CSCE 4930: GPU Computing
Homework 3 (double weight)

This assignment should be completed by teams of 2 to 3 students. Teams of 2 students are generally preferred. Teams of 3 might be required to implement extra functionalities. Only one submission per team is required.

You are required to implement and run all the tasks below on a computer having a CUDA-Enabled device. You can use any such computer you have access to, or you can use Google Colab. Please specify the specific GPU you are using and its CUDA compute capability in your assignment submission. Your submission should be a single compressed folder (e.g., zip) containing a source code subfolder for each task and a single PDF including output screenshots and question answers for all tasks.

For all tasks you are required to do the following:

1. Write a base (sequential) function that implements the required functionality.
2. Write a CUDA-kernel and a wrapper function that initializes the necessary device data and calls that kernel to implements the same functionality. **Make sure to use error checking for all CUDA API calls**.
3. Write a full program that calls both functions on the same input data and verifies that they both produce the same output.
4. Time both versions and compute the speedup (or slowdown) obtained from parallelization. When timing the parallel version, you should do it in two different ways and compute the speedup in each case:
    a. Timing the kernel only
    b. Timing the entire wrapper function (including the memory allocation and data transfer overheads).
5. **Your program should also print the performance of the sequential and parallel versions in GFLOPS**
6. **Report the GPU device occupancy**

Task 1:

Write a program to perform 2D convolution on **greyscale** images of **arbitrary size**. Your program should support the following eight operations: blur, emboss, outline, sharpen, and sobel (left, right, top, and bottom). **The image file name and the selected operation should be provided by the user**. The program should then open the file converting it to a 2D array, apply convolution to it based on the selected mask, and save the result as a new image file. The parallel version of your program should use **tiling (using 8x8 tiles, teams of 3 should also report on block sizes of 16x16 and 32x32)** and **constant memory** and should rely on **general caching** for halo-cells. **Boundary conditions should be handled by replicating the edge values**.
Note: to be able to get user inputs on Google Colab, you will need to use Scenario 1 proposed in the CUDA Google Colab guide that was posted on Blackboard earlier. This scenario (where you upload a `.cu` file and compile it using the `nvcc` compiler) will enable you to get inputs using `scanf`, `cin`, or as program arguments. This same scenario will also allow you to link the `libjpeg` library needed to read/write jpeg image files as explained below.

Here are the masks to be used for each of the requested operations:

Blur: $\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$, Emboss: $\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$, Outline: $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$,

Sharpen: $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$, Left Sobel: $\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$, Right Sobel: $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$,

Top Sobel: $\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$, Bottom Sobel: $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

For a visual (interactive) explanation of the different convolution operations and their effect on image, you can check this URL: http://setosa.io/ev/image-kernels/

You are free to use any C library to open and save image files. One such library, the Cool Image Library (or CImg library for short), can be found here:
https://cimg.eu/
This library supports several standard image file formats including jpeg files (given that the `ImageMagick` library is installed on Google Colab by default). The library does not need any installation as it only consists of a single header file `CImg.h`.
To use this library on Google Colab, you need to add the following directives in your `.cu` file:

```
#define cimg_display 0
#define cimg_use_jpeg
#include "CImg.h"
using namespace cimg_library;
```

You will also need to **upload** your `.cu` file as outlined in the Google Colab guide along with the `CImg.h` file (after downloading it from the link above) and the `jpeg` file you will use as an input.

Finally, to compile and run, you will need to create a Colab code cell with the following commands (assuming your `.cu` file is called `task1.cu`):

```
!nvcc "task1.cu" -o task1 -ljpeg
!./task1
```

If you want, you can disable compilation warnings by changing the first command to:

```
!nvcc "task1.cu" -o task1 -ljpeg --disable-warnings
```

The `CImg` class has many constructors, the most useful of which are:

- **`CImg<T> CImg(char *filename)`**: Constructs an image using an image file name. For example:

  `CImg<float> img1("cat.jpeg");`

- **`CImg<T> CImg(T *values, int width, int height, int depth=1, int channels=1)`**: constructs an image using a pointer to an array of pixel values (typically as floating-point values), and 4 integers representing the width, the height, the depth (1 for 2D images), and the number of channels (1 for grayscale images) of the constructed image. For example:

  `CImg<float> img2(data, 400, 300, 1, 1); //where data has type float*`

The first constructor is typically used when opening an image file for reading, while the second constructor is typically used when creating an image file from data in preparation to save it to a file.

The `CImg` class also the following useful functions:

- **`int width()`**: returns the width of the image object.
- **`int height()`**: returns the height of the image object.
- **`int depth()`**: returns the depth of the image object.
- **`int spectrum()`**: returns the number of channels of the image object.
- **`float *data()`**: returns a pointer to a flattened array containing all pixel values of the image object.
- **`CImg<T>& save(char *filename)`**: writes the image object to an file. You can ignore the return type and use it as a void function.

Task 2:

Write a program to compute the **total intensity of one or more rectangular subareas of a greyscale image of arbitrary size**. The program should use the **summed-area table** (aka **integral image**) to do this computation in constant time. The image file name and the coordinates of the pixels delimiting the rectangular subareas should be provided by the user. The program should then open the file converting it to a 2D array, compute the corresponding summed-area table, and finally use the obtained table to quickly compute the total intensity of the areas specified by the user.

The summed-area table is a 2D array **S** of the same size as the original image 2D array **I** such that the value of each **S** element at a position (x,y) is the sum of the intensities of all pixels in **I** with lower coordinates (i.e., all the pixels above and to the left of the pixel at (x,y)).

$$S(x,y) = \sum_{\substack{x' \leq x \\ y' \leq y}} I(x',y')$$

So, the summed-area table computation can be seen as a 2D extension of the prefix-sum (scan) computation. As such you should use a 2D extension of the hierarchal approach studied in the lecture.

So, if the original image I = $\begin{bmatrix} 10 & 50 & 100 & 0 \\ 50 & 255 & 35 & 0 \\ 0 & 16 & 10 & 80 \\ 10 & 20 & 200 & 150 \end{bmatrix}$, the summed-area table S will be $\begin{bmatrix} 10 & 60 & 160 & 160 \\ 60 & 365 & 500 & 500 \\ 60 & 381 & 526 & 606 \\ 70 & 411 & 756 & 986 \end{bmatrix}$

To find the total intensities in a rectangular area delimited by the points pixels A (x_1,y_1), B (x_2,y_1) , C(x_1,y_2) and D (x_2,y_2), we can do this in constant time using the formula:

$$\sum_{\substack{x_1 < x \leq x_2 \\ y_1 < y \leq y_2}} I(x,y) = S(D) + S(A) - S(B) - S(C)$$

For more information about summed-area tables, check this URL:
https://computersciencesource.wordpress.com/2010/09/03/computer-vision-the-integral-image/

Task 3:

Write a program to compute the **histogram** of the color intensities of a **greyscale** image. The image file name and the **number of histogram bins** should be provided by the user. Your kernel should use a **2D block of threads** to compute the histogram. You should use **interleaving** (strategy 2) and **privatization (without aggregation**, **teams of 3 should also report on privatization with aggregation)**. The output of your program is the histogram array which should be printed.

- **This assignment's weight is equivalent to 2 assignments** (40 marks instead of 20 marks). Task 1 alone is considered as one assignment, while tasks 2 and 3 are considered a second assignment.
- Submission Deadline: Sunday, December 11, 2022 11:59 PM.
- Submission method: Electronically through BB.
- Your submission should be **a single compressed folder (e.g., zip)** containing a single PDF report including output screenshots from all tasks and the device occupancy in each case. The compressed folder should also contain the source code for each task, **in addition to sample input and output images where applicable**.
- Check the readability of your submission. Hard to read submissions will be ignored.
- Plagiarism is NOT tolerated and will be reported.
- Late submissions will be penalized as described in the course syllabus and will NOT be accepted at all if more than 24 hours late.