Parallel Computing
ECSE 420

Roman Andoni - 260585085
Yordan Neshev - 260587938

MULTITHREADING AND SIGNAL PROCESSING ALGORITHMS

Lab Experiment Report

Prof. Zeljko Zilic
McGill Faculty of Engineering

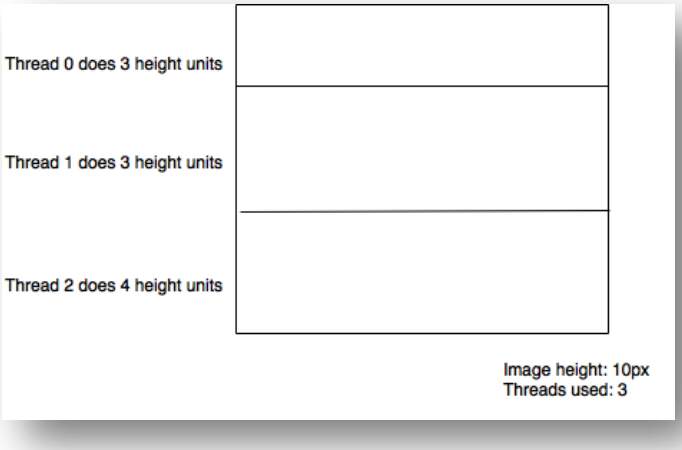
October 4th, 2016

Abstract:

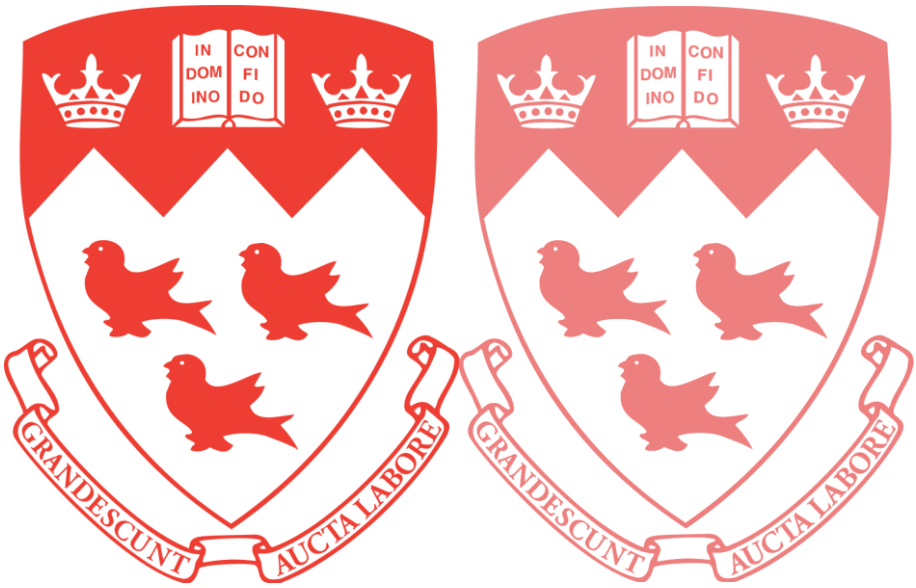
In this experiment, it was required to write several signal processing routines in order to test and demonstrate the effect of parallelism. In fact, three image-processing algorithms were written using Pthreads. All three algorithms were tested using 1 thread and 2, 4, 8, 16 and 32 threads in parallel for purposes of comparison between the time taken with one thread versus multiple threads. Base files were provided with the lab (lodepng files) along with the test and comparison PNG files. Details of parallelization scheme and results will be discussed in each section.

Rectification

Rectification consisted of scanning through each pixel of the image and changing the value of pixels, hence changing their RGB properly. Specifically, if a value of pixel was found to be lower than 127, it was required to set it to 127. Otherwise, the pixel value must be untouched. The algorithm that we implemented parallelizes the computational part only, omitting the loading/reading to the new image file. This was done due to the nature of the calculations, where it was possible to divide the image into several parts and allow each thread to scan through each array of pixels belonging to the image and change the pixel values. Hence, before issuing threads, it was required to write a helper function that would take the height of the image and divide it into almost equal parts. Specifically, the function is named **sliceHeights**, which takes the height of the loaded image (via lodepng) and slices it into several heights, depending on the amount of threads. Each thread then is assigned its own height in the image in the thread argument struct. Finally, each new thread is created with the respected input image height and computation happens in parallel on different heights. See the image below for details.



To show the actual output of the algorithm, below are demonstrated an input and an output for the rectification:



Original → Rectified

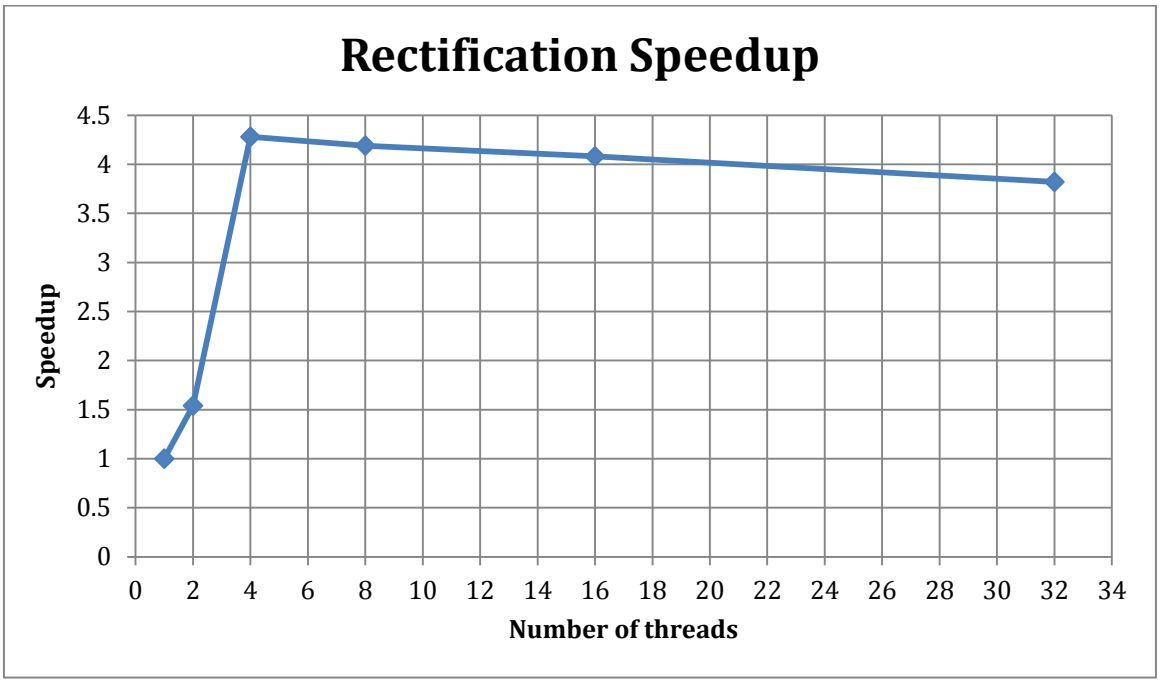
As for testing purposes, several data records were taken at each run. We launched the algorithm with a fixed number of threads 5 times to take the average run time using n threads, where n= {1, 2, 4, 8, 16, 32}.

n = 1. Times in (ms): {16.04, 20.17, 15.41, 20.23, 20.03}	Avg time @ n = 1 : 18.42 ms
n = 2. Times in (ms): {13.14, 13.05, 13.10, 10.32, 9.98}	Avg time @ n = 2 : 11.92 ms
n = 4. Times in (ms): {4.41, 4.20, 4.13, 4.76, 4.01}	Avg time @ n = 4 : 4.30 ms
n = 8. Times in (ms): {4.35, 4.42, 4.38, 4.53, 4.36}	Avg time @ n = 8 : 4.40 ms
n = 16. Times in (ms): {4.61, 4.43, 4.46, 4.59, 4.49}	Avg time @ n = 16 : 4.51 ms
n = 32. Times in (ms): {4.87, 5.12, 4.74, 4.73, 4.67}	Avg time @ n = 32 : 4.82 ms

Using the average values at each number of threads used, we can now compute the speedup:

Speedup using:
2 threads = 18.42/11.92 = 1.54
4 threads = 18.42/4.30 = 4.28
8 threads = 18.42/4.40 = 4.19
16 threads = 18.42/4.51 = 4.08
32 threads = 18.42/4.82 = 3.82

Finally, below are the graphs representing speedup vs number of threads used.

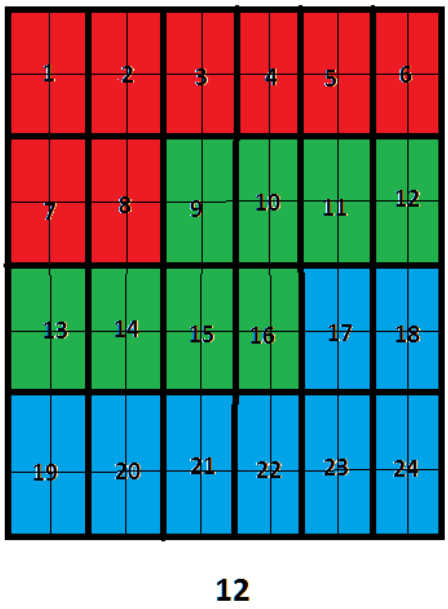


As it can be observed, the peak speedup of 4.28 happens when we execute the rectification algorithm using 4 threads. After this, the speedup decreases. This phenomenon is observed throughout rectifying, pooling and convolution and will be explained at the end of the report.

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. This way, the image gets compressed to a new width and height two times smaller than the original image. The algorithm that we implemented parallelizes the computational part only, omitting the loading/reading to the new image file. This was done due to the nature of the calculations, 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. Hence, before issuing threads, it was required to write a helper function that would take the width and the height of the image and divide it into almost equal parts. Specifically, the function is named **sliceBlocks**, which takes the width and height of the loaded image (via lodepng) and slices it into several arrays of 2x2 blocks, depending on the amount of threads. Each thread then is assigned its amount of blocks in the image in the thread argument struct. The start index offset of the block is calculated by each thread with the help of the thread number. Finally, each new thread is created with the respected

input amount of 2x2 blocks and computation happens in parallel on different sections of the image. See the 8x12 pixels image below for details.

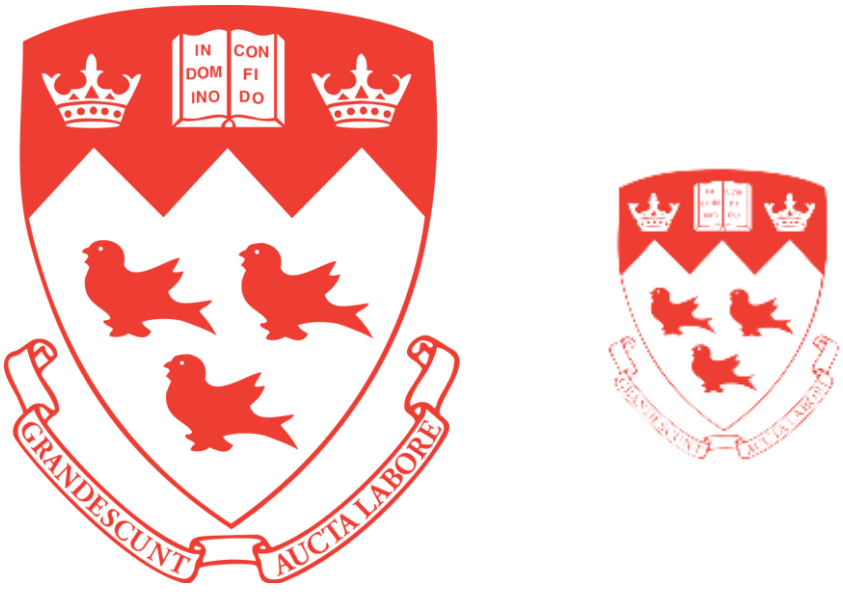


8x12 pixels image separated in 24 blocks of 2x2 pixels

If we use 3 threads, each one is going to be assigned 8 2x2 blocks to compress.

Thread 1 gets assigned blocks 1 to 8 (red).
Thread 2 gets assigned blocks 9 to 16 (green).
Thread 3 gets assigned blocks 17 to 24 (blue).

To show the actual output of the algorithm, below are demonstrated an input and an output for the pooling:



Original → Pooled

As for testing purposes, several data records were taken at each run. We launched the algorithm with a fixed number of threads 5 times to take the average run time using n threads, where n= {1, 2, 4, 8, 16, 32}.

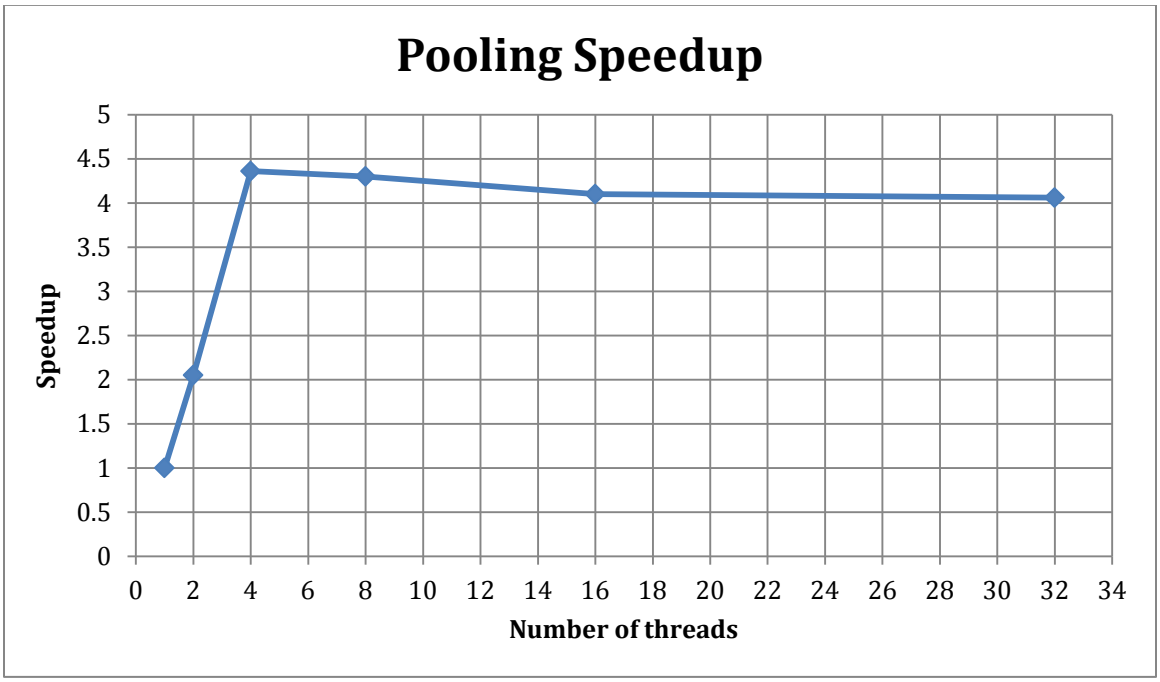
n = 1. Times in (ms): {32.74, 31.23, 27.29, 28.22, 27.88}	Avg time @ n = 1 : 29.59ms
n = 2. Times in (ms): {14.41, 16.11, 14.24, 14.30, 13.10}	Avg time @ n = 2 : 14.43ms
n = 4. Times in (ms): {6.64, 6.73, 6.66, 7.04, 6.82}	Avg time @ n = 4 : 6.78 ms
n = 8. Times in (ms): {6.92, 6.88, 6.94, 6.81, 6.86}	Avg time @ n = 8 : 6.88 ms
n = 16. Times in (ms): {7.96, 6.99, 7.21, 6.26, 7.67}	Avg time @ n = 16 : 7.22 ms
n = 32. Times in (ms): {7.23, 7.39, 7.27, 7.31, 7.25}	Avg time @ n = 32 : 7.29 ms

Using the average values at each number of threads used, we can now compute the speedup:

Speedup using:

2 threads = 29.59 / 14.43 = 2.05
4 threads = 29.59 / 6.78 = 4.36
8 threads = 29.59 / 6.88 = 4.30
16 threads = 29.59 / 7.22 = 4.10
32 threads = 29.59 / 7.29 = 4.06

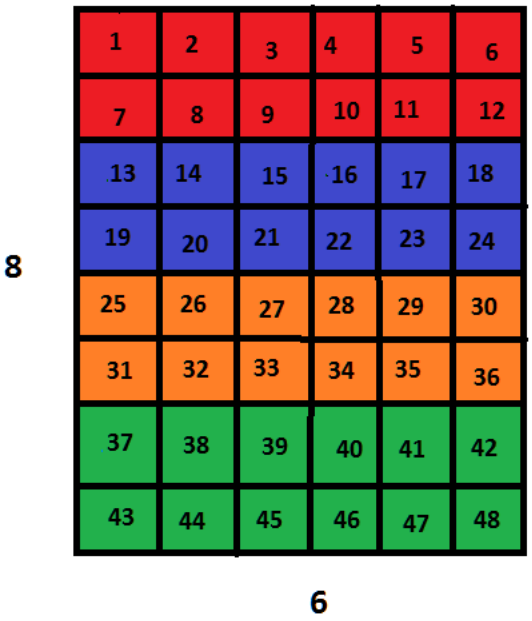
Finally, below are the graphs representing speedup vs number of threads used.



As it can be observed, the peak speedup of 4.36 happens when we execute the pooling algorithm using 4 threads. After this, the speedup decreases. This phenomenon is observed throughout rectifying, pooling and convolution and will be explained at the end of this report.

Convolution

Convolution consisted in producing a new image whose pixel values represent the weighted sum of the input image pixels and their neighbors. Since we are using 3x3 matrices in the current problem and the valid padding of convolution reduces the width (m-1) and height (n-1) of the initial image, the output image is of size m-2 by n-2. The algorithm that we implemented parallelizes the computational part only, omitting the loading/reading to the new image file. This was done due to the nature of the calculations, 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 convolute this selection of pixels individually. Hence, before issuing threads, it was required to write a helper function that would take the total amount of pixels in the output image and divide it into equal parts. In the case where an equal division is not possible, the last thread is given the remainder of the blocks to process. Specifically, the function is named **sliceBlocks**, which takes the amount of pixels in the output image and slices it into several equal portions of pixels, depending on the amount of threads. Each thread then is assigned its block of pixels in the image in the thread argument struct and the start offset index of its block. Finally, each new thread is created with the respected amount of blocks and the blocks offset and computation happens in parallel on different sections of the image. See the 8x12 pixels image below for details.



8x12 pixels image

If we use 4 threads, each one is going to be assigned 12 pixels to convolute.

Thread 1 gets assigned pixels 1 to 12 (red).
Thread 2 gets assigned pixels 13 to 24 (blue).
Thread 3 gets assigned pixels 25 to 36 (orange).
Thread 4 gets assigned pixels 37 to 48 (green).

To show the actual output of the algorithm, below are demonstrated an input and an output for the convolution:

Original → Convoluted

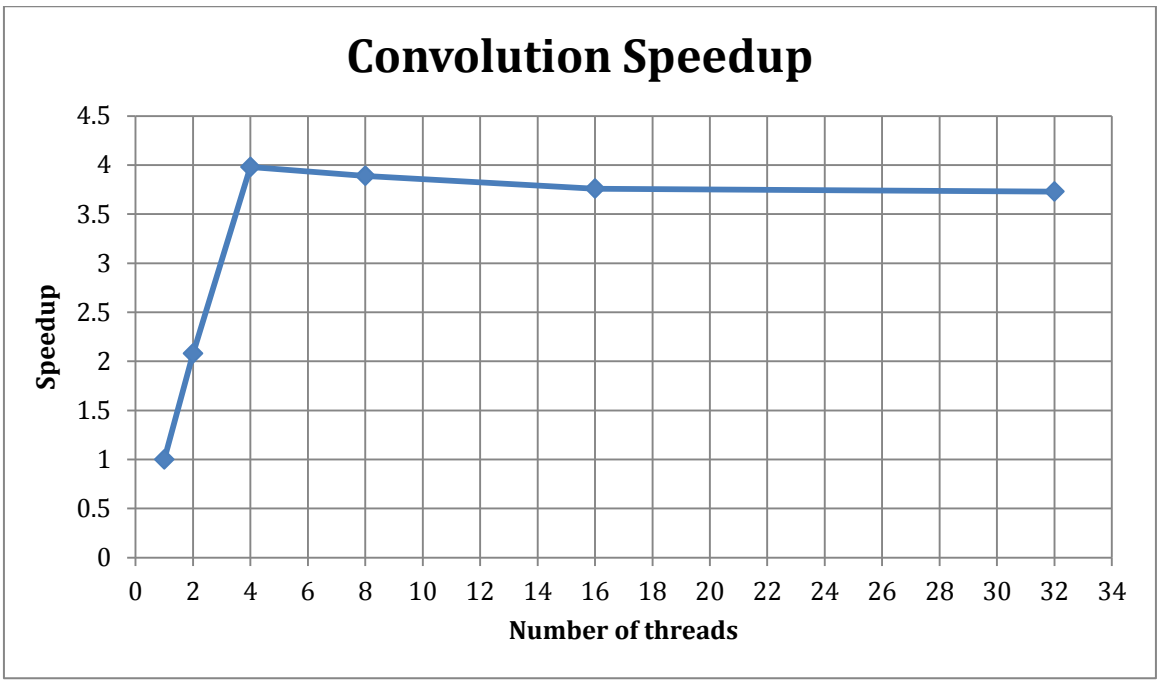
As for testing purposes, several data records were taken at each run. We launched the algorithm with a fixed number of threads 5 times to take the average run time using n threads, where n= {1, 2, 4, 8, 16, 32}.

n = 1. Times in (ms): {189.86, 211.02, 216.56, 201.63, 203.00}	Avg time @ n = 1 : 204.41ms
n = 2. Times in (ms): {100.38, 94.90, 94.85, 99.53, 100.80}	Avg time @ n = 2 : 98.09ms
n = 4. Times in (ms): {51.02, 50.69, 52.21, 50.95, 52.20}	Avg time @ n = 4 : 51.41ms
n = 8. Times in (ms): {51.01, 52.06, 54.29, 52.00, 53.11}	Avg time @ n = 8 : 52.49ms
n = 16. Times in (ms): {52.13, 57.12, 55.17, 52.33, 55.14}	Avg time @ n = 16 : 54.38ms
n = 32. Times in (ms): {53.63, 53.75, 58.13, 55.67, 52.34}	Avg time @ n = 32 : 54.70ms

Using the average values at each number of threads used, we can now compute the speedup:

Speedup using:
2 threads = 204.41 / 98.09 = 2.08
4 threads = 204.41 / 51.41 = 3.98
8 threads = 204.41 / 52.49 = 3.89
16 threads = 204.41 / 54.38 = 3.76
32 threads = 204.41 / 54.70 = 3.73

Finally, below are the graphs representing speedup vs number of threads used.



As it can be observed, the peak speedup of 3.98 happens when we execute the convolution algorithm using 4 threads. After this, the speedup becomes. This phenomenon is observed throughout rectifying, pooling and convolution and will be explained at the end of this report.

Conclusion

As we saw in all 3 experiments, speedup was optimized when the program execution was assigned to 4 separate threads. This was expected since the machine the tests were running was a 2011 MacBook Pro with an Intel Core i7, a quad core CPU. Given that this CPU has 4 cores, it is optimized to run one thread per core. Obviously, the results were disturbed by other active processes running on the machine. Nevertheless, adding more threads for the execution created bottleneck situation, where instead of one thread running per core freely, we have several threads running on each core but in a queue, fighting for CPU clock time, hence slowing down the whole processing of the data. This confirms the fact that having fewer threads for the processing will not fully utilize the CPU resource (in case of 1 thread and 2 threads). On the other hand, having more threads would cause threads competing for the CPU resource, thus increasing context switching, causing overall execution time to increase and speedup to decrease. The similar behavior is assumed to persist in further experiments.