Michael Rettus, Griffin Craft
Prof Selina Akter
CSCI 347
11/26/2021

Project 2

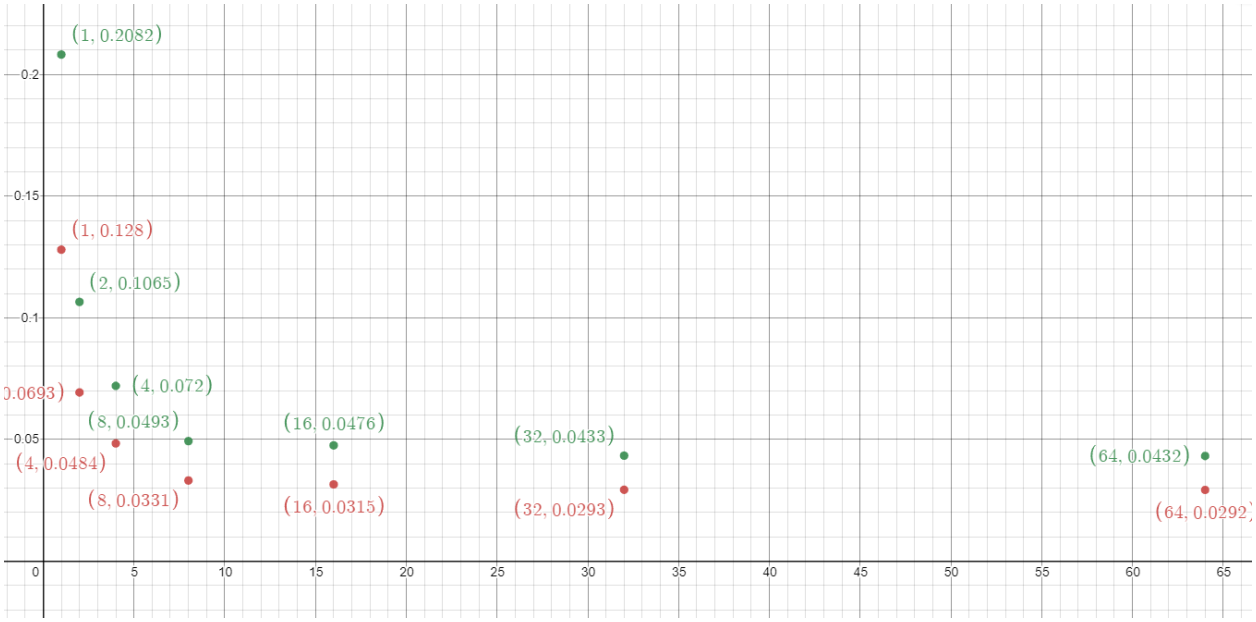
The purpose of this project is to apply a laplacian filter that highlights the edges in an image by detecting regions of rapid intensity change. Areas of significant contrast are more noticeable after the filter intensifies the edges while reducing intensity in areas of similar contrast.

Using concurrency was able to speed up the process, but also added a level of complexity in the process that required the programmer to be fully spatially aware of what was happening to the image being read, as it was rendering a new image.

There were two struct to handle the process, PPMPixel held the rgb values for a pixel, the parameter structure had two PPMPixel pointers one to store the value of a prefilter pixel and one for post filter pixel. It also had the width and height of an image w, h followed by start which had the starting point on the image for a thread to begin work and a size telling the thread how much of the image to process so work could be distributed evenly amongst all but the last thread which did equal to or less than the others depending on what work was left.

The threads were implemented using a four loop utilizing numThreads as a variable we could easily change to control the number of threads, memory allocation for each thread (sizeof(*params) * numThreads) and the work (h/numThreads) to be done. Memory had to be allocated to hold the values of the new image as well as the parameter structure for each thread. Each thread would filter the full width of the image structure times h/numthreads height and start at i * h/numthreads along the height to filter the image to store in a new structure. After we used pthread_join to wait for each thread to finish and avoid race conditions.

My experiment I ran 10 tests for each thread count where number of threads included $\{2^0, 2^1 ... 2^6]$ and for both image sizes 800 x 600, and 1080 x 720. For the environment I ran this in it seemed around ten threads you stopped seeing significant improvement as can be seen in the following graph.



The green coordinates represent the 1080 x 720 image average run times x = thread count.
The red coordinates represent the 800 x 600 image average run times x = thread count.

As you can see, adding just a few threads had significant improvement in time performance. Going from 1 to 2 didn't cut run times in half but almost, going from 2 to 4 again saw good improvement but less significant, and then 4 to 8 is the last big change. In fact while 64 showed some improvement over 32 it was so close that it could be within the margin of error of this test. Not included on the graph I pushed the threadcount to the maximum I could by setting it equal to the height of the image and saw a reduction in efficiency. The 800 x 600 image on average ran at .032591 which is between 8 and 16, and the 1080 x 720 image had an average of .048521 which was worse than 8 threads the difference in results probably due to the larger image running more threads showing that you can end up with a loss by using too many threads.

In conclusion, when working in a multicore environment concurrency is a great way to speed up an application, but adds additional complexity and reduced potential as you add more processes. Like other methods of optimization a good understanding of when to use it and how to implement it is key in maximizing it's potential.