Assignment #3

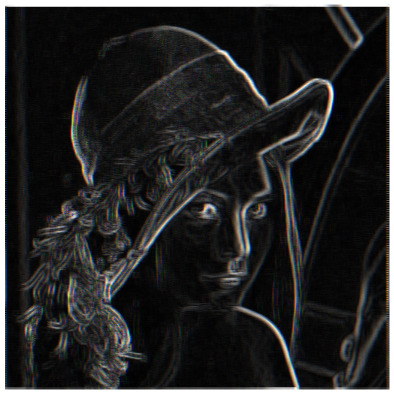
CS 398 GPGPU Programming, Fall 2020

Due Date:	As specified on the moodle
Topics covered:	Tiled convolution, shared memory, constant cache/memory, loop unrolling
Deliverables:	The project files (release version only) including source code and the report `A3Report.pdf`. These files should be put in a folder and subsequently zipped according to the stipulations set out in the course syllabus. If you follow the reference code given on the moodle, you need only to include the source code of kirsch_gpu.cu.
Objectives:	Implementation of Kirsch edge detection in CUDA C. Learn how to use shared memory, constant cache/memory, tiled convolution, and loop unrolling to write more efficient kernel functions.

Implementing the Kirsch Edge Detection Filter

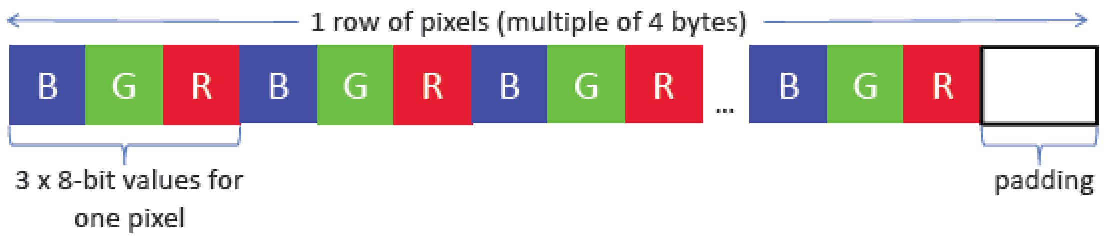
The aim of this assignment is to implement a GPU version of the Kirsch Filter, which is a kind of filter used in image processing for edge detection. The purpose of edge detection is to identify points in the image where there are discontinuities or major shifts in colour change (hence detecting the "edges" of the image). This is useful for applications such as feature extraction or face detection etc. For example, the following pictures show the effect of edge detection.

Before Edge Detection


Kirsch Edge Detection

The BMP file format

The 24-bit colour bitmap file consists of two parts. A 54-byte header that contains information such as image height and width, and pixel array that contains the colour information of each pixel. Every row in the pixel array has the following format:

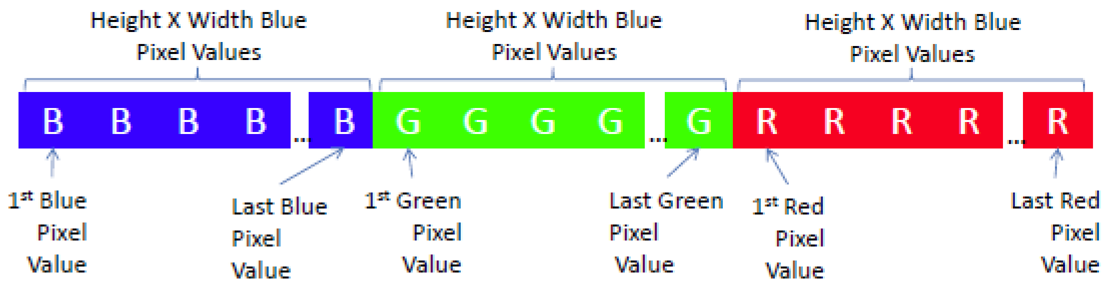

1 row of pixels (multiple of 4 bytes)
B
G
R
B
G
R
B
G
R
...
B
G
R
3 x 8-bit values for one pixel
padding

Each pixel contains 3 8-bits value for Blue, Green and Red respectively. Each of these 8-bits value should be considered as an unsigned byte ranging between 0 to 255 in values. Different combinations of the RGB values determine the colour of the particular pixel. The number of pixels in a row is determined by the width of the picture. For example, if the picture has a width of 50 pixels, the pixel row will be $50 \times 3B + 2B = 152B$. The additional 2 bytes of padding was including to make entire row a multiple of 4 bytes. Of course, the height of a picture will determine the number of rows that we have.

However, the original format of the file is not convenient for data manipulation. We have included a function in the file bmp.cpp that has the following two functions:

```
void bmp_read(char *filename, bmp_header *header, unsigned char **data);
void bmp_write(char *filename, bmp_header *header, unsigned char *data);
```

The `bmp_read` takes in a filename and fill in the header information into the header object passed by reference into the function. It also allocates an array for the data variable and fills it in with a reformatted pixel information in one single array. In particular, the array is presented in the following form:



All the paddings have been removed and now instead, we have a single array instead. The bmp write function performs the reverse of the bmp read function and writes a given single array into the bitmap file with the correct format. Therefore, the following code simply copies the bitmap file from one to another:

```
bmp_header h;
unsigned char *data;

bmp_read("in.bmp", &h, &data );
bmp_write("out.bmp", &h, &data);
```

The definition of the bmp header and the functions are given in the files bmp.cpp and bmp.h. You may turn on interleave option by defining COLOR_INTERLEAVE as shown in bmp.cpp.

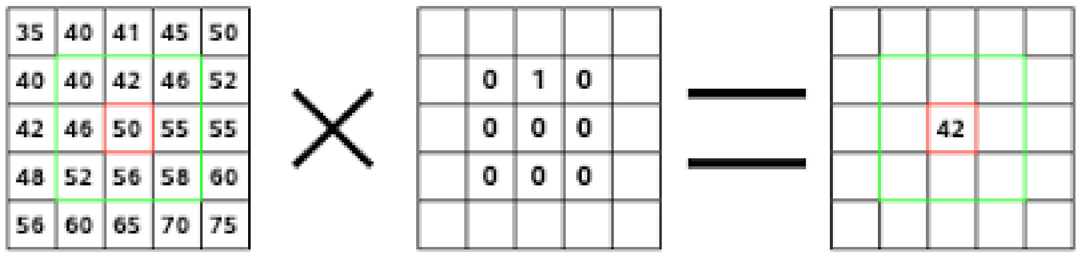
Kirsch Edge Detection Algorithm

At the heart of the Kirsch edge detection algorithm is one mask that is rotated 8 times and applied. The following are 8 masks that are used in Kirsch edge detection:

$$g^{(1)} = \begin{pmatrix} +5 & +5 & +5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix}, g^{(2)} = \begin{pmatrix} +5 & +5 & -3 \\ +5 & 0 & -3 \\ -3 & -3 & -3 \end{pmatrix}, g^{(3)} = \begin{pmatrix} +5 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & -3 & -3 \end{pmatrix}, g^{(4)} = \begin{pmatrix} -3 & -3 & -3 \\ +5 & 0 & -3 \\ +5 & 5 & -3 \end{pmatrix},$$

$$g^{(5)} = \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ +5 & +5 & +5 \end{pmatrix}, g^{(6)} = \begin{pmatrix} -3 & -3 & -3 \\ -3 & 0 & +5 \\ -3 & 5 & 5 \end{pmatrix}, g^{(7)} = \begin{pmatrix} -3 & -3 & +5 \\ -3 & 0 & +5 \\ -3 & -3 & +5 \end{pmatrix}, g^{(8)} = \begin{pmatrix} -3 & +5 & +5 \\ -3 & 0 & +5 \\ -3 & -3 & -3 \end{pmatrix}.$$

The filter works by applying the masks as convolution matrices on every 3×3 pixels in the image. The following image shows how the convolution works in general:



We apply the mask to each 3×3 squares of the whole image in a similar fashion to how we apply the mask for the above. For all the convolution results after applying the eight Kirsch edge detection masks, we take the maximum convolution value to be the final value of the center pixel. The value of the ghost cell/pixel is regarded as 0 when the boundary pixels are computed. The student is invited to consult `kirsch_cpu.cpp` to find out more details. Please note that this is different from Assignment 2 in CS315.

The Assignment

In the given file `kirsch_cpu.cpp`, we have a basic implementation of the kirsch edge detection in a function with the following prototype:

```
void kirschEdgeDetectorCPU(
    const unsigned char *data_in,
    const int *mask,
    unsigned char *data_out,
    const unsigned channels,
    const unsigned width,
    const unsigned height
)
```

Please read the function and understand the code in the light of what was described in the previous section.

You are supposed to use CUDA C to optimize the basic implementation. Two empty functions, `convolution` and `convolution2`, is supposed to be filled in by you in a kirsch_gpu.cu file, where the declaration of `convolution` is as follows:

```
__global__ void convolution(unsigned char *I,
                            const int *__restrict__ M,
                            unsigned char *P,
                            int channels,
                            int width,
                            int height)
```

and `convolution2` has the similar declaration.

Please consider the following hints:

- There are two design options as shown in our lecture notes. You are required to implement both options.
    1. Design 1: The size of each thread block matches the size of an output tile. All threads participate in calculating output elements. You should define a shared memory area that is (TILE_WIDTH + Mask_width - 1) $\times$ (TILE_WIDTH + Mask_width - 1) and thread block size is TILE_WIDTH $\times$ TILE_WIDTH. Assume the shared memory area size is smaller than 2$\times$ TILE_WIDTH $\times$ TILE_WIDTH. Mask_width is the width of convolution mask, which is 3. This should be implemented as `convolution`.
    2. Design 2: The size of each thread block matches the size of an input tile. Each thread loads one input element into the shared memory with the size of BLOCK_WIDTH $\times$ BLOCK_WIDTH. BLOCK_WIDTH should be O_TILE_WIDTH + MASK_width - 1, where O_TILE_WIDTH is the output tile width. This should be implemented as `convolution2`.
- You should associate the 2D coordinates of the thread with the correct location of the shared memory area.
- The operations must be done at least in 16 bits to accommodate for overflow/underflow arithmetic.
- You may use blockIdx.z to associate with each channel.

Please compare the timings of your code under Microsoft Visual Studio Release mode. You may use the following utility to collect the measurements.

```
nvprof --metrics achieved_occupancy yourApp.exe [yourApp argslist]
nvprof --metrics gld_throughput yourApp.exe [yourApp argslist]
nvprof --metrics gst_throughput yourApp.exe [yourApp argslist]
nvprof --metrics gld_efficiency yourApp.exe [yourApp argslist]
nvprof --metrics gst_efficiency yourApp.exe [yourApp argslist]
```

where *achieved_occupancy* is defined as the ratio of the average active warps per cycle to the maximum number of warps supported on an SM. *gld_efficiency* is defined as

$$gld\_efficiency = \frac{RequestedGlobalMemoryLoadThroughput}{RequiredGlobalMemoryLoadThroughput} \tag{1}$$

Similarly, *gst_efficiency* is defined as

$$gst\_efficiency = \frac{RequestedGlobalMemoryStoreThroughput}{RequiredGlobalMemoryStoreThroughput} \tag{2}$$

Rubrics

This assignment will be graded over 100 points. Here is the breakdown:

- If your code fails to compile, zero is awarded immediately.
- If you did not use tiled convolution and shared/constant memory to implement the optimized kirsch edge detection, zero is awarded immediately.
- Comments/Code Readability - 10 points. Comments are important for CUDA C code to enhance readability. Otherwise, C code cannot be easily maintained.
- Correctness - 30 points. You must ensure that the code works for images of all sizes and the output image of CPU and GPU version should be the same.
- Speedup - 40 points. If your code is correct and there's speedup larger than 5 but less than 10, you will still get 10 out of 40 points here. Close to at least 60 times speedup on the computers (with NVIDIA GTX 1060 6GB) in the lab is required for full credit here (assume 512×512 bmp format color image).
- Report - 20 points. You should include into the report the results of latency for both CPU and GPU version (two design options). Report the achieved occupancy, load-/store throughput, and load/store efficiency for GPU version. Answer the following question for each design option of GPU version. Assume 512×512 bmp format color image for the following questions.

  1. How many multiplication/addition operations are being performed in your convolution kernel? explain.
  2. How many global memory reads are being performed by your kernel? explain.
  3. How many global memory writes are being performed by your kernel? explain.
  4. What is the minimum, maximum, and average number of real operations that a thread will perform? Real operations are those that directly contribute to the final output value.
  5. How much time is spent as an overhead cost for using the GPU for computation? Consider all code executed within your host function with the exception of the kernel itself, as overhead. How does the overhead scale with the size of the input?
  6. What do you think happens as you increase the mask size (say to 1024) while you set the block dimensions to 16x16? What do you end up spending most of your time doing? Does that put other constraints on the way you'd write your algorithm (think of the shared/constant memory size)?

7. Do you have to have a separate output memory buffer? Put it in another way, why can't you perform the convolution in place?
8. What is the best parameter for block size?