Programming Assignment 4 - CUDA

Due Date: Tues. 11/15

Using CUDA at Ohio Supercomputer Center

The OSC clusters are equipped with Tesla K40 GPUs. Some relevant stats for the K40:

```
Total amount of global memory:                 12288 MBytes (12884705280 bytes)
(15) Multiprocessors, (192) CUDA Cores/MP:     2880 CUDA Cores
GPU Clock rate:                                876 MHz (0.88 GHz)
Memory Clock rate:                             3004 Mhz
Memory Bus Width:                              384-bit
L2 Cache Size:                                 1572864 bytes
Total amount of constant memory:               65536 bytes
Total amount of shared memory per block:       49152 bytes
Total number of registers available per block: 65536
Warp size:                                     32
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

To use CUDA on the OSC cluster, you must allocate a node with an attached GPU. To interactively allocate such a node, use:

$ **qsub -I -l walltime=0:59:00 -l nodes=1:gpus=1**
(qsub -EYE -ell walltime ... -ell nodes ...).

To ensure the best resource availability for everyone, please only log on to a GPU host node when you are ready compile and run, then please exit when you are not actively testing.

To compile and test your programs you will need to load the CUDA environment:

$ **module load cuda**

and then use the Nvidia compilers. For example:

$ **nvcc -O -o lab4p1 jones_jeffrey_lab4p1.cu**

The "-O" flag sets the compiler to the default level (3), which the "-o lab4p1" flag, specifies the name for the the executable file, "lab4p1," which you can then execute by name:

$ **lab4p1**

Note that compilation (use the nvidia compiler, nvcc) can be performed on the login nodes, and does not require a node with a GPU. You will need to load the cuda module on the login node if you wish to do this. You will not be able to test your programs successfully on the login nodes, as they have no GPUs.

Nvidia CUDA drivers available free on-line

If your laptop/desktop has an Nvidia graphics card, you can download the CUDA drivers directly from Nvidia for your own local development and testing. Please see `https://developer.nvidia.com/cuda-downloads`. Nvidia's "CUDA Zone" also provides a wide array of tools and documentation: `https://developer.nvidia.com/cuda-zone`.

Part 1

Create both serial and CUDA parallel programs based on the following code segment, which multiplies the transpose of a matrix with itself:

```
double A[4096][4096], C[4096][4096];
// insert code to initialize matrix elements to random values between 1.0 and 2.0
for (i = 0; i < 4096; i++)
    for (j = 0; j < 4096; j++)
        for (k = 0; k < 4096; k++)
            C[ i ][ j ] += A[ k ][ i ] * A[ k ][ j ];
```

Use the code as above for your **Serial** version on OSC using 1 node with 12 processors (full node). Use whatever techniques you feel appropriate to design a **Parallel** version.
a) Report your results in estimated GFlops.
b) Measure both serial and parallel performance.
c) Report the CUDA compute structure (Grid, Block and Thread) you used and explain your results.

Part 2

Implement both serial and CUDA program to perform Sobel operator for edge detection, on a given set of images.

Background

The Sobel operator performs a 2-D spatial gradient measurement on images. The Sobel edge detector uses a pair of 3 x 3 stencils, or "convolution masks," one estimating gradient in the x-direction and the other estimating gradient in y-direction. The Sobel detector is incredibly sensitive to noise in pictures, it effectively highlights them as edges.

Sobel Operator Description

An image gradient is a change in intensity (or color) of an image. An edge in an image occurs when the gradient is greatest and the Sobel operator makes use of this fact to find the edges in an image. The Sobel operator calculates the approximate image gradient of each pixel by “convolution" of the image with a pair of 3x3 filters. These filters estimate the gradients in the horizontal (x) and vertical (y) directions. The magnitude of the gradient is simply the sum of these 2 gradients.

Gx:

-1	0	+1
-2	0	+2
-1	0	+1

Gy:

+1	+2	+1
0	0	0
-1	-2	-1

The magnitude of gradient is calculated using

$$G = \sqrt{Gx^2 + Gy^2}$$

Determining threshold for Pixel Classification

To perform Sobel edge detection the gradient magnitude is computed for each pixel (excluding the pixels in boundary), with the pixel being classified as white or black based on comparing the gradient to a threshold. Below are the steps to follow

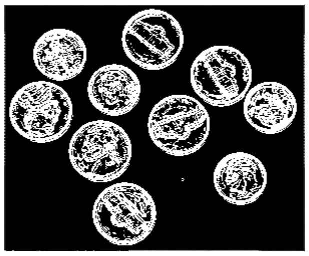
- Take the input image (N x M) pixels
- Iterate over each pixel (excluding the first/last row and first/last column pixels)
- Determine the Magnitude G of new pixel, from Gx and Gy where:
  - G_x is the sum of the products of the G_x stencil multiplied by the corresponding pixel values that align with the stencil when the stencil is centered on the current pixel (that is, the center element of the stencil corresponds to the current pixel).
  - G_y is computed similarly by employing the G_y stencil.
  - For each pixel, G(pixel) = $\sqrt{G_x^2 + G_y^2}$
- Use a threshold for classifying the pixel as black or white, if the magnitude is greater than threshold assign white(255), else black(0)
- The Resultant image (N x M) containing new magnitude will be a black & white image with explicit edges
- **Note:** Assume that the boundary pixels are simply copied to new image without any modifications

Our convergence criterion for this experiment is to achieve a image having **greater than 75% of percentage pixels being black**. We iterate over different threshold values starting at 0 and incrementing by 1%, until this convergence is achieved. Below is the logic flow

```
threshold = 0
while(black_cell_count < 0.75 * num_of_pixels)
{
     iterate over all pixels
    {
         G = sobel_output(pixel)
         if(G > threshold)
             sobel_image[pixel] = white (255)
         else
             sobel_image[pixel] = black (0)
             black_cell_count++
    }

    increment_threshold
}
```

Input Image:

Sobel Output Image: 

Reading/Writing Images

Your program will work on 24-bit bmp style image format. To help you with reading and writing images, a bmp reader support library will be provided to you. You will be provided with bmp_reader.o and read_bmp.h file with the following support API calls:

- **void* read_bmp_file(FILE *bmp_file)**:
  Will take in a FILE* pointing to the input image file and will return a buffer pointer (void *). The buffer returned will contain the pixel values arranged in linear fashion running column first. i.e. if your image is N x M pixels. The buffer will have M pixels of first row followed by M pixels of 2nd row and so on. Note: You have to open the image file in 'rb' mode into the FILE* before calling this function. Ensure to free the buffer*, returned by the function before exiting the program.
- **void write_bmp_file(FILE *out_file, uint8_t *bmp_data)**:
  Will take in FILE* pointing to output image file and buffer pointer containing the pixel values arranged in linear fashion. Note: The output file should be opened in 'wb' mode into FILE* before calling write_bmp_file. Ensure to free the buffer* before exiting the program
- Both these functions are in class bmp_image, along with 3 other useful information, which you can use to build your program.
  - image_width : Will contain width of image or no of pixel columns, after read API is issued.
  - image_height: Will hold the height or no of pixel rows in an image, after read API is issued.
  - num_pixel : Will be equal to image width x height.

Serial Code for Sobel operator with Classification convergence

Use the below code as a reference for your serial version of code

```
//Read the binary bmp file into buffer
bmp_data = (uint8_t *)img1.read_bmp_file(file_name);

//Allocate new output buffer of same size
new_bmp_img = (uint8_t *)malloc(img1.num_pixel);

//Get image attributes
wd = img1.image_width();    ht = img1.image_height();

//Convergence loop
threshold = 0;
while(black_cell_count < (75*wd*ht/100))
{

    black_cell_count = 0;
    threshold += 1;
    for(i=1; i < (ht-1); i++)
    {
        for(j=1; j < (wd-1); j++)
        {
            Gx = bmp_data[ (i-1)*wd + (j+1) ] - bmp_data[ (i-1)*wd + (j-1) ] \
                    + 2*bmp_data[ (i)*wd + (j+1) ] - 2*bmp_data[ (i)*wd + (j-1) ] \
                        + bmp_data[ (i+1)*wd + (j+1) ] - bmp_data[ (i+1)*wd + (j-1) ];

                Gy = bmp_data[ (i-1)*wd + (j-1) ] + 2*bmp_data[ (i-1)*wd + (j) ] \
                        + bmp_data[ (i-1)*wd + (j+1) ] - bmp_data[ (i+1)*wd + (j-1) ] \
                        - 2*bmp_data[ (i+1)*wd + (j) ] - bmp_data[ (i+1)*wd + (j+1) ];

                mag = sqrt(Gx * Gx + Gy * Gy);
                if(mag > threshold)
                {
                    new_bmp_img[ i*wd + j] = 255;
                }else{
                    new_bmp_img[ i*wd + j] = 0;
                    black_cell_count++;
                }
             }
         }
     }

//Write back the new bmp image into output file
write_bmp_file(out_file, new_bmp_img);
```

Instrumentation

- Use the instructions and example code above to develop both serial and CUDA parallel version.
  Note: In CUDA version the entire convergence need not be parallel, try your best to optimize the sobel operator and do a serial loop for threshold convergence.
- Include read_bmp.h into your program file and link the object file to compile in your make file.
- The necessary sample images and program file will be uploaded to the /class/cse5441 directory.
- Your program should take the following command line parameters:
  ./a.out <input image file.bmp> <serial processd image.bmp> <cuda processd image.bmp>
- Your program should output
  a) Time taken for serial execution
  b) Time taken for cuda execution
  c) Threshold obtained in serial vs cuda version
  Results can adhere to format below but not restricted:

```
./indresh_sira_lab4p2.out image_1.bmp serial_image.bmp cuda_image.bmp
*******************************************************************
Image Info::
           Height=3658       Width=2962
Time taken for serial sobel operation: 12.0457 sec
Threshold during convergence: 69

Time taken for CUDA sobel operation: 1.0457 sec
Threshold during convergence: 69
*******************************************************************
```

Reporting

Run your program against all the sample images provided using instructions specified above. Be sure to provide the following

- Provide a timing and threshold convergence summary for all images
- Explain your cuda organization (grid, block, thread) distribution
- Did you see any performance improvement in using GPU?
  Support your answer with numbers from your observation.

Submitting Results

Generally, follow the submission guidelines for the previous labs, with the following specifics:

- Create submission directory name "cse5441_lab4." and place your files in it.
- Ensure you provide read,write and executable permission to the folder, else it will not be graded.
- name your program files <lastname>_<firstname>_lab4p1.cu and <lastname>_<firstname>_lab4p2.cu.
- provide a single make file that will name your executables lab4p1 and lab4p2
- **submit a printed copy** of your report in class on Thursday, 11/10. Be sure to include in your report your name and section number as well as all relevant CUDA parameter settings you used.