ECSE 420 – Parallel Computing – Fall 2020

Lab 0 – Simple CUDA Processing

Aleksas Murauskas 260718389

Anne-Julie Côté 260803768

Group 16

1. Image Rectification

The image rectification portion of our lab is accomplished in the file image_rect.cu.

The program is called with the command [./rectify <input png> <output png> <# threads>]. If there are not four arguments, the user will be prompted for their error and then exit. The input is processed first by assigning the number of threads the program will rectify the image with. If the value entered is a negative it is rectified to a value of 1, and if it is larger than the maximum number of threads, 1024, then it is set to 1024. To prepare to decode, memory for height, width, and image data are allocated. The method lodepng_decode32_file() given in the lodepng.cu file is used to find the address of the image data, width, and height. If there is an error while opening the time, the program will display it to the user using the lodepng_error_text() method.

Then, the total image size is calculated by multiplying the image width, image height, the size of an unsigned char and 4 to represent the 4 bytes per RGBA pixel. We then allocate memory of the image size for both the input and the output images that will be processed on the device (using cudaMalloc). The image data is copied into allocated CUDA memory to prepare for the rectification.

The total number of bytes that will be rectified is then calculated, and a timer begins. The global kernel method rectify() is then called, which is the parallelization part of the program. In order to accomplish a parallel computation, the method starts the number of threads specified by the input of the user. Since there are more bytes in the image than threads, each call of the rectify() method will process bytes starting at its thread ID and incrementing by the number of threads, until it reaches the number of bytes. This is to reduce overlap and to ensure that all bytes are covered.

The method rectifies the values by setting the value of the output memory location at the index to 127 if the value of the input at the index is less than 127, otherwise the value is simply copied from input to output. The method cudaDeviceSynchronize() is used to end the parallel threads. Once the rectification is over, the timer ends.

After the parallel computing has concluded, the output is copied out of the device and then lodepng_encode32_file() is used to turn the raw image data back into a png file. The program concludes by freeing the memory allocated for storing the output image and CUDA input/output, and by showing how much time elapsed during the parallelization.

To ensure that the program worked appropriately, we ran it with the Test_1.png image that was provided, and then compared our new image to the Test_1_rectified.png image. We received the following output, which ensured that the rectification was done correctly:

```
(base) [annejuliecote] 15:56:55 - ECSE420$ ./a.out
Images are equal (MSE = 0.000000, MAX_MSE = 0.000010)
```

Figure 1. Output of running the test_equality.c program

Figure 2. Test_1.png after our rectification program

We then ran the program on the two other provided images, Test_2.png and Test_3.png. They were also rectified correctly.

Figure 3. Test_2.png after our rectification program

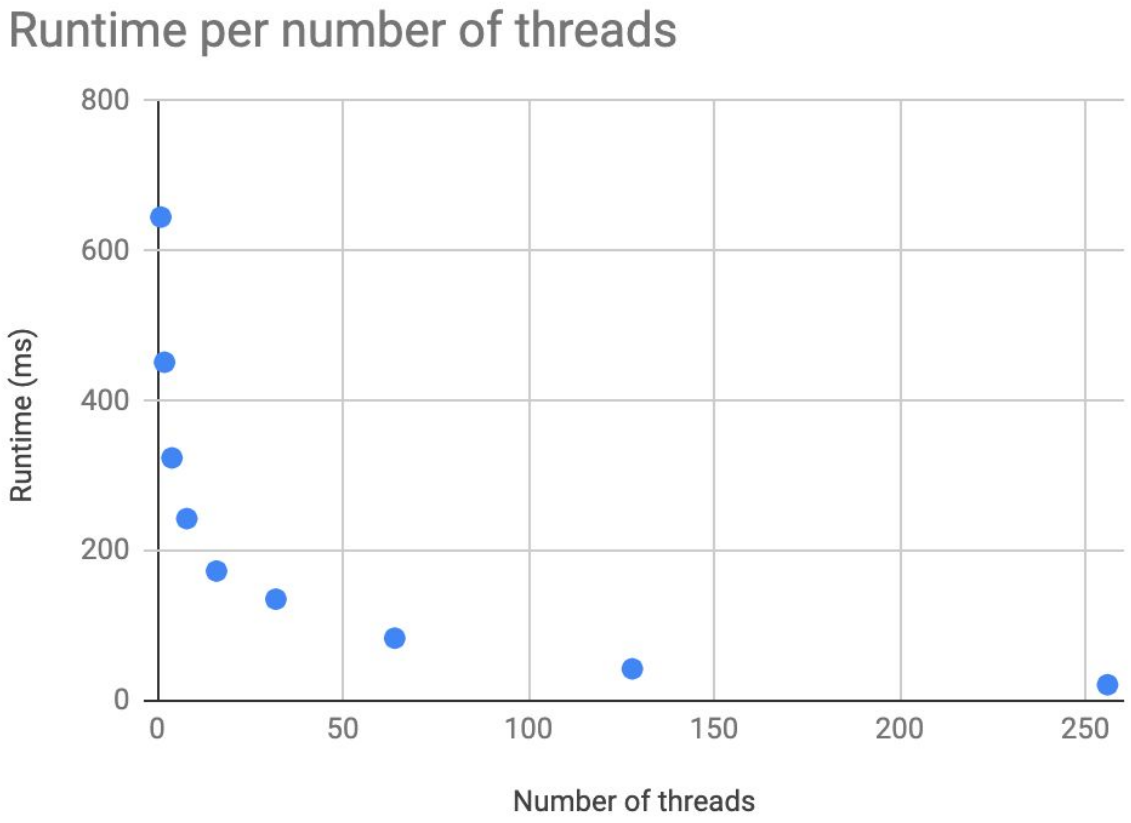
Figure 3. Test_3.png after our rectification program

Measuring the runtime the rectification of the Test_3.png images with different numbers of threads gave the following results:

Table 1: Runtime of rectification with different numbers of threads

Number of threads	Runtime (ms)
1	645.040649
2	451.443970
4	323.866211
8	242.870651
16	172.922470
32	135.456284
64	83.455772
128	42.547550
256	21.313984

Graph 1: Runtime of rectify based on number of threads



We can see that the runtime decreases significantly when the number of threads increase from 1 to approximately 32, and then keep decreasing but at a lower rate as they increase from 32 to 264. The function traced seems to be exponentially decreasing.

2. Pooling

Design:

The Pooling process is accomplished in the file pool.cu.

The program parses user input in the same way as our image rectifier. When allocating memory for the input and output in CUDA, a quarter of memory is allocated for the output as the pooling will reduce the amount of pixels that amount.

In this program we ran parallel threads by finding how many times larger the pixels of output the output will be compared to the number of threads we are allowed to use. We then task each thread to complete that amount of output pixels by calling the pool_process() method. There is also a check for leftover pixels, in that case there is a remainder of pixels still needing processed. Each of those pixels is handed off to its own thread in that case.

Pool_process() is the method that pools the input images pixels into the new image's memory. We begin by getting an index location from the thread ID, this is the point this output the thread will begin to pool. In the case of leftovers, the index is incremented This is accomplished through a series of nested loops. The first loop iterates through each pixel that thread will pool. The second loop iterates through the RGBA values. For A, the opacity is set to the max value 255. For RGB, another 2 nested loops occur to find the locations of the desired 4 color values from each pixel. At this level, the comparison begins to find the maximum value for that color, each location of memory is iterated through and compared to a maximum value(initially 0), and it is replaced as necessary. After the four pixels have been read, the maximum value or opacity is set stored in the output.

After the initial thread pooling and the leftovers are completed, the program terminates in the same way as the rectifier program.

Issues in the program:

We were unable to get this portion to properly run. From what we believe, an error in our mathematics to update and/or find the correct values in the memory was causing the issue. The erroneous result does not change regardless of the amount of threads used, so we believe it to be independent of our parallel computational strategy.

Input Test 2: In this case, Remy's ears are somewhat visible and he appears twice, which leads us to believe either the the output is being improperly placed in the output buffer, or the mathematics of pooling the values from the input memory is incorrect. It's possible this is due to an error in the mathematics in creating or updating the pointer actively traversing the input.


Input Test_3.jpg: In this case, the returned image looks nothing at all like Po and his band of heroes. The resulting image is scrambled. It's likely the same error is occurring, but the file change in file dimensions may be causing the image to become much more distorted.