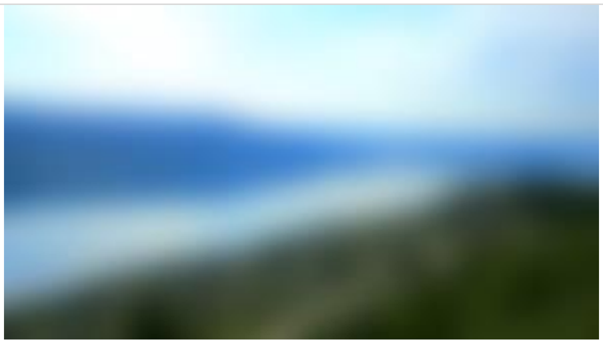
A7 (25 marks + up to 5 bonus marks)

***Focus**: CUDA Basic Thread Synchronization*

In this assignment, you will write code to compute a convolution of an input image with a filter (also known as a convolution kernel[1]). In this process, the convolution process takes as input two matrices, the image A and the filter F, and outputs a matrix, the convolved image H. The example below shows result of applying a blur filter to an image of the Okanagan valley:

Input Image A


Output Image H

Convolution Filter F

To illustrate the convolution operation, consider a matrix of floating-point numbers, where each cell stores a value reflecting the intensity of an image pixel ranging from 0 for black to 1 for white. For example:

$$A = \begin{bmatrix} \mathbf{1.0} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{1.0} & \mathbf{1.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{1.0} & \mathbf{1.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

And assume we are using the convolution filter below (which is a matrix usually with smaller size):

$$F = \begin{bmatrix} 0.0 & 0.1 & 0.0 \\ 0.1 & 0.6 & 0.1 \\ 0.0 & 0.1 & 0.0 \end{bmatrix}$$

The convolution of A with F would be

$$H = \begin{bmatrix} \mathbf{0.8} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{0.1} & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & \mathbf{0.1} & \color{red}{\mathbf{0.8}} & \mathbf{0.8} & \mathbf{0.1} & 0.0 \\ 0.0 & \mathbf{0.1} & \mathbf{0.8} & \mathbf{0.8} & \mathbf{0.1} & 0.0 \\ 0.0 & 0.0 & \mathbf{0.1} & \mathbf{0.1} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix}$$

[1] This term does not refer to the CUDA kernel function, but instead to a matrix used during the convolution operation.

The algorithm for computing the output image H includes the following steps:

1- For each pixel of *A* located at *(i , j)*, consider a submatrix S centered at *(i , j)* with the same size as the kernel F. For example, for *(i , j)* = (2 , 2), i.e. the red value in A, the resulting submatrix includes the values highlighted in yellow in A.

$$A = \begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & \mathbf{0.0} & \mathbf{0.0} & \mathbf{0.0} & 0.0 & 0.0 \\ 0.0 & \mathbf{0.0} & \mathbf{1.0} & \mathbf{1.0} & 0.0 & 0.0 \\ 0.0 & \mathbf{0.0} & \mathbf{1.0} & \mathbf{1.0} & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \rightarrow S = \begin{bmatrix} \mathbf{0.0} & \mathbf{0.0} & \mathbf{0.0} \\ \mathbf{0.0} & \mathbf{1.0} & \mathbf{1.0} \\ \mathbf{0.0} & \mathbf{1.0} & \mathbf{1.0} \end{bmatrix}$$

2- Compute the element-wise sum-of-products of S and F, i.e. ,

$$value\ for\ (\ i\ ,j\ ) = \ (0.0 + 0.0 + 0.0) + (0.0 + 0.6 + 0.1) + (0.0 + 0.1 + 0.0) = 0.8$$

3- Store *value* in *H(i , j)*. For the above example, *H(2 , 2) = 0.8*

For pixels at the boundary, you could replicate the values at the matrix edges to fill in the cells in S that represent the out-of-bound pixels. For example, the submatrix S for the pixel A(0,0) would be as follows (replicated values are written in green):

$$A = \begin{bmatrix} \mathbf{1.0} & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \rightarrow S = \begin{bmatrix} 1.0 & 1.0 & 0.0 \\ 1.0 & \mathbf{1.0} & 0.0 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

There are other options for the boundary pixels, e.g., to set the out-of-boundary pixels to 0.0.

In this assignment, the input image *A* is a 32-bit color image represented using RGBA format (e.g., the Okanagan valley image above). That is, every pixel in the image is 32-bits: 8-bits to represent the Red component, 8-bits for Green, 8-bits for Blue, and 8-bits for the Alpha channel. The filter *F* is a square floating-point matrix representing a *blur* operator (similar to the filter given in the examples above). The filter is stored as an image (e.g., filter_blur_21.bmp which is a 21-pixel wide filter)[2].

You are provided with the serial implementation of the convolution operation. The provided code uses the open-source EasyBMP[3] for reading and writing a BMP image. You are also provided with extra functions to convert an image to a format suitable for our program. In this format, each matrix element (i.e., pixel) is represented as `uchar4` CUDA type, which is a 4-byte type that has four components: x, y, z, and w, each represented in 1 byte (unsigned char). This type is perfectly suitable to represent a RGBA image pixel where each uchar4 component can be used to represent on image channel (R,G,B,A).

[2] Note that different filters produce different effects e.g., blur, sharpen, edge-detect, etc. See https://en.wikipedia.org/wiki/Kernel_%28image_processing%29 for an example of different filters.

[3] The full EasyBMP library and samples can be downloaded from easybmp.sourceforge.net. You don't need the full library as the required files are already attached to this assignment.

You need to do the following:

1. First, read the copyright note of the EasyBMP library.
2. Then, read the code in the file "convolution.cu" and understand its operation. Detailed comments are provided in order to explain how the code works. There is no need to read the code in the other files.
3. Once you finish the above, you need to:
   - **[+26]** Provide CUDA implementation of `convolution_32bits()` and the helper function `convolution_8bits()`
     - +8 marks for the host code
     - +4 marks for an additional function `convolution_8bits_parallel()`
     - +10 marks for an additional function `convolution_32bits_parallel()` as follows: +3 for splitting the image into four channels, +3 for proper threads synchronization, +4 for applying the 8bit convolution to each channel.
     - +4 marks for checking for errors from CUDA API calls and from Kernel Launch.

   - **[+4]** Time your code and report the speedup of using your CUDA implementation vs. the provided serial implementation.
     - +1 mark for reporting the serial time
     - +2 marks for reporting the parallel time and speedup.
     - +1 mark for including the output image from the parallel launch (even if it has errors)

     Add the timing notes and the output image from running your code in an MS Word.

   - Up to **+5 bonus marks** for creating two more filters, testing them using your code, and reporting the resulting images.

Submission Instructions

For this assignment, you need to submit to Blackboard Connect the following items in **one zip file** with your ID as its name (e.g., 1234567.zip):

- The **convolution.cu** file which has two additional functions: convolution_8bits_parallel() and convolution_32bits_parallel()
- An MS Word file that includes the output image and the time comparison (speedups): serial vs. parallel