Blur Optimisation Experiment Report

Alexander Balinsky CID: 02214992

Abstract

In this experiment, I evaluated the impact of various thread allocation models on performance through the highly parallelisable blur algorithm. This report demonstrates that varying the number of threads for the task and hardware can have a significant performance effect, so it is crucial to strike a balance as both extremes have substantial performance degradation. I ran these tests on the Linux vertex 10 machine at the Imperial Computing Laboratories: HP EliteDesk 800 G4 TWR, Nvidia GPUs (Geforce 730GT 2GB). The processor was the Intel Core i7-8700 3.20 GHz with 6 cores. I performed these tests with no programs running/operating in the background to avoid interference. One algorithm (sector-blur) utilised the exact number of CPU cores as its thread number. To facilitate this, a global constant was set to the number of CPU cores, 6 on this machine. To run the experiments, I developed the command: *"./blur_opt_exprmt #repetition *.jpg"*. The values of repetition and **.jpg* are taken as input, like my test *"./blur_opt_exprmt 100 blurtest_images/BigBenRes.jpg"*. Results are stored in BlurExprmt.txt automatically, image results are stored in the folder *BlurExprmt_output_images* and the input image is in *blurtest_images* folder. Repetition = number of times blur is repeated per algorithm.

Figure 1: (left) original cropped image; (right) blurred image

Image for Experiments

Choosing the right image is a very delicate balance. A very high-resolution image would result in the 9-pixel neighbouring blurring algorithm being not noticeable as the resolution remains high. While, for relatively low-resolution images, while the blurring has a very pronounced effect, the low resolution creates less accurate testing of concurrent algorithm performance due to increasing the weight of setup time. The image, shown in figure 1, was selected for its sharp edges with a high level of detail, making the blur more easily observable (specifically through: Big Ben tower edge and left clock). Additionally, the original rectangular image was cropped to a square to avoid biasing the row vs column experiments. While a lower resolution could have made the blur more visible, it would make the tests less accurate so this was a balancing act. The image used has the dimensions: 600x600 pixels.

Blurring Algorithms

Every blurring algorithm used the sum of its 8 neighbours + itself / 9 to find its new blurred value.

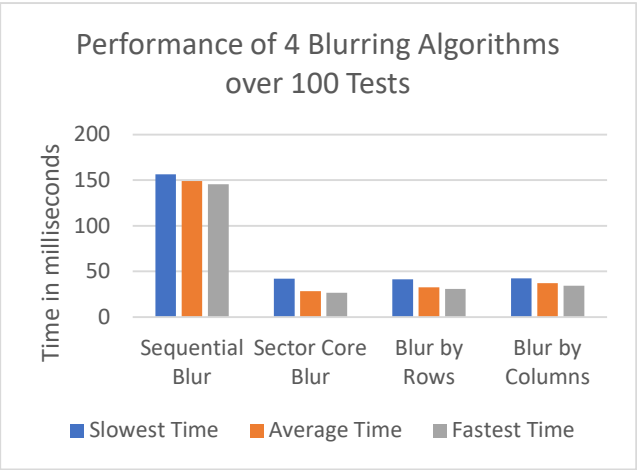
1	**Sequential Blur**	Single process algorithm which sequentially loops through and blurs every pixel. (provided)
2	**Pixel by Pixel Blur**	The main process creates a separate thread for every pixel in the image and each thread/process computes the blur value for 1 pixel. All threads are created and managed by the main process.
3	**Sector Core Blur**	The algorithm creates a thread for every CPU core, by splitting the image into roughly equal vertical sectors corresponding to the core count. A constant is used to set the core number.
4	**Blur by Rows**	Partitions the image into a sequence of rows and assigns each thread to blur 1 row of pixels.
5	**Blur by Columns**	Partitions the image into a sequence of columns and assigns each thread 1 column of pixels to blur.

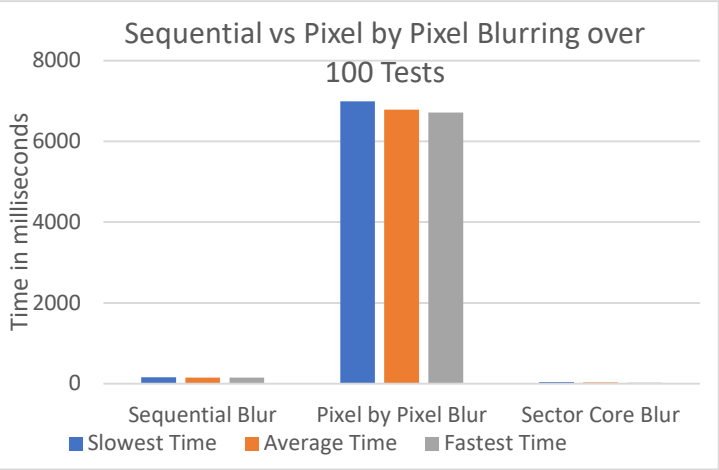
Experiment/Test results

(100 experiments, nearest 2 decimal places for rounding)

N	ALGORITHM NAME	SLOWEST TIME (MS)	AVERAGE TIME (MS)	FASTEST TIME (MS)
1	Sequential Blur	156.45	149.09	145.66
2	Pixel by Pixel Blur	6993.24	6787.57	6715.77
3	Sector Core Blur	42.19	28.50	26.55
4	Blur by Rows	41.21	32.67	30.67
5	Blur by Columns	42.37	36.97	34.17

Graphical Visualisation





These graph emphasises the importance of avoiding the extremes, highlighted by the top 3 algorithms having relatively similar performance in scale. Pixel by pixel is so substantially slower, it required its own graph to not distort the other graph.

Technical Issues (during experiments)

The primary technical challenge was managing zombie threads because pixel by pixel blur created enough threads to hit the machine thread cap and thus require cleanup of the old (zombie) threads to make new threads. My initial assumption was that threads would self-free/memory manage when finished and I was only responsible for freeing the arguments passed in. This led me to overlook the need to join threads thus not letting the thread fully free itself (zombie thread) and thus failing later blur tests. This misunderstanding resulted in 2 code issues: not joining threads to make room for new threads causing hangs and not even storing the pthreads that I was creating for joining. To fix this problem, I implemented a queue so my main process could join a thread to clear it whenever there was a thread limit issue rather than hanging from too many un joinable zombie threads. The queue approach returned the earliest un-joined thread which would be more likely to finish quickly compared to other threads and thus reducing the time spent in thread_join by the main process: holding it up from additional thread creation. Moreover, the queue provided an easy way for my main program to keep track of all the threads and adequately wait for them to finish after all the threads were created. And finally, I began allocating my thread structs to the heap and storing them in my queue rather than losing them as local stack variables.

Findings and Analysis

Throughout my experiments, sector-blur emerged as the fastest, closely followed by row and then column. Then sequential lagged far behind and pixel by pixel was performance degradingly slow. The reason sector-core-blur was the best performing algorithm was the design to match the number of threads to the numbers of CPU cores through roughly equal partitioning. This maximised CPU cores usage due to splitting work amongst the cores with minimal thread scheduling involved. Increasing the number of threads would not increase the number that can be run at any moment since only 1 thread can be ran per core at a time. Thus, row and column were slower (due to scheduling) than sector even though they parallelised more. Row blur was shown to be a performance boost on column blur and I hypothesis that this is due to the way the picture is likely stored, which is with the pixels being sequential in memory. When the program decides on what to cache after a memory access, it is more likely to cache closer to where the last memory access was since future accesses are typically near past accesses. If the picture pixels are sequential, then any neighbour or nearby memory access caches could be cache hits for the row thread. While for a column thread, the gap between its memory accesses is the picture width thus it does not benefit from this neighbouring caching increasing cache hits. With row blur having more cache hits than column blur, it thus avoids more time-consuming memory accesses and is faster as a result.

The lowest performance was pixel by pixel blue and this occurred due to the substantial overhead of creating so many threads. Additionally, the machine 6 cores only allows 6 parallel threads, thus: no parallelisation performance boost compared to sector blur.

While sector -blur was on average noticeably faster than row or column, they were all very similar in their worst cases, column blur is just more consistently close to its worst case. For row blur, its advantage over column blur relies on less cache misses but this can have some variance based on what is in the cache and the caching algorithm resulting in a potential for a worst case where row blur is equivalent to column blur. Similarly for sector blur, it has a worst case where concurrency variance causes unfair scheduling. Sector core has exactly the right number of threads to utilise all cores, so any thread finishing early due to separate thread scheduling conflicts with possibly the main process would result in sector blur not utilising all the cores. So, column blur has more consistency than row or sector because it does not get either of their performance benefits. This blur task in general, is highly parallelisable due to the disjoint nature of the dataset: picture pixels requiring none of the synchronisation which slows down and complicates concurrent solutions. This is why the speed gains compared to sequential are so vast.

Conclusion

Overall, I determined that there is a rough optimal number of threads based on hardware but showed that it was more crucial to avoid the extremes of both sequential and over parallelising. A future exploration could be on the impact of locks on the number of threads since conceivably that would reduce concurrency benefits and add complexity in a way that might change optimal thread numbers.