ECSE420 Lab0 Report

Liang Zhao 260781081
Hongshuo Zhou 260792817

October 1 2020

Group 6

1 Introduction

Generally in this Lab, we are required to write two signal processing algorithms in order to test the effect of parallel computing. We used cuda C to perform our experiments on Google Colab. The two algorithms were tested using 1, 2, 4, 8, 16 and 32 threads in parallel for purposes of comparison between the time taken with one thread versus multiple threads. The test files are provided from lab files (test images)along with the test equality program and lodepng program. Further details of lab implementation and results will be discussed in following section.

2 Algorithm Design And Testing

2.1 Implementation

For this lab, in algorithm part we split the code into serveral individual algorithm function: *global void rectify*, *global void pool*,*void imageRectify*,*void imagePooling* and main function.

2.1.1 Image Rectification

Rectification process is to scan through each pixel of the image and changing the value of pixels, hence changing their RGB properly. If a pixel's value was found to be lower than 127, it was then set to 127. The algorithm that we implemented parallelizes the computational. After the png file is loaded, the function would check for error first. Then the length of png file is calculated, and our function allocate corresponding space in memory for image. The algorithm then initialized png data array and run rectify function on GPU with input threads number. Finally our alogotithm used cudaDeviceSynchronize() to wait for GPU threads to complete and write the rectified image data to output new image file.

$$Speedup = \frac{Told}{Tnew}$$

Told stands for the runtime with only one thread, *Tnew* stands for the runtime under different number of threads. In order to compare runtime performances under different number of threads, we used Speedup to measure it. We ran the algorithm with a fixed number of threads 5 times and

take the average using n threads, where n= 1, 2, 4, 8, 16, 32, 64, 128, 256.

Thread number(n)	Runtime records(ms)	Average runtime(ms)
1	68.479,62.136,68.706,62.101,68.483	65.98
2	31.259,34.374,34.364,34.378,34.376	33.75
4	17.260,17.305,17.249,17.248,17.276	17.27
8	8.687,8.665,8.650,8.674,8.683	8.672
16	4.333,4.392,4.408,4.366,4.393	4.378
32	2.230,2.229,2.195,2.232,2.239	2.225
64	1.189,1.158,1.189,1.161,1.159	1.171
128	0.630,0632,0.623,0.621,0.620	0.625
256	0.363,0.404,0.423,0.394,0.408	0.398

Table 1: runtime table

Speedup Calculation:
2 threads = 65.98 / 33.75 = 1.955
4 threads = 65.98 / 17.27 = 3.820
8 threads = 65.98 / 8.672 = 7.608
16 threads = 65.98 / 4.378 = 15.07
32 threads = 65.98 / 2.225 = 29.65
64 threads = 65.98 / 1.171 = 56.35
128 threads = 65.98 / 0.625 = 105.6
256 threads = 65.98 / 0.398 = 165.8

Below are the graphs representing speedup vs number of threads used. As the figure shows, the peak speedup of 165.8 happens when we run the rectification algorithm using 256 threads. The figure also shows a gradually increasing trend of speedup as number of threads increased. We ran our algorithm on Google Colab platform, clearly there is no intermediate peak between 1 and 256 threads, thus we can conclude that Google Colab could support up to 256 threads for parallel computing. As the number of threads increased, the execution speed would increase too. Also, it is worth mentioning that the speedup for each number of threads is close to its corresponding number of thread, although when reach 128 and 256 threads the speedup are not really close. This phenomenon support the success of our parallel computing, and the optimize thread number shall be 64 .

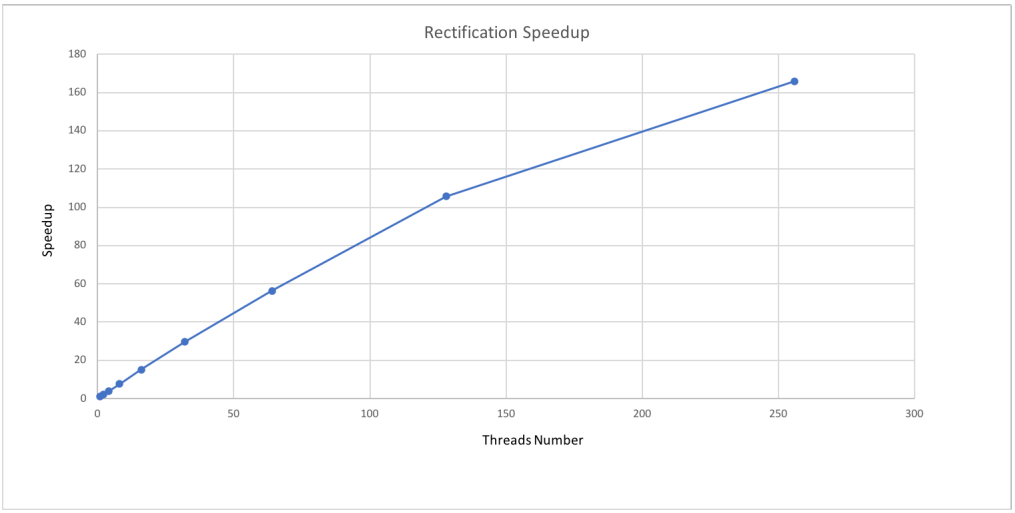

Figure 1: Rectification Speedup

. Below are the output of our rectification algorithm on three test images. All of the output images passed the testequality test.

Figure 2: Test1

Figure 3: Test2

Figure 4: Test3

2.1.2 Image Pooling

Pooling consisted of compressing an image of even size (width and height divisible by 2) by slicing the image in sections of 2x2 disjoint squares and forming a new 1x1 square by taking the maximum RGBA value in the section. Therefore, the image compressed to a new image with width and height which are two times smaller than the original image. Our algorithm successfully pooled the test images, where it was possible to divide the image into several blocks of pixels and allow each thread to scan through each

array of pixels belonging to the image and compress this selection of pixels individually. The algorithm basically compare all the four pixels value in each 2x2 disjoint squares and output the max one, then combine into a new 2x2 disjoint squares.
Similarly, in order to compare runtime performances under different number of threads, we used Speedup to measure it. We ran the algorithm with a fixed number of threads 5 times and take the average using n threads, where n= 1, 2, 4, 8, 16, 32, 64, 128, 256.

Thread number(n)	Runtime records(ms)	Average runtime(ms)
1	2589,2604,2599, 2605,2602	2600
2	2017,2031,2029,2028,2010	2023
4	1089,1088,1073,1092,1092	1087
8	636.4,630.8,634.6,632.0,635.0	633.8
16	316.9,324.0,313.1,320.5,323.1	319.5
32	163.3,167.3,167.3,156.7,162.7	163.5
64	84.36,78.63,83.03,84.05,78.67	81.75
128	43.01,42.95,43.06,43.01,43.02	43.01
256	23.28,23.27,23.28,23.29,23.29	23.28

Table 2: runtime table

Speedup Calculation:
2 threads = 2600 / 2023 = 1.285
4 threads = 2600 / 1087 = 2.392
8 threads = 2600 / 633.8 = 4.102
16 threads = 2600 / 319.5 = 8.138
32 threads = 2600 / 163.5 = 15.9
64 threads = 2600 / 81.75 = 31.80
128 threads = 2600/ 43.01 = 60.45
256 threads = 2600 / 23.28 = 111.7

Below are the graphs representing speedup vs number of threads used.
As the figure shows, the peak speedup of 111.7 happens when we run the pooling algorithm using 256 threads. The figure also shows a gradually increasing trend of speedup as number of threads increased from 4 to 256. We ran our algorithm on Google Colab platform, clearly there is no intermediate peak between 1 and 256 threads, thus we can conclude that Google Colab

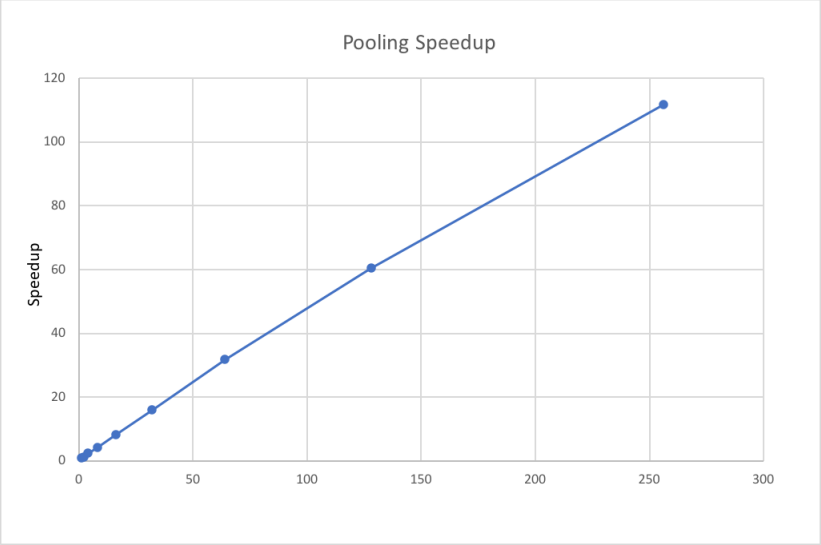

Figure 5: Pool Speedup

could support up to 256 threads for parallel computing. As the number of threads increased, the execution speed would increase too. Also, it is worth mentioning that the speedup for each number of threads is close to half of its corresponding thread number, and there is no significant speed increase when we using 2 threads to run the algorithm. This is due to the pooling algorithm used 4 threads for one group to process the pixel value in the 2x2 disjoint squares. This phenomenon support the success of our parallel computing.

Below are the output of our pool algorithm on three test images. All of the output images passed the testequality test.

Figure 6: Test1

Figure 7: Test2

Figure 8: Test3