()

        # return: the grayscale converter image
        return self.gray, start.time_till(end)*1e-3

if __name__ == "__main__":
    py_times = []
    cu_times = []

    # Create the input array
    # Change this path to the path of the image after using scp to transfer it the server
    # rgb = mpimg.imread('/Users/DrewAfromsky/Desktop/Fall 2019/EECS 4750- Heterogeneous Comp-Sign
Processing/Assignment 2/color.png')
    # rgb = mpimg.imread('/home/daa2162/color.png') # path to png on the server
    rgb = plt.imread('/home/daa2162/color.png') # path to png on the server
    rgb = rgb.astype(np.float32)

    # Create the output array
    py_output = np.zeros(shape=(rgb.shape[0],rgb.shape[1])) # Python
    cu_output = None # CUDA

    # Create instance for CUDA
    module = RGB2GRAY()
```

```python
# Serial (Python)
times = []
for e in range(3):
    start = time.time()
    for i in range(len(rgb)): # 0 to 383
        for j in range(len(rgb[i])): # 0 to 511
            grayscale = 0.21 * rgb[i][j][0] + 0.71 * rgb[i][j][1] + 0.07 * rgb[i][j][2]
            py_output[i][j] = grayscale
    times.append(time.time() - start)
    # Display and save the figure
    # plt.imshow(py_output, cmap = plt.get_cmap('gray'))
    # plt.savefig('/Users/DrewAfromsky/Desktop/Fall 2019/EECS 4750- Heterogeneous Comp-Sign
Processing/Assignment 2/serial_grayscale.png')
py_times.append(np.average(times))

# CUDA
times = []
for e in range(3):
    cu_output, t = module.rgb2gray(rgb)
    times.append(t)
cu_times.append(np.average(times))

print("Code Equality:", np.allclose(py_output, cu_output))
print("CUDA Times:", cu_times)
print("Serial Times:", py_times)
print("Speed-Up CUDA:", py_times[0]/cu_times[0])

# Optional: if you want to plot the function, set MAKE_PLOT to
# True:
MAKE_PLOT = True
if MAKE_PLOT:
    plt.figure()
    plt.imsave("./gray_scale_CUDA.png", cu_output, cmap="gray")
    plt.imsave("./py_gray_scale.png",py_output, cmap="gray")
```

Process/Methods (PyOpenCL): The process for OpenCL is similar to that of PyCuda with some adjustments in the code. The code begins with selecting the desired OpenCL platform. Next I setup a command queue and enable profiling. As before, I declare my host variables and allocate device memory (this is the same as with PyCuda). Next, the kernel code was written, which slightly varies from the kernel code for PyCuda, since our indexing for threading has different syntax (OpenCL syntax). Otherwise, our inputs are the same as well as the other operations and definitions to retrieve the grayscale intensities. Next, the kernel code is built, and the kernel function is called using the same inputs as PyCuda, while specifiying the global and local to determine how many threads of execution are run. Since the kernel only accesses the global work item ID, we can set the local size to 'None'. The launch of the kernel, the function call, and time measurement recordings look like this:

```
    self.kernel_code = self.kernel_code_template % {
    }
    self.prg = cl.Program(self.ctx, self.kernel_code).build()

    # function call
    func = self.prg.rgb_2_g

    start = time.time()
    evt = func(self.queue, self.r_d.shape, None, self.gray_d.data, self.r_d.data, self.g_d.data, self.b_d.data,
np.uint32(m), np.uint32(n))
    evt.wait()
    end = time.time()
    time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL running time
```

Now, we need to return the output on the device to our host (CPU) using the .get() function for OpenCL. The function of the class returns the grayscale intensity values in an array as well as the run-time for OpenCL. The verification code for PyOpenCL is the same structure as that of PyCuda. The outputs after running the job on the server can be seen below:

Output Grayscale Image
(PyOpenCL)

```
('Code Equality:', True)
('OpenCL Times:', [7.717333333333334e-05])
('Serial Times:', [4.359029769897461])
('Speed-Up OpenCL:', 56483.62694234789)
```

PyOpenCL Code:

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

#################################
# author = Drew Afromsky       #
# email = daa2162@columbia.edu  #
#################################

import numpy as np
import time
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import pyopencl as cl
import pyopencl.array
import matplotlib.image as mpimg

class RGB2GRAY:
    def rgb2gray(self, rgb_cpu):
        # rgb_cpu: an RGB image
        # return: the grayscale converter image
        NAME = 'NVIDIA CUDA'
        platforms = cl.get_platforms()
        devs = None
        for platform in platforms:
            if platform.name == NAME:
                devs = platform.get_devices()

        # Set up a command queue:
        self.ctx = cl.Context(devs)
        self.queue = cl.CommandQueue(self.ctx, properties=cl.command_queue_properties.PROFILING_ENABLE)

        # host variables
```

```
self.color = rgb_cpu
m,n,k = self.color.shape
self.gray = np.zeros(shape=(self.color.shape[0],self.color.shape[1]), dtype=np.float32)
self.r = np.ascontiguousarray(self.color[:,:,0]) # already dtype=np.float32 since rgb_cpu is rgb from below
self.g = np.ascontiguousarray(self.color[:,:,1])
self.b = np.ascontiguousarray(self.color[:,:,2])

# device memory allocation
self.gray_d = cl.array.to_device(self.queue, self.gray)
self.r_d = cl.array.to_device(self.queue, self.r)
self.g_d = cl.array.to_device(self.queue, self.g)
self.b_d = cl.array.to_device(self.queue, self.b)

# kernel
self.kernel_code_template = """
  __kernel void rgb_2_g(__global float *gray, __global float *r, __global float *g, __global float *b, int M, int N)
  {
    int id_x = get_global_id(0); // x-direction
    int id_y = get_global_id(1); // y-direction

    // M, N = first 2 dimensions of the color image; index of each r,g,b pixel from color image
    int pixel = id_x * N + id_y;

    float r_element = r[pixel]; // intensity of red of that pixel
    float g_element = g[pixel];
    float b_element = b[pixel];

    gray[pixel] = 0.21 * r_element + 0.71 * g_element + 0.07 * b_element;

  }
"""

self.kernel_code = self.kernel_code_template % {
}
self.prg = cl.Program(self.ctx, self.kernel_code).build()

# function call
```

```
        func = self.prg.rgb_2_g

        start = time.time()
        evt = func(self.queue, self.r_d.shape, None, self.gray_d.data, self.r_d.data, self.g_d.data, self.b_d.data,
np.uint32(m), np.uint32(n))
        evt.wait()
        end = time.time()
        time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL running time

        # memory copy to host
        self.gray = self.gray_d.get()

        return self.gray, time_

if __name__ == "__main__":
    py_times = []
    cl_times = []

    # Create the input array
    # Change this path to the path of the image after using scp to transfer it the server
    # rgb = mpimg.imread('/Users/DrewAfromsky/Desktop/Fall 2019/EECS 4750- Heterogeneous Comp-Sign
Processing/Assignment 2/color.png')
    # rgb = mpimg.imread('/home/daa2162/color.png') # path to png on the server
    rgb = plt.imread('/home/daa2162/color.png') # path to png on the server
    rgb = rgb.astype(np.float32)

    # Create the output array
    py_output = np.zeros(shape=(rgb.shape[0],rgb.shape[1])) # Python
    cl_output = None # OpenCL

    # Create instance for OpenCL
    module = RGB2GRAY()

    # Serial (Python)
    times = []
    for e in range(3):
        start = time.time()
```

```
        for i in range(len(rgb)): # 0 to 383
            for j in range(len(rgb[i])): # 0 to 511
                grayscale = 0.21 * rgb[i][j][0] + 0.71 * rgb[i][j][1] + 0.07 * rgb[i][j][2]
                py_output[i][j] = grayscale
        times.append(time.time() - start)
        # Display and save the figure
        # plt.imshow(py_output, cmap = plt.get_cmap('gray'))
        # plt.savefig('/Users/DrewAfromsky/Desktop/Fall 2019/EECS 4750- Heterogeneous Comp-Sign
Processing/Assignment 2/serial_grayscale.png')
    py_times.append(np.average(times))

    # OpenCL
    times = []
    for e in range(3):
        cl_output, t = module.rgb2gray(rgb)
        times.append(t)
    cl_times.append(np.average(times))

    print("Code Equality:", np.allclose(py_output, cl_output))
    print("OpenCL Times:", cl_times)
    print("Serial Times:", py_times)
    print("Speed-Up OpenCL:", py_times[0]/cl_times[0])

    # # Optional: if you want to plot the function, set MAKE_PLOT to
    # # True:
    MAKE_PLOT = True
    if MAKE_PLOT:
        plt.figure()
        plt.imsave("./gray_scale_OpenCL.png", cl_output, cmap="gray")
        plt.imsave("./py_gray_scale_2.png",py_output, cmap="gray")
```

Part I Discussion: After successfully showing code equality I observed the run-times for serial code, PyCuda, and PyOpenCL (~4.3 sec, 0.0005 sec, and 7.7x10-5 sec, respectively). It is obvious that parallel processing would significantly decrease the run time for converting RGB image to grayscale, but it is interesting that the PyCuda was able to perform this action 8,010x faster than python serial code and PyOpenCL 56,000x faster than python serial code. This also means that the OpenCL code was roughly 7x faster than CUDA code.

1.2 Matrix Transpose

- Implement a serial transpose algorithm. The input is a 2D matrix of any size, the output is the transpose of the input.
- Implement two parallel transpose algorithms, using PyCUDA and PyOpenCL. The input is a 2D matrix of any size, the output is the transpose of the input.

- Choose any two integers M and N from 1 to 10. Then, randomly generate matrices with sizes M x N, 2M x 2N, 3M x 3N,…. Calculate their transpose using 3 transpose algorithms (2 parallel, 1 serial) respectively. Record the running time of each call for each of the algorithm.
- Plot running time vs. matrix size in one figure, compare and analyze the results. Plot the graph only for kernel execution in each of PyCUDA and PyOpenCL implementation

Introduction and Background Part 1.2: The significance of transposing a matrix is mainly to represent linear transformations such as rotation and scaling. By taking the transpose of a matrix that represents a linear transformation, properties of the transformation can be revealed. If the transpose of a matrix and the original matrix are equivalent, for example, then the transformation is a symmetric transformation and the matrix is symmetric. (https://www.quora.com/Why-do-we-transpose-matrices)

Process/Methods:

(Serial/Python) We take a 2D matrix of any size and transpose it, which means the rows become the columns and the columns become the rows. The input matrix begins as an M X N, where M=5, and N=7, and gradually increases in size according to the instructions – 2M X 2N, 3M X 3N,…,10M X 10N. The code for the serial/python component of this assignment is below:

```
# Create the input array
matrix = np.float32(np.random.randint(low=0, high=255, size=(M*(itr+1),N*(itr+1))))

# Create the output array
py_output = np.zeros(shape=(N*(itr+1),M*(itr+1)), dtype=np.float32) # Python
# Serial (Python)
times = []
for e in range(3):
  start = time.time()
  for i in range(matrix.shape[1]): # 0 to N*(itr+1)
    for j in range(matrix.shape[0]): # 0 to M*(itr+1)
      py_output[i][j]= matrix[j][i]
  times.append(time.time() - start)
py_times.append(np.average(times))
```

(PyCuda): The structure of the code for PyCuda parallel processing is as follows:

(1.) Declare host variables

   a. Input matrix, shapes of this matrix, and a zeros matrix that will be the shape of the desired output and filled with the values calculated on the device

(2.) Allocate memory on the device/GPU for the input matrix and the output matrix

(3.) Create the kernel code/function

   a. Same indices as the rgb_2_g kernel code (index_x and index_y), however, since we the output matrix changes in dimension we have to be careful about filling the correct values in the output matrix, according to the appropriate index value of the input array. For example, when we need to map a value in the (1,1) position of the input array to the (1,1) position of the output array, we need to access the correct index value of that array element after the input array has been flattened to a 1-D array. The kernel code can be seen below:

   b. We have 32x32 threads per block, and block size is 32x32, and grid size is calculated according to the size of the input array in the M dimension divided by 32 and the same is done in the N dimension.

```
# kernel
    self.kernel_code_template = """
      #include <stdio.h>


      __global__ void mat_trans(float *T, const float *mat, int P, int L)
      {
        // 2-D thread ID assuming more than one block will be executed
        int index_x = threadIdx.x + blockIdx.x * blockDim.x; // ROWS
        int index_y = threadIdx.y + blockIdx.y * blockDim.y; // COLUMNS

        // index of the input array; L=columns
        int el = index_x * L + index_y;

        // index of the output array (transposed); P = rows
        int out = index_y * P + index_x;
        if(index_x < P && index_y < L){
          T[out] = mat[el];
        }
```

```
    }
"""
```

(4.) Similar as before, we launch and call the kernel code declaring the block and grid size, after transferring input data from the host (CPU) to the device (GPU).
(5.) To retrieve our desired output, the transposed matrix, we must then transfer the data on the GPU back to the CPU.
(6.) Finally, we check that the output for both serial code and PyCuda are the same, and expect their run times to be different. The outputs after running the code on the server are below:

```
('Code Equality:', True)
('py_time:', 7.724761962890625e-05)
('cu_time:', 0.0006686399877071381)
cu_output:
[[217.  99.  98. 254.  93.]
 [135. 237.  90. 168. 236.]
 [ 80.  22.  29.  41.  50.]
 [132. 136. 230. 203. 246.]
 [120. 233. 216. 156.  98.]
 [156. 235.  65. 227.  82.]
 [ 71. 247. 140. 119.  85.]]
py_output
[[217.  99.  98. 254.  93.]
 [135. 237.  90. 168. 236.]
 [ 80.  22.  29.  41.  50.]
 [132. 136. 230. 203. 246.]
 [120. 233. 216. 156.  98.]
 [156. 235.  65. 227.  82.]
 [ 71. 247. 140. 119.  85.]]
()
('Code Equality:', True)
('py_time:', 0.0002467632293701172)
('cu_time:', 0.00034548266728719076)
cu_output:
[[177.   6.  24. 140.  41. 221.  52.  32.  47.  78.]
 [192. 108. 161. 250. 235. 132.  26. 194. 201. 236.]
 [  0.  78. 118. 248.  18. 142. 210. 151. 188. 247.]
 [230. 233. 168. 131. 242.  96. 214. 254.  28. 183.]
 [200. 135.   4.  21.  31. 188. 162. 229. 198.  80.]
 [157. 127. 243.   0. 142. 189. 145. 154. 127. 208.]
 [240. 244. 231.   9. 150. 226.  58. 123. 214. 167.]
 [182.  59.   5. 170.   3.  30.  73.  29.  12.  33.]
 [ 43. 115. 184.  25. 252.  86. 163. 149. 241. 219.]
 [109.  76. 157. 249. 166. 181. 195. 182.  69. 150.]
 [ 50.  82.  89. 219. 202. 221.  34.  64. 193.  24.]
 [  5.   4.  77. 239.  35.  25.  75. 184. 133.  28.]
 [ 41. 170.  89. 202. 208.  65. 247. 219. 116. 199.]
 [ 63.  44. 149. 151.  29. 248. 239. 143. 121. 250.]]
py_output
[[177.   6.  24. 140.  41. 221.  52.  32.  47.  78.]
 [192. 108. 161. 250. 235. 132.  26. 194. 201. 236.]
 [  0.  78. 118. 248.  18. 142. 210. 151. 188. 247.]
 [230. 233. 168. 131. 242.  96. 214. 254.  28. 183.]
 [200. 135.   4.  21.  31. 188. 162. 229. 198.  80.]
 [157. 127. 243.   0. 142. 189. 145. 154. 127. 208.]
 [240. 244. 231.   9. 150. 226.  58. 123. 214. 167.]
 [182.  59.   5. 170.   3.  30.  73.  29.  12.  33.]
 [ 43. 115. 184.  25. 252.  86. 163. 149. 241. 219.]
 [109.  76. 157. 249. 166. 181. 195. 182.  69. 150.]
```

```
 ...
 [215. 246.  62. ...  88. 224.   8.]
 [132. 185.  71. ... 110. 211. 163.]
 [249.  13.  17. ... 157. 183.  14.]]
py_output
[[ 49. 249.   3. ...  16. 254.  33.]
 [ 73.  84.  44. ...  95.  79. 203.]
 [245. 144.  82. ...  75. 159. 204.]
 ...
 [215. 246.  62. ...  88. 224.   8.]
 [132. 185.  71. ... 110. 211. 163.]
 [249.  13.  17. ... 157. 183.  14.]]
()
('Code Equality:', True)
('py_time:', 0.005355993906656901)
('cu_time:', 0.0003809173405170441)
cu_output:
[[242. 182.  25. ...  32.  16. 206.]
 [ 97. 242.   4. ... 138. 173.  66.]
 [104.  96.  42. ...  48. 151. 134.]
 ...
 [ 76. 168. 107. ...  54.  52. 187.]
 [215. 227.  10. ...  32. 119. 219.]
 [251. 194. 197. ... 215.   7. 125.]]
py_output
[[242. 182.  25. ...  32.  16. 206.]
 [ 97. 242.   4. ... 138. 173.  66.]
 [104.  96.  42. ...  48. 151. 134.]
 ...
 [ 76. 168. 107. ...  54.  52. 187.]
 [215. 227.  10. ...  32. 119. 219.]
 [251. 194. 197. ... 215.   7. 125.]]
()
```

PyCuda Code:

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

####################################
# author = Drew Afromsky        #
# email = daa2162@columbia.edu  #
####################################

import numpy as np
import time
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt

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

class Transpose:
    def transpose(self, matrix):
        # a_cpu: a 2D matrix.

        # Host Variables
        self.matrix = matrix
        M, N = self.matrix.shape
        self.T = np.zeros((N, M), dtype=np.float32)

        # Device memory allocation
        self.transpose_d = cuda.mem_alloc(self.T.nbytes)
        self.matrix_d = cuda.mem_alloc(self.matrix.nbytes)

        # kernel
        self.kernel_code_template = """
            #include <stdio.h>
```

```
    __global__ void mat_trans(float *T, const float *mat, int P, int L)
    {
        // 2-D thread ID assuming more than one block will be executed
        int index_x = threadIdx.x + blockIdx.x * blockDim.x; // ROWS
        int index_y = threadIdx.y + blockIdx.y * blockDim.y; // COLUMNS

        // index of the input array; L=columns
        int el = index_x * L + index_y;

        // index of the output array (transposed); P = rows
        int out = index_y * P + index_x;
        if(index_x < P && index_y < L){
            T[out] = mat[el];
        }

    }
"""

self.kernel_code = self.kernel_code_template % {
}
self.mod = SourceModule(self.kernel_code)

# create CUDA Event to measure time
start = cuda.Event() #pay attention here: this is the recommended method to record cuda running time
end = cuda.Event()

# copy data to device
cuda.memcpy_htod(self.matrix_d, self.matrix)

# function call
func = self.mod.get_function('mat_trans')
start.record()
start_ = time.time()
func(self.transpose_d, self.matrix_d, np.int32(M), np.int32(N), block=(32, 32, 1), grid =
(np.int(np.ceil(float(M)/32)), np.int(np.ceil(float(N)/32)),1)) # In CUDA block=(x,y,z), grid=(x,y,z)
end_ = time.time()
end.record()
```

```
        # memory copy to host
        cuda.memcpy_dtoh(self.T, self.transpose_d)

        # CUDA Event synchronize
        end.synchronize()

        return self.T, start.time_till(end)*1e-3


if __name__ == "__main__":
    iteration = 10
    M = 5
    N = 7
    n = np.arange(0, iteration, 1) # np.arange(start,stop,step)
    py_times = []
    cu_times = []

    for itr in range(iteration):

        # Create the input array
        matrix = np.float32(np.random.randint(low=0, high=255, size=(M*(itr+1),N*(itr+1))))

        # Create the output array
        py_output = np.zeros(shape=(N*(itr+1),M*(itr+1)), dtype=np.float32) # Python
        cu_output = None # CUDA

        # Create instance for CUDA
        module = Transpose()

        # Serial (Python)
        times = []
        for e in range(3):
            start = time.time()
            for i in range(matrix.shape[1]): # 0 to N*(itr+1)
                for j in range(matrix.shape[0]): # 0 to M*(itr+1)
                    py_output[i][j]= matrix[j][i]
```

```
        times.append(time.time() - start)
    py_times.append(np.average(times))

    # CUDA
    times = []
    for e in range(3):
        cu_output, t = module.transpose(matrix)
        times.append(t)
    cu_times.append(np.average(times))

    print("Code Equality:", np.allclose(py_output, cu_output))
    print("py_time:", py_times[itr])
    print("cu_time:", cu_times[itr])
    print("cu_output:")
    print(cu_output)
    # print("Original:")
    # print(matrix)
    print("py_output")
    print(py_output)
    print()

# Optional: if you want to plot the function, set MAKE_PLOT to
# True:
MAKE_PLOT = True
if MAKE_PLOT:
    plt.gcf()
    plt.plot((M*n + N*n), py_times,'r', label="Python") # matrix size versus python run times
    plt.plot((M*n + N*n), cu_times,'g', label="CUDA") # matrix size versus CUDA run times
    plt.legend(loc='upper left')
    plt.title('Matrix Transpose')
    plt.xlabel('Matrix Size')
    plt.ylabel('output coding times(sec)')
    plt.gca().set_xlim((min(M*n + N*n), max(M*n + N*n)))
    plt.savefig('plots_pycuda.png')
```

PyOpenCL: The process for OpenCL is similar to that of PyCuda with some adjustments in the code. The code begins with selecting the desired OpenCL platform. Next I setup a command queue and enable profiling. As before, I declare my host variables and allocate device memory (this is the same as with PyCuda). Next, the kernel code was written, which slightly varies from the kernel code for PyCuda, since our indexing for threading has different syntax (OpenCL syntax). Otherwise, our inputs are the same as well as the other operations and definitions to get the matrix transpose. Next, the kernel code is built, and the kernel function is called using the same inputs as PyCuda, while specifiying the global and local to determine how many threads of execution are run. Since the kernel only accesses the global work item ID, we can set the local size to 'None'. The launch of the kernel, the function call, and time measurement recordings look like this:

```
# kernel
    self.kernel_code_template = """
      __kernel void mat_trans(__global float *T, __global const float *mat, int P, int L)
      {
        // 2-D thread ID assuming more than one block will be executed
        int index_x = get_global_id(0); // ROWS
        int index_y = get_global_id(1); // COLUMNS

        // index of the input array; L=columns
        int el = index_x * L + index_y;

        // index of the output array (transposed); P = rows
        int out = index_y * P + index_x;

        if(index_x < P && index_y < L)
        {
          T[out] = mat[el];
        }

      }
    """

    self.kernel_code = self.kernel_code_template % {
    }
    self.prg = cl.Program(self.ctx, self.kernel_code).build()
```

```
    # function call
    func = self.prg.mat_trans

    start = time.time()
    evt = func(self.queue, self.matrix.shape, None, self.transpose_d.data, self.matrix_d.data, np.uint32(M),
np.uint32(N))
    evt.wait()
    end = time.time()
    time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL running time
```

Now, we need to return the output on the device to our host (CPU) using the .get() function for OpenCL. The function of the class returns the transposed matrices as well as the run-time for OpenCL. The verification code for PyOpenCL is the same structure as that of PyCuda. The outputs after running the job on the server can be seen below:

```
('Code Equality:', True)
('py_time:', 5.070368448893229e-05)
('cl_time:', 3.8378666666666666e-05)
cu_output:
[[179. 219.  85. 155.  70.]
 [ 95. 156.  48.  42. 223.]
 [124.  26. 236.  57. 181.]
 [ 52. 229. 137. 182.  27.]
 [130.  85.  62.  20. 111.]
 [ 12.  30.   6. 144. 118.]
 [155. 112. 253. 185. 132.]]
py_output
[[179. 219.  85. 155.  70.]
 [ 95. 156.  48.  42. 223.]
 [124.  26. 236.  57. 181.]
 [ 52. 229. 137. 182.  27.]
 [130.  85.  62.  20. 111.]
 [ 12.  30.   6. 144. 118.]
 [155. 112. 253. 185. 132.]]
()
('Code Equality:', True)
('py_time:', 0.00016299883524576822)
('cl_time:', 2.9781333333333334e-05)
cu_output:
[[183. 249. 114.   0. 181.  31. 185. 132. 253. 126.]
 [ 91. 196. 188.  40.  13. 113.  93.  58. 141.   6.]
 [145. 177. 170. 179. 161.  50.  93. 240. 250. 247.]
 [202.  23. 196.  50.  90. 193. 248. 150.  87.  95.]
 [ 82. 127. 206. 230.  82. 199.  97. 227.  83.   6.]
 [ 18. 128. 231.  42. 148.  83.   3. 174.  60. 218.]
 [ 65. 164.  90.  11. 106. 128. 217. 160. 226.  52.]
 [ 48. 199. 105. 184. 190.  39. 197. 185. 231.  56.]
 [103. 156. 182.  20. 198. 215.  42. 148.  85. 241.]
 [130.  88. 219. 132.  15.  84.  79.  85.  92.  65.]
 [229.  18.  77.  92. 133. 136. 125. 105. 140.  34.]
 [226.   2. 139. 109. 243. 175. 180. 188.  75.  27.]
 [105.  35. 202. 113. 228. 115.  78.  11. 196. 231.]
 [191.  68.  77.  76. 254. 226. 203.  77. 170. 131.]]
py_output
[[183. 249. 114.   0. 181.  31. 185. 132. 253. 126.]
 [ 91. 196. 188.  40.  13. 113.  93.  58. 141.   6.]
 [145. 177. 170. 179. 161.  50.  93. 240. 250. 247.]
 [202.  23. 196.  50.  90. 193. 248. 150.  87.  95.]
 [ 82. 127. 206. 230.  82. 199.  97. 227.  83.   6.]
 [ 18. 128. 231.  42. 148.  83.   3. 174.  60. 218.]
 [ 65. 164.  90.  11. 106. 128. 217. 160. 226.  52.]
 [ 48. 199. 105. 184. 190.  39. 197. 185. 231.  56.]
 [103. 156. 182.  20. 198. 215.  42. 148.  85. 241.]
 [130.  88. 219. 132.  15.  84.  79.  85.  92.  65.]
```

```
 ...
 [125. 132. 108. ... 220. 144.   3.]
 [227.  79. 219. ...  43. 133. 145.]
 [133. 183. 225. ...  98. 228. 152.]]
py_output
[[100. 145.  29. ... 239. 136.  27.]
 [205. 145. 184. ... 115. 220. 127.]
 [ 55.  85. 106. ... 233. 173. 133.]
 ...
 [125. 132. 108. ... 220. 144.   3.]
 [227.  79. 219. ...  43. 133. 145.]
 [133. 183. 225. ...  98. 228. 152.]]
()
('Code Equality:', True)
('py_time:', 0.00358430544535319)
('cl_time:', 3.0709333333333334e-05)
cu_output:
[[ 28. 122.  23. ... 152.  83. 135.]
 [173. 194. 156. ... 185.  65.  64.]
 [167. 242.   4. ... 151. 174. 237.]
 ...
 [128. 122. 102. ... 124.  51. 183.]
 [188.  76.  65. ... 218.  79.  74.]
 [  3.  47. 129. ...  48.   8.  56.]]
py_output
[[ 28. 122.  23. ... 152.  83. 135.]
 [173. 194. 156. ... 185.  65.  64.]
 [167. 242.   4. ... 151. 174. 237.]
 ...
 [128. 122. 102. ... 124.  51. 183.]
 [188.  76.  65. ... 218.  79.  74.]
 [  3.  47. 129. ...  48.   8.  56.]]
()
```

PyOpenCL Code:

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

##################################
# author = Drew Afromsky        #
# email = daa2162@columbia.edu  #
##################################

import numpy as np
import time
import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt
import pyopencl as cl
import pyopencl.array

class Transpose:
    def transpose(self, matrix):
        # a_cpu: a 2D matrix.
        # return: the transpose of a_cpu
        NAME = 'NVIDIA CUDA'
        platforms = cl.get_platforms()
        devs = None
        for platform in platforms:
            if platform.name == NAME:
                devs = platform.get_devices()

        # Set up a command queue:
        self.ctx = cl.Context(devs)
        self.queue = cl.CommandQueue(self.ctx, properties=cl.command_queue_properties.PROFILING_ENABLE)

        # Host Variables
        self.matrix = matrix
        M, N = self.matrix.shape
        self.T = np.zeros((N, M), dtype=np.float32)
```

```
# Device memory allocation
self.transpose_d = cl.array.to_device(self.queue, self.T)
self.matrix_d = cl.array.to_device(self.queue, self.matrix)

# kernel
self.kernel_code_template = """
    __kernel void mat_trans(__global float *T, __global const float *mat, int P, int L)
    {
        // 2-D thread ID assuming more than one block will be executed
        int index_x = get_global_id(0); // ROWS
        int index_y = get_global_id(1); // COLUMNS

        // index of the input array; L=columns
        int el = index_x * L + index_y;

        // index of the output array (transposed); P = rows
        int out = index_y * P + index_x;

        if(index_x < P && index_y < L)
        {
            T[out] = mat[el];
        }

    }
"""

self.kernel_code = self.kernel_code_template % {
}
self.prg = cl.Program(self.ctx, self.kernel_code).build()

# function call
func = self.prg.mat_trans

start = time.time()
evt = func(self.queue, self.matrix.shape, None, self.transpose_d.data, self.matrix_d.data, np.uint32(M),
np.uint32(N))
```

```
        evt.wait()
        end = time.time()
        time_ = 1e-9 * (evt.profile.end - evt.profile.start) #this is the recommended way to record OpenCL running time

        # memory copy to host
        self.T = self.transpose_d.get()

        return self.T, time_

if __name__ == "__main__":
    iteration = 10
    M = 5
    N = 7
    n = np.arange(0, iteration, 1) # np.arange(start,stop,step)
    py_times = []
    cl_times = []

    for itr in range(iteration):

        # Create the input array
        matrix = np.float32(np.random.randint(low=0, high=255, size=(M*(itr+1),N*(itr+1))))

        # Create the output array
        py_output = np.zeros(shape=(N*(itr+1),M*(itr+1)), dtype=np.float32) # Python
        cl_output = None # OpenCL

        # Create instance for OpenCL
        module = Transpose()

        # Serial (Python)
        times = []
        for e in range(3):
            start = time.time()
            for i in range(matrix.shape[1]): # 0 to N*(itr+1)
                for j in range(matrix.shape[0]): # 0 to M*(itr+1)
                    py_output[i][j]= matrix[j][i]
            times.append(time.time() - start)
```

```
    py_times.append(np.average(times))

    # OpenCL
    times = []
    for e in range(3):
        cl_output, t = module.transpose(matrix)
        times.append(t)
    cl_times.append(np.average(times))

    print("Code Equality:", np.allclose(py_output, cl_output))
    print("py_time:", py_times[itr])
    print("cl_time:", cl_times[itr])
    print("cu_output:")
    print(cl_output)
    # print("Original:")
    # print(matrix)
    print("py_output")
    print(py_output)
    print()

# Optional: if you want to plot the function, set MAKE_PLOT to
# True:
MAKE_PLOT = True
if MAKE_PLOT:
    plt.gcf()
    plt.plot((M*n + N*n), py_times,'r', label="Python") # matrix size versus python run times
    plt.plot((M*n + N*n), cl_times,'g', label="OpenCL") # matrix size versus CUDA run times
    plt.legend(loc='upper left')
    plt.title('Matrix Transpose')
    plt.xlabel('Matrix Size')
    plt.ylabel('output coding times(sec)')
    plt.gca().set_xlim((min(M*n + N*n), max(M*n + N*n)))
    plt.savefig('plots_pyOpenCL.png')
```

Part II Discussion: After successfully showing code equality I observed the run-times for serial code, PyCuda, and PyOpenCL. For the second part of the assignment, the serial code was taking between 10 and 20 times as long to complete the task when comparing the first and last matrix transpose operations , whereas Cuda and OpenCL remained pretty steady and much faster than Python alone. It is obvious that parallel processing would significantly decrease the run time for matrix transpose operation, but it is interesting that the PyOpenCL was able to perform this computation between 100-1000 times faster than python alone. PyCuda seemed to only speed up the computation by a factor of 10 at most. It appears that OpenCL is more effective at both converting an rgb image to grayscale, as well as performing a matrix transpose operation on any 2-D matrix.

Note: Many of the ideas mentioned early in this assignment under methods for PyCuda RGB to Gray, are similar and relevant for PyOpenCL as well as PyCuda for matrix transpose, and due to the current length of the assignment submission, I felt that it was redundant and unnecessary to re-discuss some of these ideas.