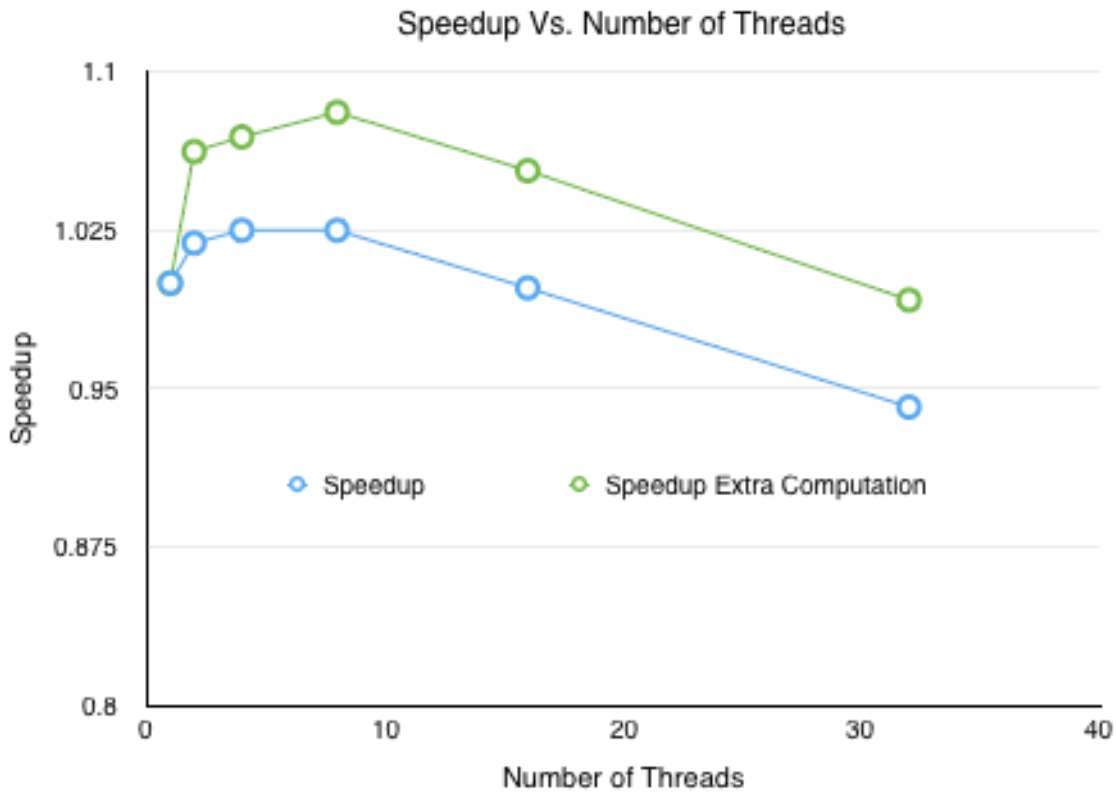
ECSE-420: Lab 1
Bernard Kaminski 260444815
Tuan-Anh Nguyen 260532842

# ECSE-420: Lab 1 Report

All tests and programs were run on a virtual machine to produce the speedup graphs. The virtual machine was run on a Macbook Pro (2.6 GHz Intel Core i5 processor, 8GB 1600 MHz DDR3 memory). The virtual machine was allocated 2GB memory with 2 processors. All programs were run 10 times and the execution was averaged out over these 10 runs. In order to track the execution time, the command time was used when running the binary for the program. The openMP thread framework was used to achieve parallelism in programs and so a lot of threading features were abstracted from us.

## 1. Rectification

Image rectification is a parallelizable process whereby each single pixel can be processed independently from one another. One pixel does not depend on any other so a set of pixels can be assigned to a thread and it can process each pixel and set the appropriate value in the output image. Sequential (1 thread) runtime was compared versus parallel (2, 4, 8, 16, 32 threads) runtime and small speedup can be noticed. The parallelization scheme would be to take every row of the matrix and perform the check on each pixel of the row. This way, a thread can get a certain number of rows each and set the appropriate values for the output image.



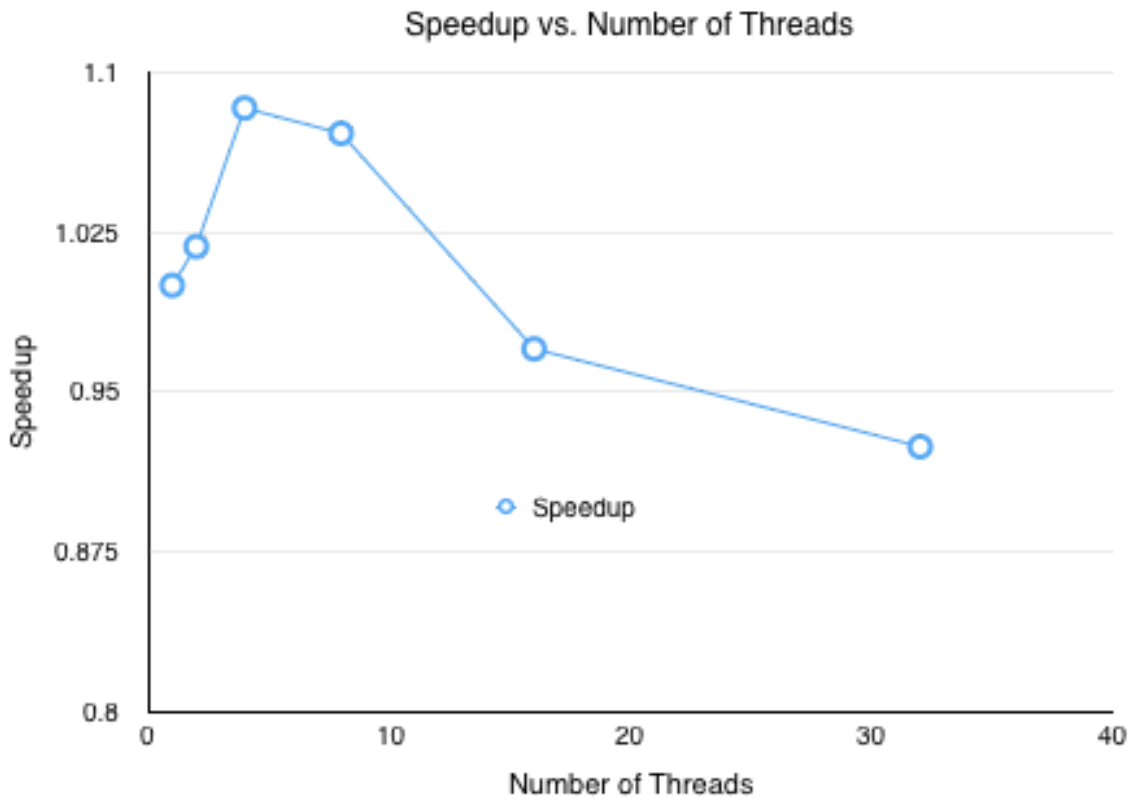

It is logical that increasing the number of threads to introduce parallelism would increase the performance (higher speedup). In blue is the original program and there is little speedup that is noticeable. At first, we were skeptical about the speedup because it was so small. This is probably due to the simple computations (additions) and small input image (only 994 x 998 pixels) so in order to verify that there is actual speed up, we wrote the same program except

added expensive useless computations within the nested for loops. Expensive computations included trigonometric functions (sine and cosine) as well as the power function (a^x). This program is in green in the figure above and shows that there is a clear speedup when going from 1 thread to 2 and 4 threads.

Another observation is that when the number of threads is significantly high, there is expected slowdown due to the overhead to create threads. We begin to see slowdown starting when there are 16 threads. Another interesting thing is that trying to use a ridiculously high number of threads (256 threads) does not slow down the program tremendously. The slowdown is capped due to the openMP framework. It is “smart” when scheduling tasks for threads and realizes if there are too many threads and that some will be wasted, it will cap the number of threads created and used.

## 2. Max-Pooling

Max-pooling consisted of breaking the entire image into 2x2 squares then taking the max of those 2x2 squares and using that result in the new image. Again, this is highly parallelizable because each 2x2 square is independent of each other. We can assign multiple different 2x2 squares to a number of different threads to do parallel work and speedup computation.




Similar to rectification, we can again see the speedup moving from 1 to 2 or 4 threads and a relative decrease in speedup starting with 8 threads. This slowdown is due to the overhead when creating new threads. The parallelization scheme was to take all of an element’s neighbours (to the right, down, diagonal) then take the max and move in indices of 2 such that we don’t overlap regions of 2x2 squares. The structure of the code is two nested for

loops, to traverse all the pixels, then perform the max computation. Since the two for loops don't depend on each other, we can use the openMP construct for parallelizing for loops to achieve a parallel program.

### 3. Convolution

The convolution algorithm was implemented using 4 nested for loops. Therefore, a sequential program with that many number of nested for loops is a very slow implementation. Splitting up the pixels into 3x3 regions and multiplying it by a weight matrix then summing all the results is somewhat independent from other 3x3 regions. There is overlap between all 3x3 regions to be processed so it may be vulnerable to race conditions if the original input image was being written to. However, with convolution, each thread performs a read operation on the original input image so it would be impossible for a race condition to occur.

There is a great increase from going from 1 to 4 threads as seen in the graph below. Again, this is expected because taking 4 "independent" for loops and running them in parallel would result in a faster time than 4 sequential for loops running. The speedup plateaus when going above 16 threads and it is still faster than a sequential program even with all the openMP threading overhead.

Another approach we tried for a parallel program was to parallelize the outer 2 for loops then parallelize again the inner 2 for loops. In other words, a thread would create another thread. This resulted in similar numbers as the graph below (significant increased from 1,2 threads to 4).