



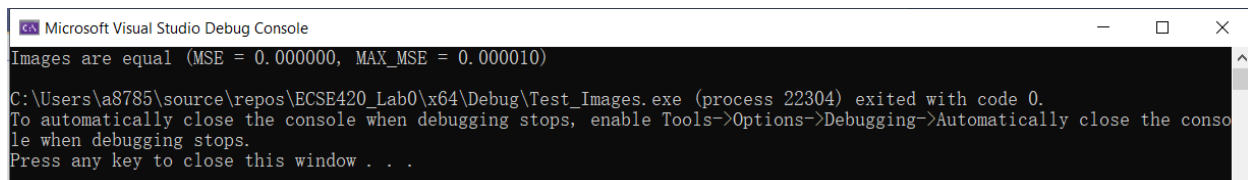
ECSE420 Parallel Computing  
Lab 0 Simple CUDA Processing  
Group 43  
Hanwen Wang 260778557  
Mai Zeng 260782174

## Image Rectification

This method compares all RGB values to 127 in each pixel in the input image. If the value is greater or equal to 127, then it remains unchanged. If the value is smaller than 127, it is set to 127. All new values are stored in the same position in the output image.

## Parallelization scheme

Since in the instruction that the maximum threads used is 256 and one block has up to 1024 threads, the method “one block and parallel threads” is enough to finish this task. As a result, we set “index = threadIdx.x” and call “rectify <<<1, size>>>” to make sure it is parallel. Each thread is supposed to handle one pixel, and a group of threads will run together depending on the thread size. We need to calculate how many groups of threads are needed to finish this task at first. Then for each group of threads we call the function “rectify”, which tracks the current group of pixels needed to be handled by that group of threads and does calculation and comparison afterwards. After we build and run the code, we tested the output of Test\_1.png with Test\_1\_rectified.png provided and found that they are almost the same.

A screenshot of the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio Debug Console". The console output shows: "Images are equal (MSE = 0.000000, MAX\_MSE = 0.000010)", followed by "C:\Users\as785\source\repos\ECSE420\_Lab0\Debug\Test\_Images.exe (process 22304) exited with code 0.", and then instructions: "To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops." and "Press any key to close this window . . .".

```
Microsoft Visual Studio Debug Console
Images are equal (MSE = 0.000000, MAX_MSE = 0.000010)
C:\Users\as785\source\repos\ECSE420_Lab0\Debug\Test_Images.exe (process 22304) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 1. Test image equality

## Speedup plots

The speedup is the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on  $p$  processors.

Its equation is  $S = \frac{T_s}{T_p}$ . The timer starts just before we are going to rectify and stops after `cudaDeviceSynchronize()` is called. The runtime images and graphs are listed below.

```

Microsoft Visual Studio Debug Console
Rectify image Test_1.png using 1 thread(s): 38.214718seconds
Rectify image Test_1.png using 2 thread(s): 19.297478seconds
Rectify image Test_1.png using 4 thread(s): 9.681293seconds
Rectify image Test_1.png using 8 thread(s): 4.944248seconds
Rectify image Test_1.png using 16 thread(s): 2.645226seconds
Rectify image Test_1.png using 32 thread(s): 1.325371seconds
Rectify image Test_1.png using 64 thread(s): 0.751980seconds
Rectify image Test_1.png using 128 thread(s): 0.372131seconds
Rectify image Test_1.png using 256 thread(s): 0.208194seconds
Rectify image Test_2.png using 1 thread(s): 9.314495seconds
Rectify image Test_2.png using 2 thread(s): 4.657412seconds
Rectify image Test_2.png using 4 thread(s): 2.237837seconds
Rectify image Test_2.png using 8 thread(s): 1.186132seconds
Rectify image Test_2.png using 16 thread(s): 0.594062seconds
Rectify image Test_2.png using 32 thread(s): 0.290896seconds
Rectify image Test_2.png using 64 thread(s): 0.148546seconds
Rectify image Test_2.png using 128 thread(s): 0.081445seconds
Rectify image Test_2.png using 256 thread(s): 0.038088seconds
Rectify image Test_3.png using 1 thread(s): 10.099084seconds
Rectify image Test_3.png using 2 thread(s): 5.248481seconds
Rectify image Test_3.png using 4 thread(s): 2.605483seconds
Rectify image Test_3.png using 8 thread(s): 1.325451seconds
Rectify image Test_3.png using 16 thread(s): 0.700619seconds
Rectify image Test_3.png using 32 thread(s): 0.387056seconds
Rectify image Test_3.png using 64 thread(s): 0.192659seconds
Rectify image Test_3.png using 128 thread(s): 0.110742seconds
Rectify image Test_3.png using 256 thread(s): 0.056531seconds

```

Figure 2. Different threads runtime of different images

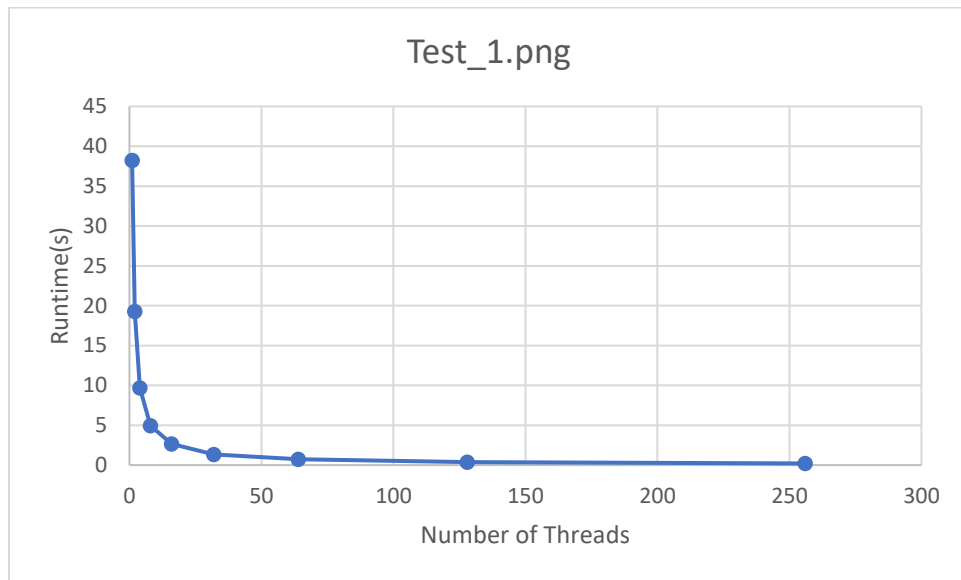


Figure 3. Plot of graph of runtime versus number of threads of Test\_1.png

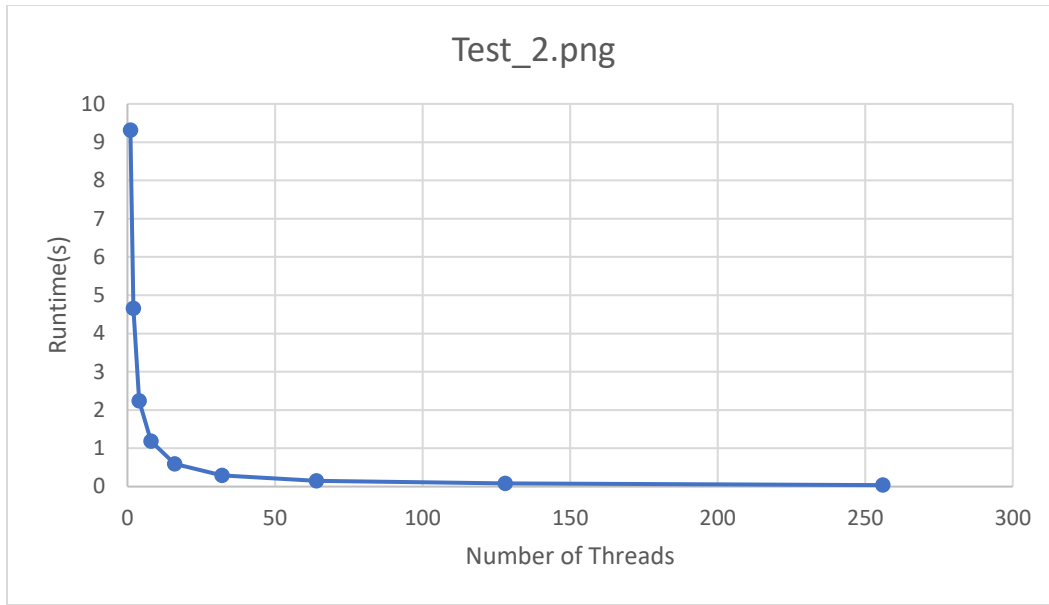


Figure 4. Plot of graph of runtime versus number of threads of Test\_2.png

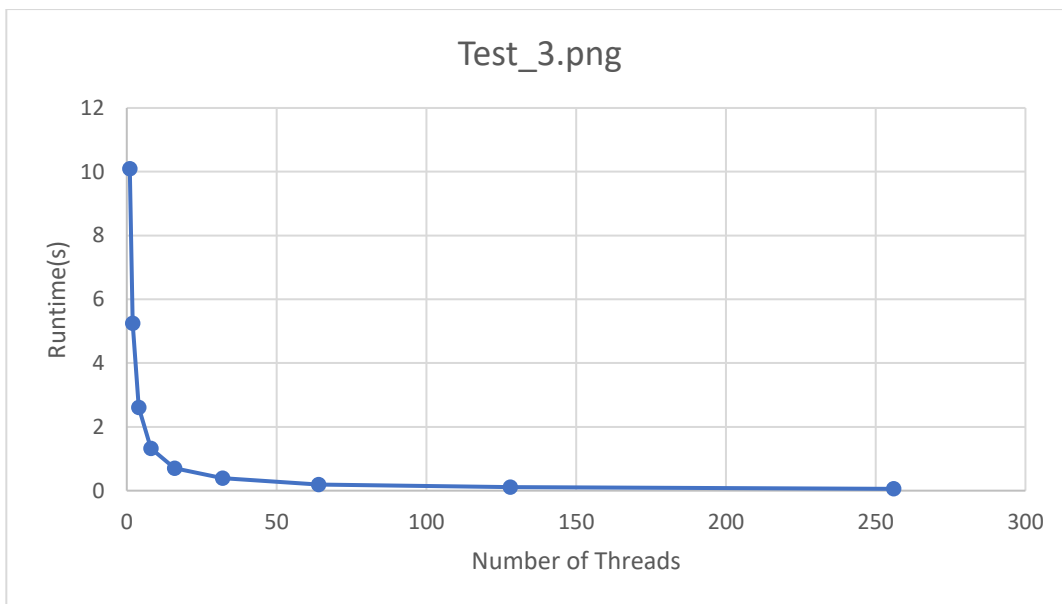


Figure 5. Plot of graph of runtime versus number of threads of Test\_3.png

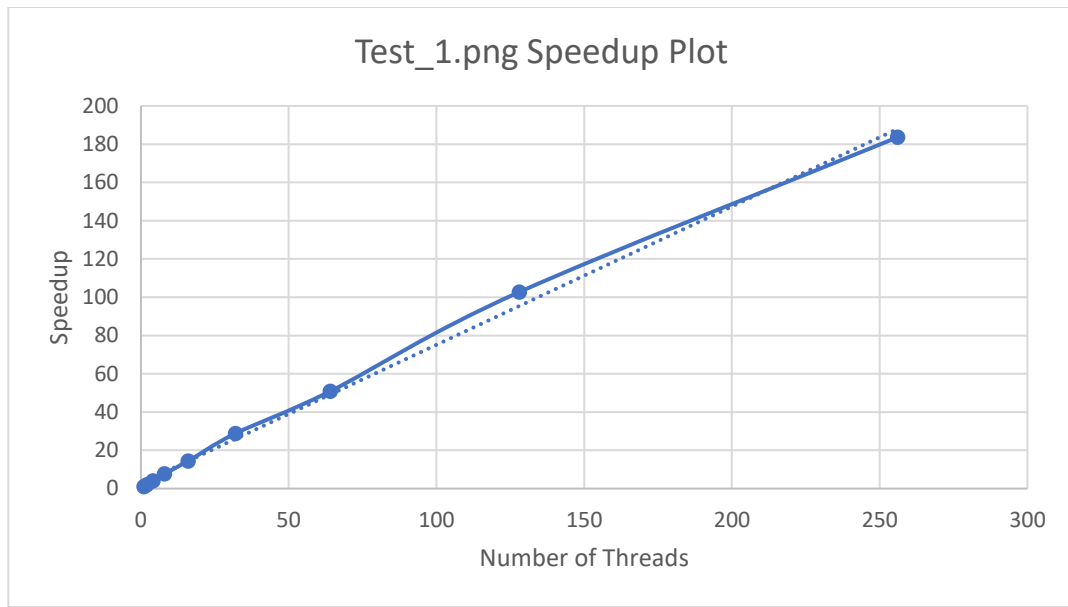


Figure 6. Plot of graph of speedup versus number of threads of Test\_1.png

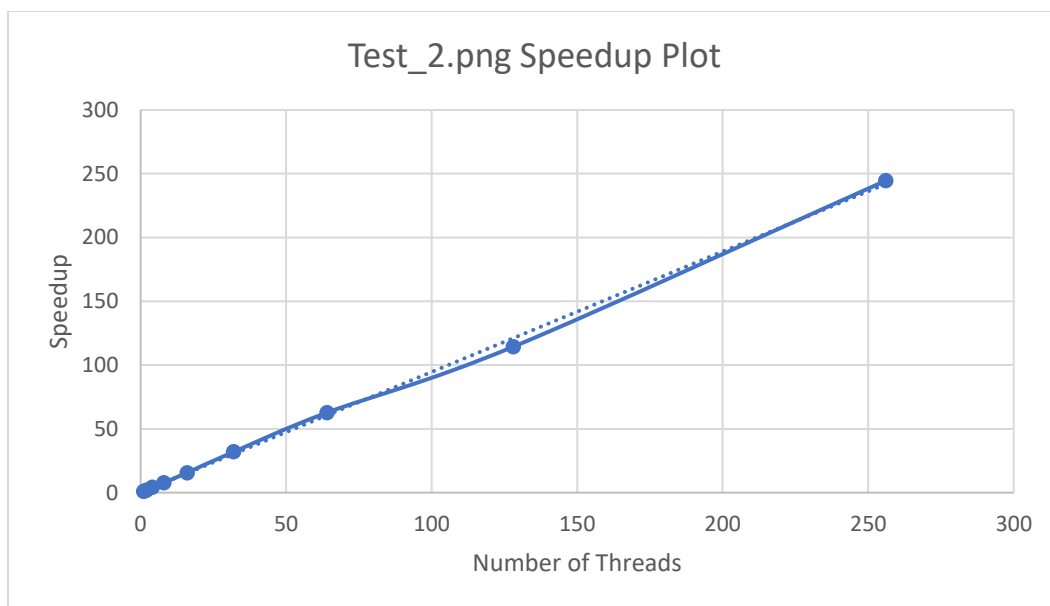


Figure 7. Plot of graph of speedup versus number of threads of Test\_2.png

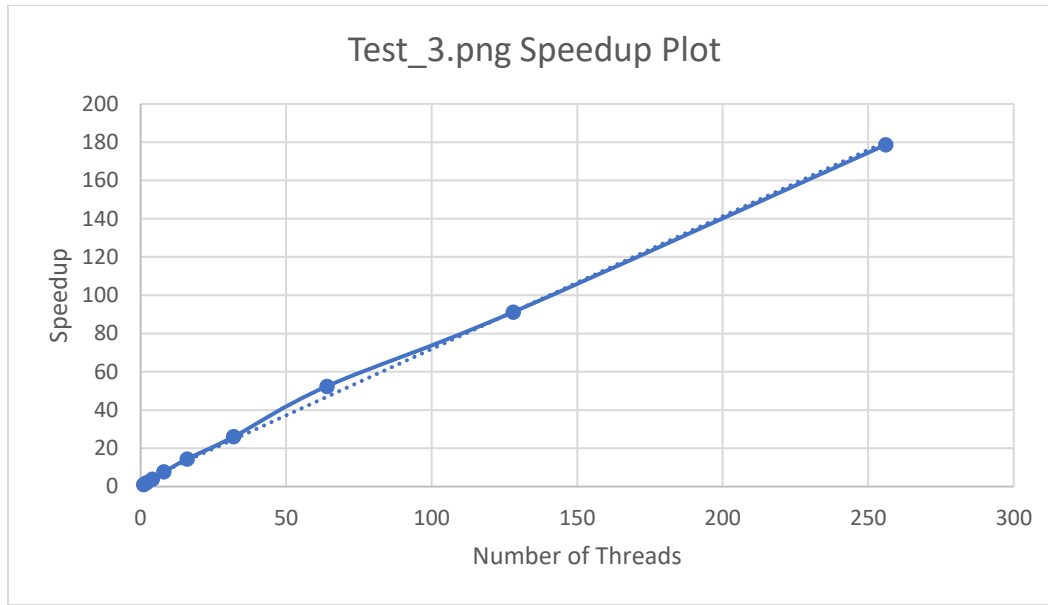


Figure 8. Plot of graph of speedup versus number of threads of Test\_3.png

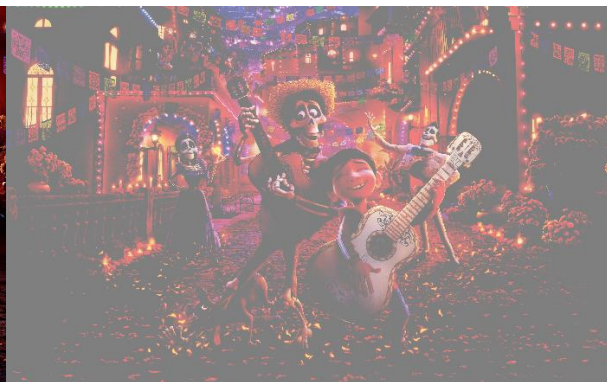
In the graphs above we can see that the lines are almost straight, which means that the speedup value is almost linear to the number of threads. This is because we are using the one block multiple threads method. The thread we tested increases by 2 times each time and the speedup calculated increases by around 2 times as well. As the threads increasing the operations that we need will become  $\frac{N_{parallel}}{\text{Number of thread}} + N_{serial}$ . Since the number of threads are increasing exponentially which is  $2^n$  in X direction. Because of that the speed up according to the increasing trend for the number of threads will seems like linear. As the threads increasing the serial part computation will become the dominant.

Three groups of images comparisons are listed below:

Original Image:



Rectified Image:



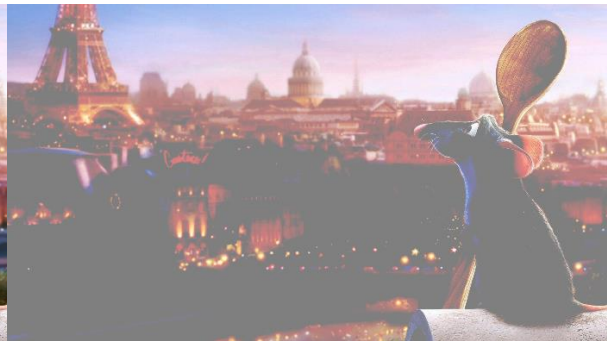
Size  
15.3 MB

Dimensions  
3840 x 2400



Size  
5.4 MB

Dimensions  
3840 x 2400



Size  
2.6 MB

Dimensions  
1920 x 1080



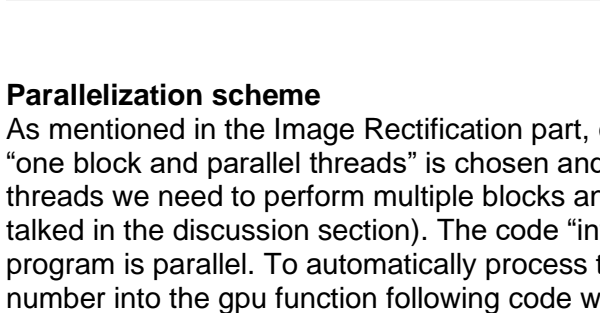
Size  
1.3 MB

Dimensions  
1920 x 1080



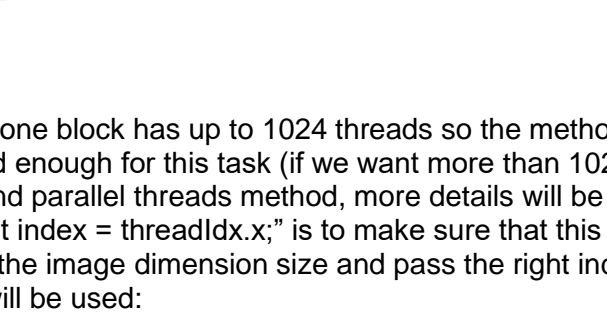
Size  
3.2 MB

Dimensions  
1920 x 1200



Size  
2.3 MB

Dimensions  
1920 x 1200



### Parallelization scheme

As mentioned in the Image Rectification part, one block has up to 1024 threads so the method “one block and parallel threads” is chosen and enough for this task (if we want more than 1024 threads we need to perform multiple blocks and parallel threads method, more details will be talked in the discussion section). The code “int index = threadIdx.x;” is to make sure that this program is parallel. To automatically process the image dimension size and pass the right index number into the gpu function following code will be used:



```

126 // each time run size[s] threads until task finished
127 int total_pixel = width * height / 4; // for output
128 int quotient = (total_pixel + size[s] - 1) / size[s];
129 // start timer
130 timer.Start();
131 for (int j = 0; j < quotient; j++) {
132     pool <<1, size[s] >>> (input_image, output_image, width, j, size[s]);
133 }
134 cudaDeviceSynchronize();
135 timer.Stop();

```

Figure 9. Screenshot of code

As shown in the code fragment above, since the methodology is that each thread will process one pixel in the output image, we need to calculate how many threads that we need for total to fully produce the output image. The total number of pixels for the output image is  $\frac{1}{4}$  of the total number of pixels for the input image since the 2x2 max pooling is to take the maximum value in a 2x2 square pixels (4 pixels) and write it into one pixel. The reason why we need the line 128 showing in the code above is that some image size may not be able to divide the size of thread of threads totally and since it is an integer operation it will lose the remainder of the quotient. As we traverse all the number in the quotient, we get which round that it is for processing the image and we pass it into the function running in GPU.

```

30 global __void pool(unsigned char *input_image, unsigned char *output_image, unsigned width, int times, int thread_size) {
31     unsigned char lb, rb, lt, rt, max;
32     int index = threadIdx.x;
33     unsigned pixel_number_in_array = 0; // This is the current pixel number in the array formed by image pixels.
34
35     if (index < thread_size) {
36         for (int i = 0; i < 4; i++) {
37
38             pixel_number_in_array = (times * thread_size + index) * 2; // (times*thread_size + index): output current thread number; *2 since # of input width / # of
39             pixel_number_in_array += (width * (pixel_number_in_array / width)); // pixel_number_in_array/width represents how many rows we have passed(round down).
40             // If we just multiply by 2 then we would only increase in one dimension (X) and miss the increase in the other dimension (Y).
41             // The increase in the other dimension is like a width scale that is we want to increase 1 in Y dimension we need to increase width units in X dimension
42             // So we need to use PixelNumberInArray/width cast to int to know how many unit we should increase to Y.
43             lb = input_image[pixel_number_in_array * 4 + i];
44             rb = input_image[(pixel_number_in_array + 1) * 4 + i];
45             lt = input_image[(pixel_number_in_array + width) * 4 + i];
46             rt = input_image[(pixel_number_in_array + width + 1) * 4 + i];
47             max = lb;
48             if (rb > max)
49                 max = rb;
50             if (lt > max)
51                 max = lt;
52             if (rt > max)
53                 max = rt;
54
55             output_image[(thread_size * times + index) * 4 + i] = max;
56         }
57     }
58 }

```

Figure 10. Screenshot of code

The code snippet shows the function for executing the pooling operation in the GPU. As mentioned before the index = threadIdx.x is for the parallel programming. The for loop on line 36 is because each pixel has 4 value (1 byte each) inside which are R, G, B and transparency. The line 38 to line 46 is to locate 2x2 pixels in the 1-D array, the methodology is that we want to use the output image pixel location to locate the 2x2 pixels in the input image. The pixel\_number\_in\_array is to get the first pixel in the 2x2 pixels squares.



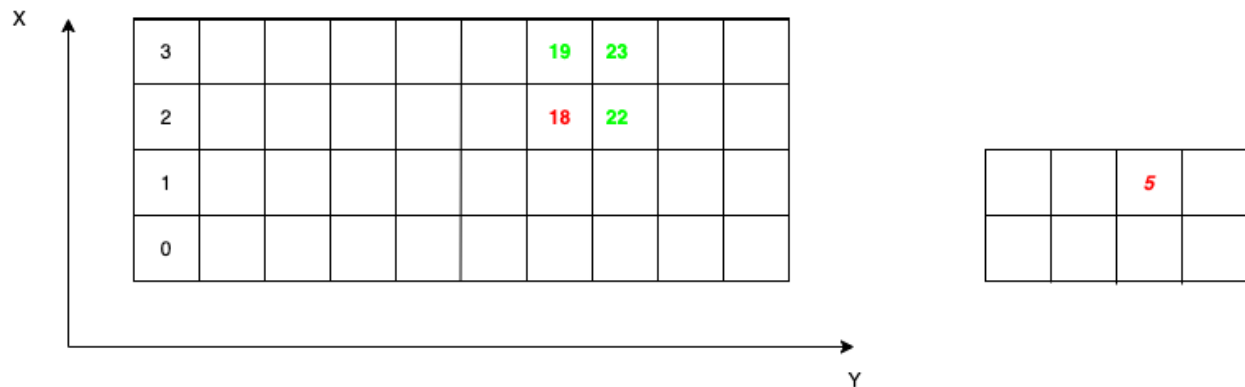


Figure 11. Plot of logic explained

As shown in the picture above, the left is the output image and the right is the original image, if we want to use 5 to relocate the 18, we need to use the equation:

$$5 \times 2 + 4 \times \text{floor}(5 \times 2 \div 4) = 18$$

If we want to locate the other pixel in the 2x2 square (19, 22, 23) we simply need to add the 1 and/or width (width = 4) to it.

The result shows that our algorithm is right as the comparison between the output image and the provided test image is really close.

```
Microsoft Visual Studio Debug Console
Images are equal (MSE = 0.000000, MAX_MSE = 0.000010)
C:\Users\as785\source\repos\ECSE420_Lab0\x64\Debug\Test_Images.exe (process 13940) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 12. Test image equality

## Speedup plots

The figure showing below is the result for running the program in multiple threads. We used the data to do the speed up analysis.

```

Pool image Test_1.png using 1 thread(s): 22.690800seconds
Pool image Test_1.png using 2 thread(s): 11.498417seconds
Pool image Test_1.png using 4 thread(s): 5.875898seconds
Pool image Test_1.png using 8 thread(s): 3.024880seconds
Pool image Test_1.png using 16 thread(s): 1.539348seconds
Pool image Test_1.png using 32 thread(s): 0.798632seconds
Pool image Test_1.png using 64 thread(s): 0.395509seconds
Pool image Test_1.png using 128 thread(s): 0.236551seconds
Pool image Test_1.png using 256 thread(s): 0.119531seconds
Pool image Test_2.png using 1 thread(s): 5.124331seconds
Pool image Test_2.png using 2 thread(s): 2.568944seconds
Pool image Test_2.png using 4 thread(s): 1.310789seconds
Pool image Test_2.png using 8 thread(s): 0.670093seconds
Pool image Test_2.png using 16 thread(s): 0.334481seconds
Pool image Test_2.png using 32 thread(s): 0.175975seconds
Pool image Test_2.png using 64 thread(s): 0.086940seconds
Pool image Test_2.png using 128 thread(s): 0.042848seconds
Pool image Test_2.png using 256 thread(s): 0.027283seconds
Pool image Test_3.png using 1 thread(s): 5.711767seconds
Pool image Test_3.png using 2 thread(s): 2.852777seconds
Pool image Test_3.png using 4 thread(s): 1.461013seconds
Pool image Test_3.png using 8 thread(s): 0.741572seconds
Pool image Test_3.png using 16 thread(s): 0.369263seconds
Pool image Test_3.png using 32 thread(s): 0.187673seconds
Pool image Test_3.png using 64 thread(s): 0.093073seconds
Pool image Test_3.png using 128 thread(s): 0.046731seconds
Pool image Test_3.png using 256 thread(s): 0.030029seconds

```

Figure 13. Different threads runtime of different images

As mentioned in the previous section, the equation for speed up is:

$$SpeedUp = \frac{T_{serial}}{T_{parallel}}$$

| N. of Thread | Test1 Time | Speed up 1 | Test2 Time | Speed up 2 | Test3 Time | Speed up 3 |
|--------------|------------|------------|------------|------------|------------|------------|
| 1            | 22.6909    | 1          | 5.124331   | 1          | 5.711767   | 1          |
| 2            | 11.498417  | 1.97339338 | 2.568944   | 1.99472273 | 2.85277    | 2.00218279 |
| 4            | 5.875898   | 3.86169059 | 1.310789   | 3.90934849 | 1.461013   | 3.90945666 |
| 8            | 3.02488    | 7.50142154 | 0.670093   | 7.64719375 | 0.741572   | 7.70224199 |
| 16           | 1.539348   | 14.7405915 | 0.334881   | 15.3019461 | 0.369263   | 15.4680187 |
| 32           | 0.798632   | 28.4122099 | 0.175975   | 29.1196534 | 0.187673   | 30.4346763 |
| 64           | 0.395509   | 57.3713873 | 0.08694    | 58.9410053 | 0.093073   | 61.3686783 |
| 128          | 0.236551   | 95.9239234 | 0.042848   | 119.593237 | 0.046731   | 122.226509 |
| 256          | 0.119531   | 189.832763 | 0.027283   | 187.821391 | 0.030029   | 190.208365 |

The table above show the time that each test image needs to be processed and the speed up after using multiple threads for parallel processing.

The Figures below show for each test image, the trend of the speed up and the time using for pooling the image with different threads.

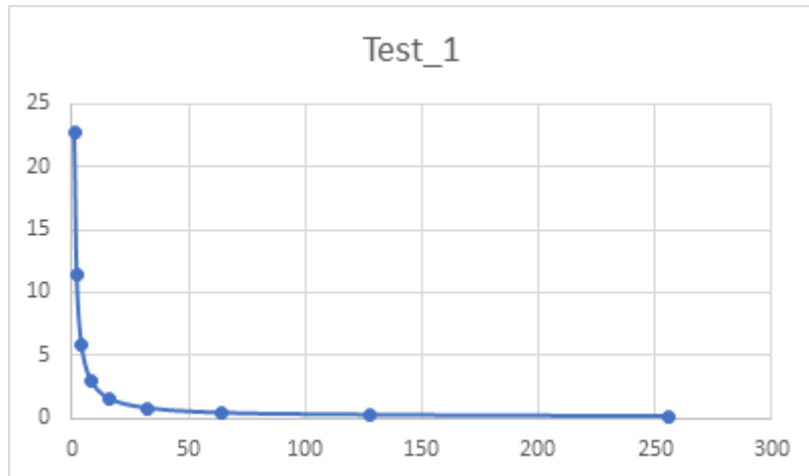


Figure 14. Plot of graph of runtime versus number of threads of Test\_1.png

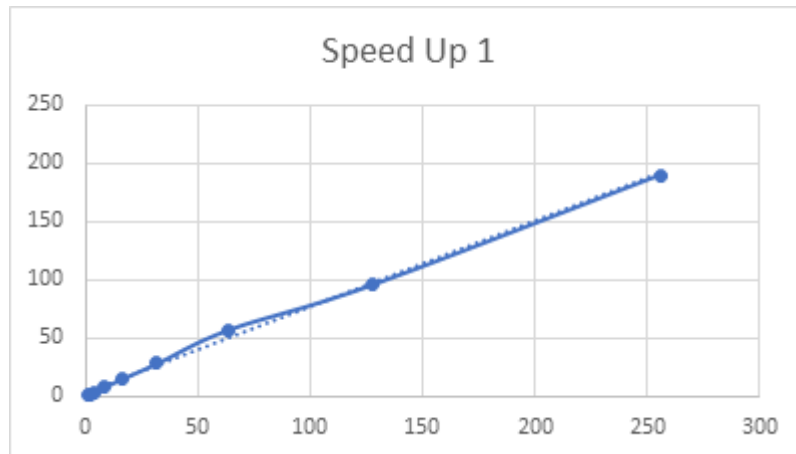


Figure 15. Plot of graph of speedup versus number of threads of Test\_1.png

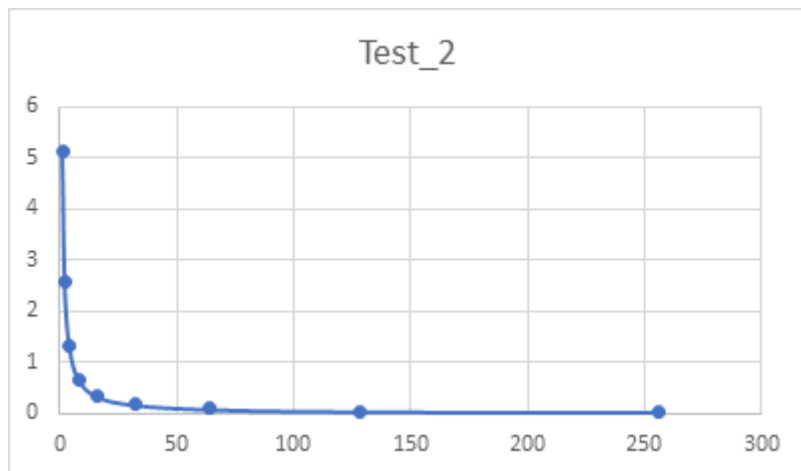


Figure 16. Plot of graph of runtime versus number of threads of Test\_2.png

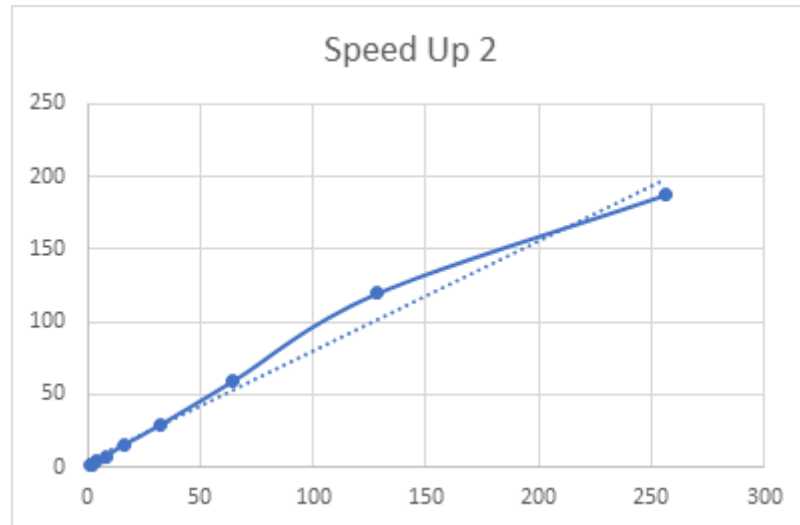


Figure 17. Plot of graph of speedup versus number of threads of Test\_2.png

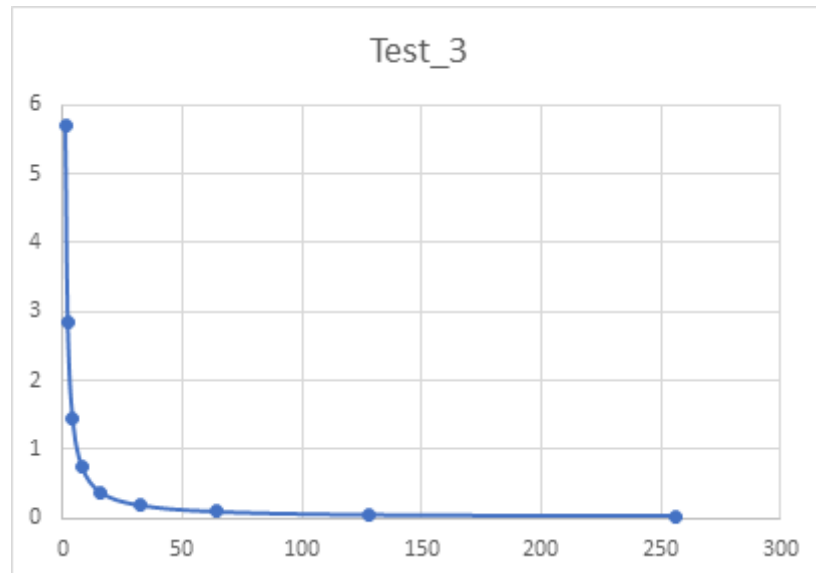


Figure 18. Plot of graph of runtime versus number of threads of Test\_3.png

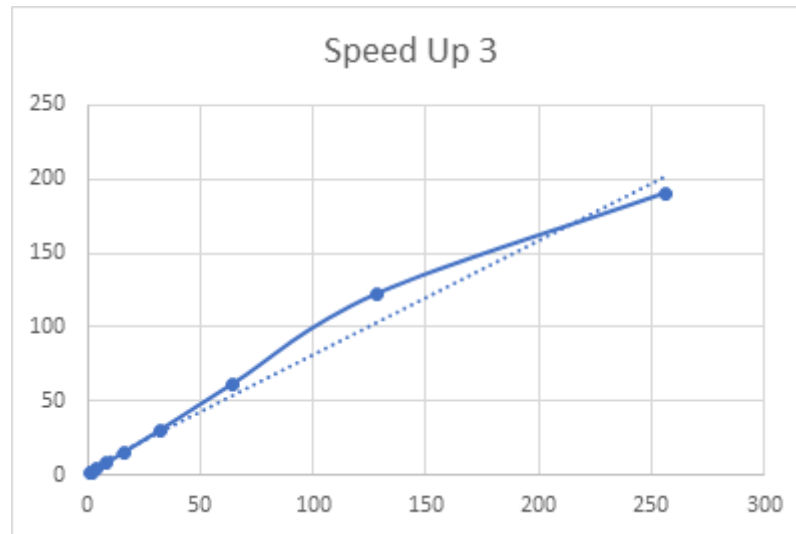


Figure 19. Plot of graph of runtime versus number of threads of Test\_3.png

As we can see in the Figures above, if we look at the Figures for showing the speed up for pooling the image, we can see that the speed up is increasing linearly with the increase of the number of threads working in parallel. It is because that we are using one block multiple threads then with the threads increasing, before we need to process the image using

$N_{parallel} + N_{serial}$  operation now we got is  $\frac{N_{parallel}}{\text{Number of thread}} + N_{serial}$ .

We can also see that in the Figure showing time decreasing with the number of threads increasing. First it will drop significantly and then it will remain steady. It means that the multiple threads will increase the performance significantly, but the serial part will still take some time to process and when the threads number is increasing the time that the program needs are almost all from the serial part.

The results are showing below

4712ie1140fe72l6rtgsv3pnq3ka23c2


Dimensions: 1920×1200  
Color space: RGB  
Alpha channel: No

▶ Name & Extension:

▶ Comments:

▶ Open with:

▼ Preview:



▶ Sharing & Permissions:


Dimensions: 960×540  
Color space: RGB  
Alpha channel: No

▶ Name & Extension:

▶ Comments:

▶ Open with:

▼ Preview:




Title: PDF Creator  
Dimensions: 3840×2400  
Color space: RGB  
Alpha channel: No

▶ Name & Extension:

▶ Comments:

▶ Open with:

▼ Preview:



▶ Sharing & Permissions:


Dimensions: 1920×1080  
Color space: RGB  
Alpha channel: No

▶ Name & Extension:

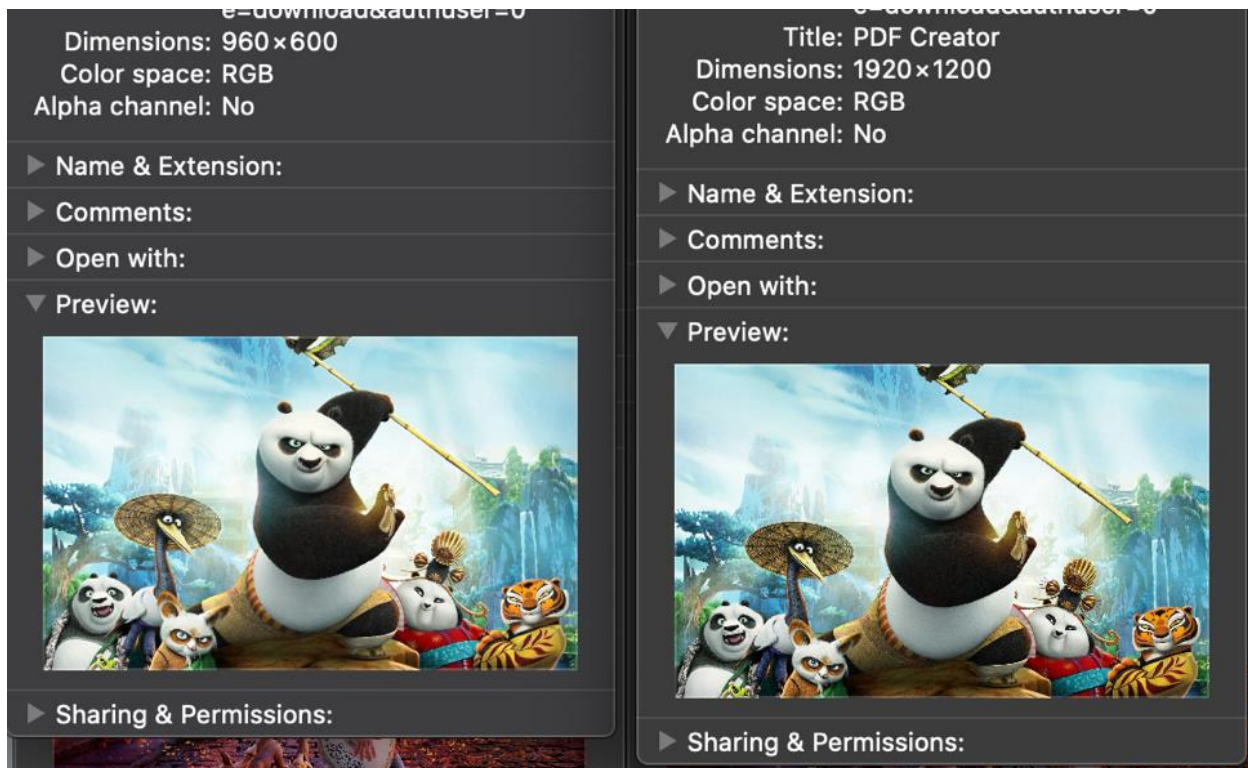
▶ Comments:

▶ Open with:

▼ Preview:







## Further Discussion

Since we use the method "one block with parallel threads" in this assignment, our code does not support thread size greater than 1024 because the maximum number of threads in a block is 1024. If we want to use more than 1024 threads to run at a time, we need to use the method "parallel blocks with parallel threads" instead, which we need to take into consideration of the number of blocks as well. In that situation, there are also two more changes in the code. First is that we need to define index as "threadIdx.x + blockIdx.x \* blockDim.x". Second is that when we call the function rectify or pool, we should change `<<<1, SIZE_OF_THREAD>>>` to `<<<SIZE/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>`. We also need to also consider how to automatically adjust the block numbers, number of threads per block. In this case, we need to modify our original code. We still need to first compute the number of rounds that we need for the parallel function to be called but more we need to use the size of the thread for example 1025 then we need to use 1025 to do an integer division on 1024 and add the quotient to the block number (block number is initialized to be 1). Then we get the total blocks that we need for the parallel programing. Now we need to know what the value of threads per block is. Initially, the value in variable `thread_per_block` will become 1025 as our example. Then we need to use this mod block number that we got for the previous step and if the remainder is not equal to zero, we need to add 1 to the thread per block until the remainder is equal to 0. At last the thread per block is equal to the quotient of integer division of thread per block and block number.

After this operation, we can get this:

$$thread\ size = block\ number \times thread\ per\ block$$