ECSE 420: Parallel Computing – Fall 2019

Lab 1 Report : Simple CUDA Processing

Group 26

Anna Bieber - 260678856

Walid Chabchoub – 260677008

Date of Submission: Tuesday October 15th, 2019

Instructor: Prof. Zeljko Zilic

I. Introduction

The goal of this lab is to become more familiar with the CUDA infrastructure and design. Two types of pixel/signal processing were implemented with the CUDA kernel and helper files. The first one consisting of removing negative pixel values, the second one of finding the max value of 2x2 grid. The implementation, testing and speedup for both methods are detailed in the next sections.

II. Rectify

a. Implementation

Rectification produces an output image by repeating the following operation on each element of an input image as shown in equation 1 below:

$$output[i][j] = \begin{cases} input[i][j] & \text{if } input[i][j] \geq 0 \\ 0 & \text{if } input[i][j] < 0 \end{cases} \quad \textbf{(1)}$$

To understand the concept of rectification and how to implement it in the code, we started by implementing it using for loops. Once the implementation was done and giving us the correct output, we started implementing it threads and blocks instead of for loops to enable parallelization.

The implementation without threads consisted of two nested for loops iterating over the array (x and y axis), then for each pixel (which we get its index at x + y * width), we would get the R, G, B and Alpha values, check if they were smaller than 127 and correct them if needed (except for alpha). The concept of rectification was understood, and we could continue to work on an implementation with threads.

First, we needed to find a solution on how to access each channel from the pixel. The for loops could be removed if the threads could be used to access each pixel. From there, we decided on implementing a 1-D grid where the number of blocks on the x-axis was the pixel number of the image divided by the number of threads (this left a remainder which will be discussed later). Once this information was gathered, we could iterate over the pixels in the image with the blocks, where the desired number of threads would be used on 1 block at a time.

When dividing the number of pixels by the number of threads, there could be a remainder, thus we needed to handle these. We know that 2^10 - 1 = 1023 is the maximum value of a remainder for decimal numbers divisions so we can use a single block. Thus, we would use all the threads on this block.

To be able to access the index of each block and the specific location in a block, the grid dimensions and block ids were used. Once these were calculated, the rectification algorithm was the same as inside the for loops (we are not using for loops inside the CUDA kernel).

b. Testing

The testing for this part of the experiment was performed in two parts. Initially, we wanted to test that the rectify function was working with for loops, to ensure that we'd understood the algorithm correctly. We inputted the test image provided and checked with the expected output image. Next, we implemented using the grid and threads. Here, the provided test image was used as well, however we ran it to multiple issues before getting a correct output. The first issue we ran into was a white image at the output. After debugging, we realized that the indexing was not working correctly.

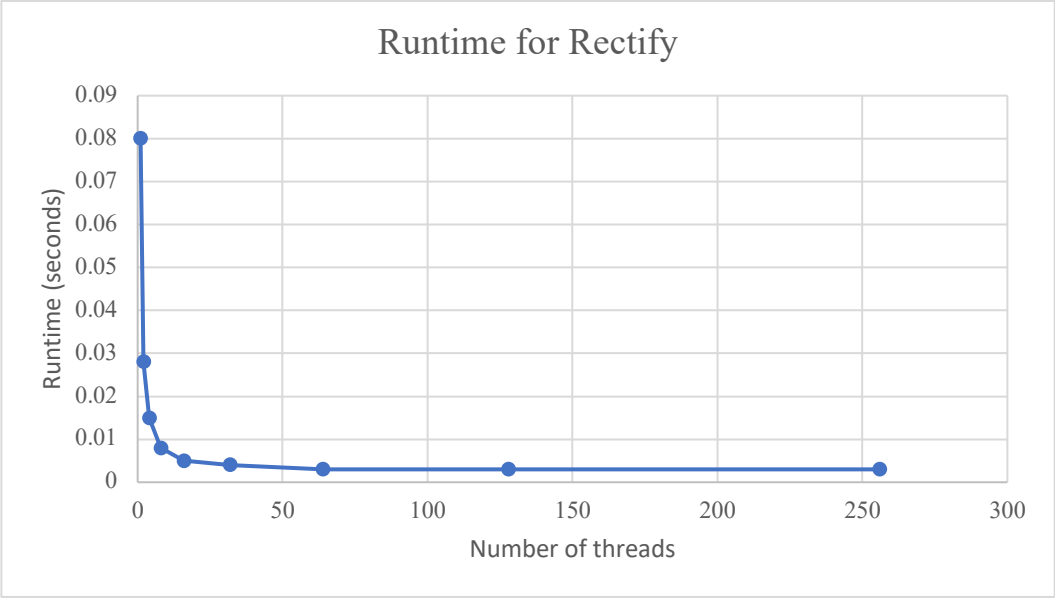
The second problem we had was only getting a slight rectification, which meant that we weren't rectifying all the pixels or not all the channels. For this we quickly found the issue by analyzing the code, the threads and grid was working properly.
When the implementation was finished and working correctly, we tested with different number of threads to be sure we obtained the same image. We noticed that using more than 1024 threads was not working, this is due to blocks only being able to handle 1024 threads, with more we would have to change the way we make the grid.

c. Speedup

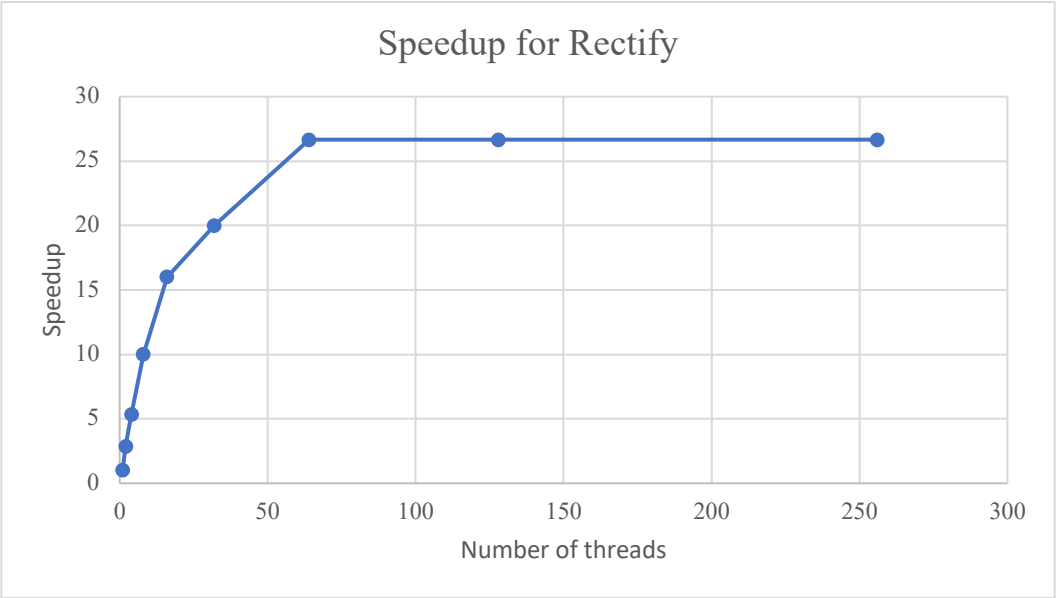
When running multiple threads versus running one, the program will execute faster. The different execution times and speedups for different number of threads can be seen in *Table 1* and the runtime plot can be seen in *Graph 1* and the speedup plot in *Graph 2* below:

Number of threads	Time	Speedup
1	0.08	1
2	0.028	2.857143
4	0.015	5.333333
8	0.008	10
16	0.005	16
32	0.004	20
64	0.003	26.66667
128	0.003	26.66667
256	0.003	26.66667

Table 1: Number of threads with its associated time and speedup for rectify



Graph 1: Runtime for Rectify



Graph 2: Speedup for Rectify

From table 1 above, we can observe that after 16 threads up to 64 the runtime is noticeable but minor. From 64 and up, the runtime is no longer noticeable by the system. This information can be visualized by the graph, where we can observe that the runtime becomes a straight line after 64. Similarly, in Graph 2, we can see that the speedup plot captures that, where in the beginning we can see an increase of the speedup until we reach a maximum plateau around a speedup of 26.

The produced results from this experiment are coherent with the ones we were expecting.

III. Pooling

a. Implementation

Pooling was implemented following the logic as described below in Fig. 1 using a 2x2 window size. This will resume in an image compression that is, the output image will have a width of original_width / 2 and a height of original_height / 2.

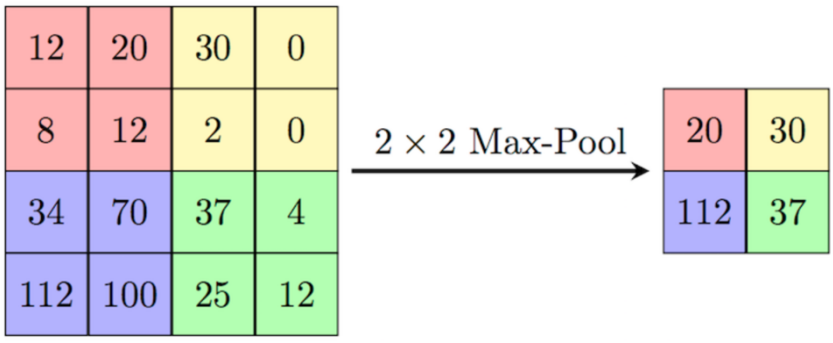


Figure 1: Max-pool visualization

In a similar fashion to the Rectify method above, we are navigating through the image array using x + width * y but using the block ids and thread ids as well as the grid & block dimensions. That is, we are adding the result of x + width * y to the pointer address of the char array representing the image. However, in order to implement pooling, we decided to approach it from a channel perspective, that is, focusing on treating each channel on its own.

Nevertheless, each thread is in charge of one 2x2 pool in the entire image, so each thread is computing the maximum of one pool (using threadIdx.x and blockIdx.x as well as blockIdx.y). Thus, for the entire image, we would iterate through the 4 channels (R, G, B and Alpha), next we would iterate through the y-axis of the 2x2 pool window, and determine the block offset to access the line below (in the image above, for instance if we start at the red pool at the top left, we want to access the bottom half (8 and 12), then we would compute the offset to add to the pointer to access that bottom line. Next, we would now iterate through the x-axis of the pool window and check the actual content of a channel, if it is higher than the current maximum, then update the maximum, otherwise, go to the next pool component. Finally, once a channel maximum has been found, append it to the output image array.

b. Testing

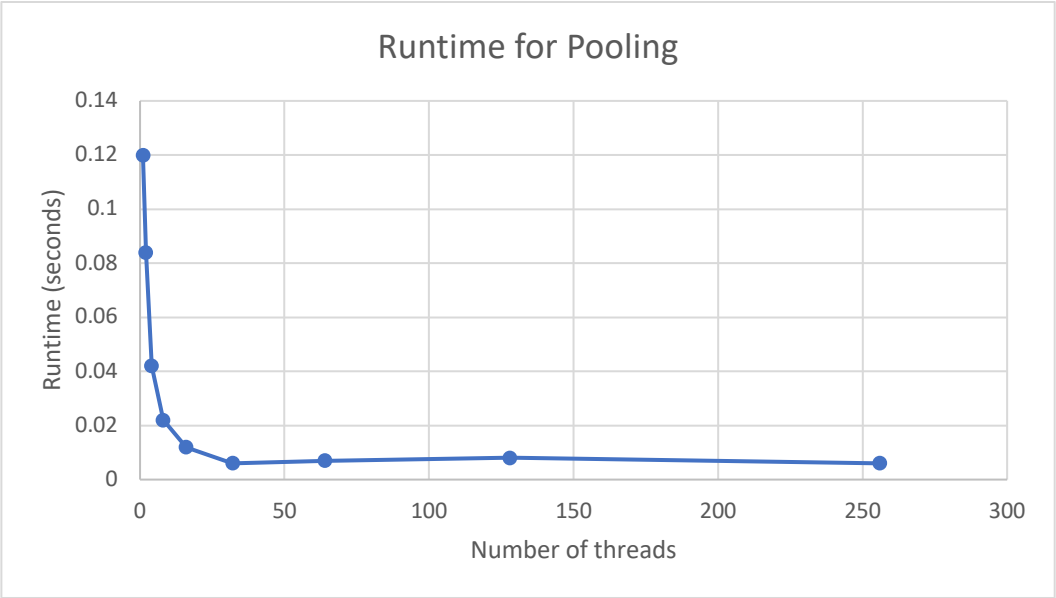
The testing procedure that we implemented for pooling was fairly simple and boiled down to two parts which we deemed sufficient for our application. First we would ensure that our program was outputting an image of the correct dimensions (that is, original dimensions divided by two). Next, we would use the test script provided to us to assess the MSE between the reference output image and our obtained output image.

c. Speedup

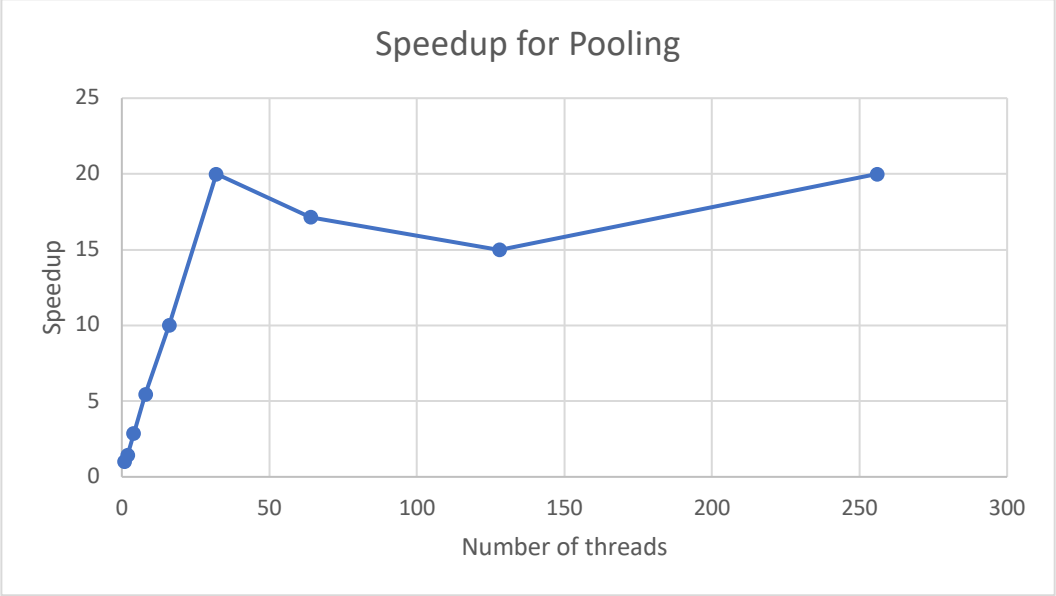
Similarly to rectify, running multiple threads results in faster execution. The different execution times and speedups for different number of threads can be seen below in *Table 2,* the runtime plot in Graph 3 and the speedup plot can be seen in *Graph 4*, below.

Number of threads	Time	Speedup
1	0.12	1
2	0.084	1.428571
4	0.042	2.857143
8	0.022	5.454545
16	0.012	10
32	0.006	20
64	0.007	17.14286
128	0.008	15
256	0.006	20

Table 2: Number of threads with its associated time for pooling



Graph 3: Runtime for Pooling



Graph 4: Speedup for Pooling

From table 2 above, we can observe that after 32 threads the runtime is very small compared to lesser threads count. This information can be visualized by Graph 3, where we can observe that the runtime becomes a straight line after 32 threads. Similarly, in Graph 4, we can see that the speedup plot captures that, where in the beginning we can see a steady increase of the speedup until we reach a maximum plateau after 32 threads, around a speedup between 15 and 20.

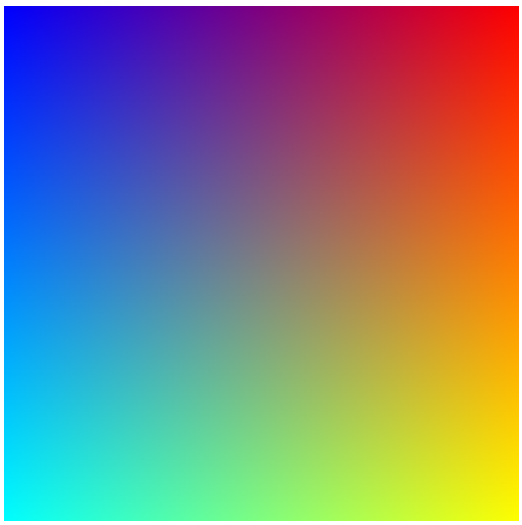
The produced results from this experiment are coherent with the ones we were expecting.

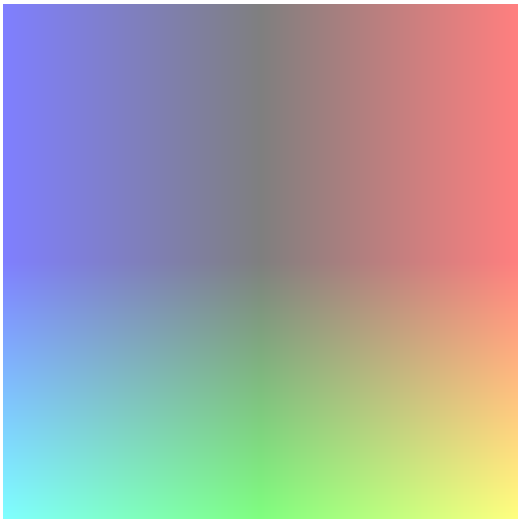
IV. Conclusion

The goal of this experiment was to become more familiar with the CUDA infrastructure and design. By implementing the rectification of an image and the pooling, the concepts of running threads instead of loops is understood. The observations made from the speedup show that increasing the number of threads can be useful up to a certain point where the computer will no longer perform better. This has allowed for a better understanding of the block and grid system used by CUDA.

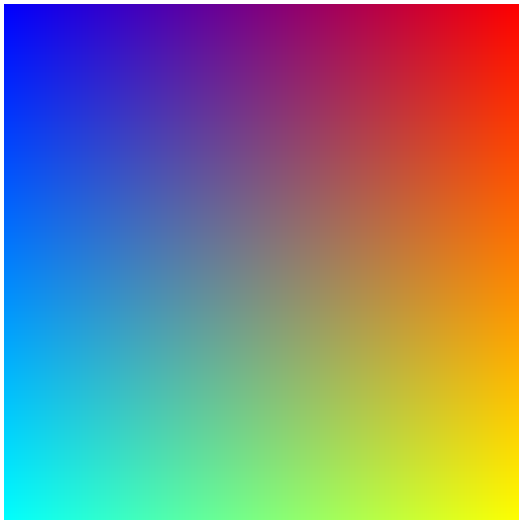
APPENDIX

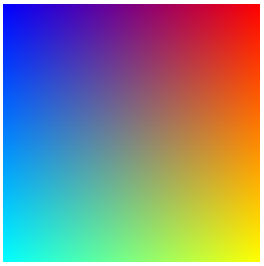
A. Original test image & output image for rectify





B. Original test image & output image for pooling





C. Rectify code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "lodepng.h"

#include <time.h>
#include <stdio.h>
#include <stdlib.h>


void rectifyWithCuda(unsigned char* image, unsigned numPixels, unsigned sizeChars,
unsigned int threads_per_block);

__global__ void rectify(unsigned char* dev_image, unsigned offset)
{
    // Numbers of blocks in a grid
    unsigned gridWidth = gridDim.x;
    // Fetching index of current block in 1-D (in comparison to the entire grid -
equivalent of x + y * width)
    int currentBlockIndex = blockIdx.x + blockIdx.y * gridWidth;

    // Numbers of threads per block
    unsigned blockWidth = blockDim.x;
    // Fetching index of current thread in 1-D inside the current block (in comparison
to the entire grid)
    int currentThreadIndex = threadIdx.x + currentBlockIndex * blockWidth;

    // In case total numbers of pixel is higher than total numbers of threads
available, then will use offset
    // This offset will "pickup" the rest of the work where it was last left.
    int currentPixelStartIndex = currentThreadIndex + offset;

    // 4 bytes per pixel (8 bits per channel R, G, B, Alpha)
    int currentPixelArray = 4 * currentPixelStartIndex;

    // Fetching the current pixel pointer to change its value from the device copy
(dev_image)
    // will overwrite the channels value and leave the alpha channel untouched
    unsigned char* currentPixelPointer = dev_image + currentPixelArray * sizeof(char);

    // Checking if Red channel value of this pixel (located at the start of the
pointer) is less than 127
    // if so, set it to 127, otherwise, leave it untouched
    if ((int)currentPixelPointer[0] < 127) {
        currentPixelPointer[0] = (unsigned char)127;
    }
```

```
    // Similarly, to Green channel
    if ((int)currentPixelPointer[1] < 127) {
        currentPixelPointer[1] = (unsigned char)127;
    }
    // Similarly, to Blue channel
    if ((int)currentPixelPointer[2] < 127) {
        currentPixelPointer[2] = (unsigned char)127;
    }
}

int main(int argc, char* argv[])
{
    // Extracting CLI arguments
    char* input_filename = argv[1];
    char* output_filename = argv[2];
    int threads_per_block = atoi(argv[3]);

    if (threads_per_block > 1024) {
        printf("Error, number of threads per block cannot exceed 1024. Please try
another value.");
        return 1;
    }

    unsigned error;
    unsigned char* image;
    unsigned width, height;

    // Decoding and loading the PNG image to the image pointer and setting the width
and height values as well
    error = lodepng_decode32_file(&image, &width, &height, input_filename);
    if (error) printf("error %u: %s\n", error, lodepng_error_text(error));

    unsigned numPixels = width * height;
    // Given that we have 4 channels, we have 4 chars per pixel (8 bytes per channel
so 32 bits in total per pixel)
    unsigned sizeChars = numPixels * 4;

    // Method handling device memory setup and launching the kernel
    rectifyWithCuda(image, numPixels, sizeChars, threads_per_block);

    // Encoding and saving the new rectified image in a PNG
    lodepng_encode32_file(output_filename, image, width, height);

    // Freeing memory
    free(image);
    return 0;
}
```

```
void rectifyWithCuda(unsigned char* image, unsigned numPixels, unsigned sizeChars,
unsigned int threads_per_block)
{
    // Counter for timing
    double time_elapsed = 0.0;

    // Pointer for device copy of the image
    unsigned char* dev_image;
    // Allocating device memory to it
    cudaMalloc((void**)(&dev_image), sizeChars * sizeof(char));
    // Perfoming the copying
    cudaMemcpy(dev_image, image, sizeChars, cudaMemcpyHostToDevice);

    // Variables used to create the dim3 struct
    int numXBlocks, numYBlocks;

    // Computing the numbers of blocks (x-axis & y-axis)
    numXBlocks = numPixels / threads_per_block;
    numYBlocks = 1;

    // Creating dim3 structs to contain the dimensionality of our GPU that we will be
requiring
    // (Z axis set to 1 for threads and blocks and Y axis set to 1 as well for threads
because of hardware possible limitationss)
    dim3 totalThreadsPerBlock(threads_per_block, 1, 1);
    dim3 totalBlocks(numXBlocks, numYBlocks, 1);

    printf("Using %d blocks (%d in x, %d in y), each with %d threads .\n",
numXBlocks*numYBlocks, numXBlocks, numYBlocks, threads_per_block);

    // Starting timer
    clock_t star_time = clock();

    // Calling the kernel with an offset value of 0 given that we are at the start (no
remainder involved yet)
    rectify <<<totalBlocks, totalThreadsPerBlock >>> (dev_image, 0);

    // Given that the division of numPixels / threads_per_block can have a remainder,
then we will be using an "offset"
    // 2^10 - 1 = 1023 is the maximum value of a remainder for decimal numbers
divisions so we can use a single block
    int remainder = numPixels - (threads_per_block * numXBlocks * numYBlocks);
    printf("Remaining pixels %d.\n", remainder);

    // Offset can never be negative
    unsigned offset = numPixels - remainder;
    // Launching another kernel to deal with the remaining pixels and giving the
offset to locate the starting pixel
```

```
    // of these remaining ones
    rectify <<<1, remainder>>> (dev_image, offset);

    cudaDeviceSynchronize();

    // After all rectification is done, copy the processed image from the device
memory to host memory (overwriting)
    cudaMemcpy(image, dev_image, sizeChars, cudaMemcpyDeviceToHost);

    // Clear the device memory allocation
    cudaFree(dev_image);

    // Ending timer
    clock_t end_time = clock();
    time_elapsed += (double)(end_time - star_time) / CLOCKS_PER_SEC;
    printf("Time elapsed for CUDA operation is %f seconds", time_elapsed);

}
```

D. Pooling code

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include "lodepng.h"

#include <time.h>
#include <stdio.h>
#include <stdlib.h>


void poolWithCuda(unsigned char* image_in, unsigned char* image_out, unsigned
numPixels_in, unsigned numPixels_out, unsigned sizeChars_in, unsigned sizeChars_out,
unsigned width, unsigned int threads_per_block);


__global__ void pool(unsigned char* dev_image_in, unsigned char* dev_image_out,
unsigned width_in, unsigned offset)
{
    // Numbers of blocks in a grid
    unsigned gridWidth = gridDim.x;
    // Fetching index of current block in 1-D (in comparison to the entire grid -
equivalent of x + y * width)
    int currentBlockIndex = blockIdx.x + blockIdx.y * gridWidth;

    // Numbers of threads per block
    unsigned blockWidth = blockDim.x;
```

```
    // Fetching index of current thread in 1-D inside the current block (in comparison
to the entire grid)
    int currentThreadIndex = threadIdx.x + currentBlockIndex * blockWidth;

    // In case total numbers of pixel is higher than total numbers of threads
available, then will use offset
    // This offset will "pickup" the rest of the work where it was last left.
    int currentPixelStartIndex = currentThreadIndex + offset;

    // Pointer with which we will navigate through out the 2x2 pool window
    unsigned char* currentPixelPointer;

    // Variable that will hold the maximum value for a channel (R, G or B)
    unsigned maxChannelValue;

    // Variable that will hold the current value of a channel
    unsigned channelValue;

    // We will be traversing channel by channel throughout the entire image
    // to fetch the maximum of each channel even if it is not in the same image inside
a pool
    for (int channelIndex = 0; channelIndex < 4; channelIndex++) {
        // For RGB channel
        if (channelIndex < 3) {
            // At the start of each channel sweep, initializing the max value to 0
            maxChannelValue = 0;
            // Starting of iterating through the Y-axis of the pool to use the (x + y
* width formula)
            for (int pool_y_index = 0; pool_y_index < 2; pool_y_index++) {
                // Fetching the X-axis index in the 2x2 pool window
                unsigned blockOffset = 4 * 2 * (width_in * (currentPixelStartIndex /
(width_in / 2)) + (currentPixelStartIndex % (width_in / 2)));
                // Pointing to the current pixel in the 2x2 pool window by adding the
X-axis index and the y * width equivalent inside this pool
                // to the input image starting address
                currentPixelPointer = dev_image_in + blockOffset + 4 * width_in *
pool_y_index;
                // Now iterating through the X-axis of the 2x2 pool window
                for (int pool_x_index = 0; pool_x_index < 2; pool_x_index++) {
                    // We know that a pixel contains 4 chars (one for each channel)
                    int currentPoolPixelIndex = 4 * pool_x_index;
                    // We can point to next pixel in the X-axis by simply skipping the
current pixel in the x-axis of the pool window
                    currentPixelPointer += currentPoolPixelIndex;
                    // Fetching the current channel value of the current pixel
                    channelValue = (int)currentPixelPointer[channelIndex];
                    // If current vlaue is higher than the previous maximu, replace it
                    if (channelValue > maxChannelValue) {
```

```
                        maxChannelValue = channelValue;
                    }
                }
            }
            // Once the current channel maximum of a pool has been found
            channelValue = (unsigned char) maxChannelValue;
        }
        // For Alpha channel, set the maximum (255)
        else {
            channelValue = (unsigned char) 255;
        }
        // After assessing the maximum value of a channel of a 2x2 pool, set it in the
output image
        dev_image_out[4*currentPixelStartIndex + channelIndex] = channelValue;
    }
}

int main(int argc, char* argv[])
{
    // Extracting CLI arguments
    char* input_filename = argv[1];
    char* output_filename = argv[2];
    int threads_per_block = atoi(argv[3]);

    if (threads_per_block > 1024) {
        printf("Error, number of threads per block cannot exceed 1024. Please try
another value.");
        return 1;
    }

    unsigned error;
    unsigned char* inputImage,* outputImage;
    unsigned width, height, width_out, height_out;


    // Decoding and loading the PNG image to the image pointer and setting the width
and height values as well
    error = lodepng_decode32_file(&inputImage, &width, &height, input_filename);
    if (error) printf("error %u: %s\n", error, lodepng_error_text(error));


    width_out = width / 2;
    height_out = height / 2;
    unsigned numPixels_in = width * height;
    unsigned numPixels_out = width_out * height_out;

    // Total size of output image is going to be the output number of pixels times the
size of a char times 4
```

```
    // given that each pixels is 32-bits so 4 chars (4 channels of 8 bits each)
    outputImage = (unsigned char*)malloc(4 * numPixels_out * sizeof(char));


    // Given that we have 4 channels, we have 4 chars per pixel (8 bytes per channel
so 32 bits in total per pixel)
    unsigned sizeChars_in = numPixels_in * 4;
    unsigned sizeChars_out = numPixels_out * 4;

    // Method handling device memory setup and launching the kernel
    poolWithCuda(inputImage, outputImage, numPixels_in, numPixels_out, sizeChars_in,
sizeChars_out, width, threads_per_block);

    // Encoding and saving the new max pooled image in a PNG
    lodepng_encode32_file(output_filename, outputImage, width_out, height_out);

    // Freeing memory
    free(inputImage);
    free(outputImage);
    return 0;
}

void poolWithCuda(unsigned char* image_in, unsigned char* image_out, unsigned
numPixels_in, unsigned numPixels_out, unsigned sizeChars_in, unsigned sizeChars_out,
unsigned width, unsigned int threads_per_block)
{
    // Counter for timing
    double time_elapsed = 0.0;

    // Pointer for device copy of the image
    unsigned char* dev_image_in, * dev_image_out;
    // Allocating device memory to it
    cudaMalloc((void**)(&dev_image_in), sizeChars_in * sizeof(char));
    cudaMalloc((void**)(&dev_image_out), sizeChars_out * sizeof(char));

    // Perfoming the copying
    cudaMemcpy(dev_image_in, image_in, sizeChars_in, cudaMemcpyHostToDevice);

    // Variables used to create the dim3 struct
    int numXBlocks, numYBlocks;

    // Computing the numbers of blocks (x-axis & y-axis)
    numXBlocks = numPixels_out / threads_per_block;
    numYBlocks = 1;

    // Creating dim3 structs to contain the dimensionality of our GPU that we will be
requiring
```

```
    // (Z axis set to 1 for threads and blocks and Y axis set to 1 as well for threads
because of hardware possible limitationss)
    dim3 totalThreadsPerBlock(threads_per_block, 1, 1);
    dim3 totalBlocks(numXBlocks, numYBlocks, 1);

    printf("Using %d blocks (%d in x, %d in y), each with %d threads .\n", numXBlocks
* numYBlocks, numXBlocks, numYBlocks, threads_per_block);

    // Starting timer
    clock_t star_time = clock();

    // Calling the kernel with an offset value of 0 given that we are at the start (no
remainder involved yet)
    pool <<<totalBlocks, totalThreadsPerBlock >> > (dev_image_in, dev_image_out,
width, 0);

    // Given that the division of numPixels_out / threads_per_block can have a
remainder, then we will be using an "offset"
    // 2^10 - 1 = 1023 is the maximum value of a remainder for decimal numbers
divisions so we can use a single block
    int remainder = numPixels_in - (threads_per_block * numXBlocks * numYBlocks);
    printf("Remaining pixels %d.\n", remainder);

    // Offset can never be negative
    unsigned offset = numPixels_out - remainder;
    // Launching another kernel to deal with the remaining pixels and giving the
offset to locate the starting pixel
    // of these remaining ones
    pool <<<1, remainder >>> (dev_image_in, dev_image_out, width, offset);

    cudaDeviceSynchronize();

    // After all pooling is done, copy the processed image from the device memory to
host memory (overwriting)
    cudaMemcpy(image_out, dev_image_out, sizeChars_out, cudaMemcpyDeviceToHost);

    // Clear the device memory allocation
    cudaFree(dev_image_in);
    cudaFree(dev_image_out);

    // Ending timer
    clock_t end_time = clock();
    time_elapsed += (double)(end_time - star_time) / CLOCKS_PER_SEC;
    printf("Time elapsed for CUDA operation is %f seconds", time_elapsed);

}
```