**PARALLELIZING IMAGE FILTERING TECHNIQUES**

Kevin Gregor – krg43, Jaydev Kshirsagar – jlk368

**ABSTRACT**

The process of image filtering can be inherently parallelized since each operation that happens on any given pixel is completely independent of operations on other pixels. Hence, we have a lot of scope for exploiting the parallel nature of the problem. For large image sizes, applying an image filter serially will take a significant amount of time. We explore multiple methods of performance tuning, including memory layout adjustments and parallelization using OpenMP. We primarily attempt to parallelize the Finite-Impulse Response (FIR) type of filters that operate in the spatial domain. We have implemented a generic framework than can apply any FIR filters with a 3x3 kernel. Configuring the kernel appropriately can help achieve the desired type of effect.

1. **BACKGROUND & RELATED WORK**

There are a variety of different image filtering techniques, but we focus specifically on applying Finite Impulse Response filters through convolution in the spatial domain. This means that we are performing a transformation on the value (i.e. the color) of a pixel, where we change the value to a weighted average of the pixel itself and all the neighboring pixels. In order to do this, we use a 3x3 matrix, called a kernel or mask, to specify the weights of the nine pixels. The center of the matrix denotes the pixel itself and the other eight locations in the matrix are the neighboring eight pixels. Below is an example of the kernel for a Gaussian Blur, a low-pass type filter.

|  |  |  |
| --- | --- | --- |
| 1/16 | 1/8 | 1/16 |
| 1/8 | 1/4 | 1/8 |
| 1/16 | 1/8 | 1/16 |

Fig 1. Gaussian Blur Kernel

Although neither of us had extensive experience in the area of image processing, we did have a preliminary understanding of signal processing concepts. A plethora of work has been done in the image processing field, and we were able to quickly learn about exactly what image filtering is and how it is implemented using various resources online.

We also found a few studies on the speedup resulting from parallelization of image filtering. As mentioned in the abstract, convolution filtering is a great candidate for parallelization because all of the pixel computations are independent of each other. In one study we found, which was done by students at Purdue University, the students tested parallelization using both OpenMP and MPI. They found that increasing the number of processes using OpenMP led to more of a speedup than doing so with MPI. Below is a graph of their findings:

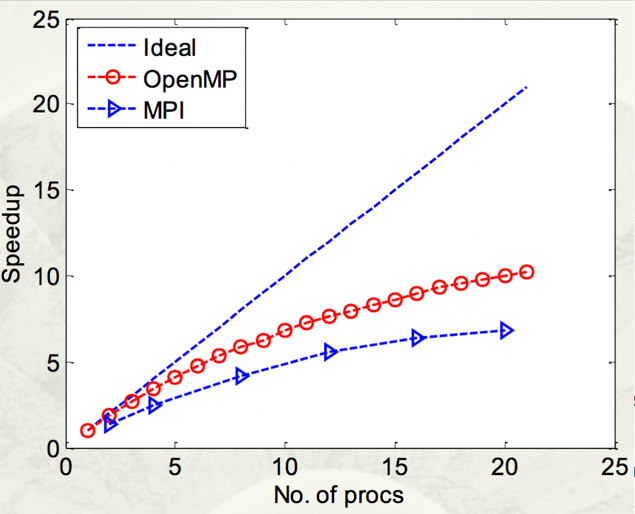


Fig 2. Speedup of OpenMP and MPI [1]

Another study by Devrim Akgün of Duzce University in Duzce, Turkey, also got similar results for the impact of number of threads/processes on speed up for image processing. From his graph below, he got a speedup of 5-6 times using six threads – similar to the Purdue students’ speed up of just under 5 times using six processes.

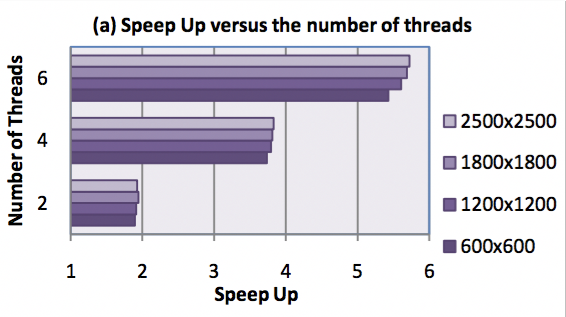


Fig 3. Speed up vs Number of threads [2]

As demonstrated by these studies, there is a fair amount of speedup resulting from simple parallelization, which we aimed to replicate. We were not able to find information regarding the impact of memory layout adjustments on speed up, so we were unsure of exactly how our changes would impact performance.

1. **IMPLEMENTATION DETAILS**

The core implementation of our project, which includes everything except for reading the images and running through test cases, was written in C++. We created a shared C++ library, which is then interfaced with a Python script. Our Python testing script reads an image into a NumPy matrix and creates a C++ object for the image based on the matrix, its dimensions, the image format, and the desired memory layout format. Then, after the filter is applied using C++, the processed image matrix is passed back into the Python script and written to the output image file. We chose to abstract the image reading and writing to a Python script because implementing it in Python is quick and easy due to the availability of libraries and also because this was not a performance critical part of the project. This also allowed us to run through test cases quickly using the Python script.

The C++ code when invoked through Python, creates an object for the image that stores all essential attributes of the image,  
and at the same time allocates required amount of memory for storing the pre-processed and the processed image data. Also the input pixel data in one of the supported image formats, is read and the pixel values are normalized to [0, 1], and the pixels are stored in the specified memory layout.

When the Python script calls in to the library for getting the image processed, the specified filter kernel is convolved with the image data and the resultant image data is stored in the memory area allocated for the processed image. The pixel values of the processed image data are still in the normalized form and in the specified layout.

When the Python script requests back the processed image data, the library code converts the pixel values back to the desired format and also converts the image layout back to the natural one; the converted image data is written to a pointer supplied by the Python script. The Python script subsequently writes the image data to a file.

The supported image formats or pixel data formats are 8-, 16-, and 32-bit grayscale images, 24-bit RGB images with byte-alignment of 3 and 4 for the pixels. The image format has to be specified when using this application.

The parallelization of the computation involved in the processing, is not very difficult and also the performance is not limited by compute, but by memory bandwidth. Hence, it was more important to use optimal memory layouts for the images so as to maximize the cache efficiency. We implemented two special memory layouts that would be suitable for the spatial locality of the pixel references that are involved in the filtering computation. We also implement a naïve row-major layout for baselining purposes. This implementation is not ideal for applying 3x3 matrices, as there is not a strong sense of data locality – only three of the nine pixels needed for each computation are in the same row as the pixel we are doing the computation on. Clearly, this can be improved upon by using another memory layout.

The first special layout that we implemented (which is actually having 3 variants) is the Block-Linear layout with block sizes of 8, 16, and 32. This involves dividing the image into blocks of fixed sizes (*n*-by-*n* (8/16/32)). The blocks stored in row-major order, and the pixels within each block are also stored in row-major order. Therefore, the data locality can be greatly increased, as each pixel needed for computation will be no more than indices away from the pixel being modified. For image sizes that are not perfect multiples of the block size, we add extra “dummy” pixels to fill up the remaining rows and/or columns in order to create a matrix that can be blocked perfectly. These dummy pixels are not touched when actually applying the filter and they are not saved as a part of the image after the filter has been applied.

The second special memory layout adjustment we implemented was a twiddled Z-Morton style layout. Our Z-Morton layout can be seen below in the figure below.

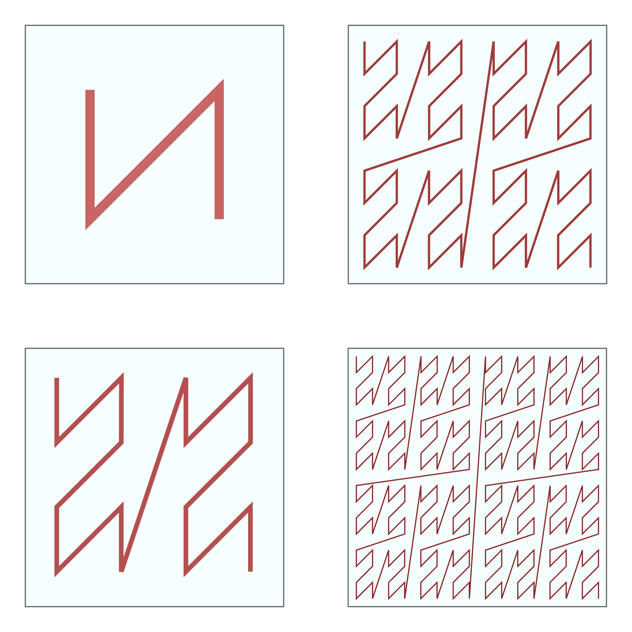


Fig. 4. Z-Morton layout visualization

We divide the input matrix into four large blocks – top left, bottom left, top right, bottom right – that are stored in memory in that order. We recursively divide the blocks into smaller and smaller blocks, applying the same ordering technique, as seen in the Figure 4. In theory, this gives us even better data locality than the Block-Linear layout, due to pixels in any given 3x3 grid being stored very close together in memory. Moreover this layout is better suited for hierarchical caches, since the data corresponding to the higher level of the recursion tree lies in the next higher level of cache. Just like with the Block-Linear layout, this layout also suffers when image dimensions are not perfectly aligned. However, the Z-Morton layout needs to add dummy pixels when the image size is not a power of two, and dummy pixels are added to round up to the next power of two.

Finally, we implemented another layout, which we refer to as a “ghost cell” layout with 5x5 blocks. The basic idea is that we want to apply the convolution to a whole 3x3 block at once, and we need all the neighboring pixels in a 5x5 block in order to do that. However, some pixels will rely on the same pixel even though they are in different blocks, so we overlapped the blocks and added ghost cells to solve this problem. Thus, many pixels are actually stored in memory twice, which means the initial conversion to and from this layout has a larger cost to it. Essentially, the sliding window of 5 pixels for the blocks only moves by 3 pixels when going to the next block, so two rows (or columns) of pixels overlap with the previous block. This means that when we do a computation on the middle nine pixels of a block, we are never repeating pixel computations and we end up doing computations on all the pixels.

For the actual computation, we created a matrix addition problem (perhaps similar to the matrix multiplication project). We found that if we apply the weights of the kernel to all possible 3x3 matrices within the 5x5 block and add them all up, we would get a 3x3 matrix with all the correct pixel values for the middle nine pixels of the block. This is visualized in figure 5 below.

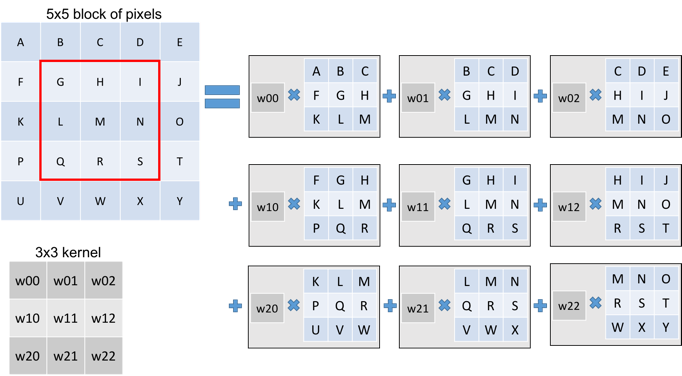


Fig 5. “Ghost Cells” computation

This should improve performance because we are effectively doing nine pixel computations at once by adding all the matrices up. However, there is a cost to creating all nine matrices, so we do not expect an extremely large speedup. In addition, just like in the block linear and Z-morton layouts, we add dummy pixels if

All of our non-naïve memory layouts also have a hidden cost of converting the input matrix into the desired layout before applying a filter and back afterwards. Since FIR filters can already be applied very quickly on an image of any reasonable size, converting to and from these layouts just to apply one filter proves not to be worth it: the total time from reading the input image to writing the output image is longer than the naïve version. However, we hope that the speedup in the actual filter application means that using these filters would be better when applying multiple filters in succession.

1. **PARALLELIZATION**

We planned to use OpenMP to have the processing done in parallel. The first approach we tried was to parallelize the loops in our convolution code that iterated over the pixels of the image. We didn’t observe the performance benefitting much from this. We believe the probable reason is that the number of pixels to be processed is far more in comparison to the available number of processor. Hence the OpenMP runtime would experience a significant overhead in delegating the huge workload among the available processors.

Thus we thought of an alternative strategy, wherein the division of workload was done in our code itself, and we used OpenMP to parallelize the for loop going over our divisions, or “chunks”. For such a division, the image was split up into 16 rectangular chunks we used:

#pragma omp parallel for collapse(2) above the nested for loops to parallelize the loop over the chunks.

For the ghost cells layout, we wrote the loop in a column-major way, so all the blocks in the first column of blocks are computed, then the next column and so on. This made it very easy to parallelize the process with OpenMP using the same line of code as above. This made it so our parallel models were very comparable to each other.

1. **PERFORMANCE ANALYSIS**

For performance analysis, all three memory layouts for the input image are tested in succession. Also all three types of filters are applied.

* 1. **PEAK PERFORMANCE**

Since the computation that is being parallelized here is pleasantly parallel, theoretically the amount of speedup will be equivalent to the number of threads running in parallel. I will put in the theoretical peak perf figures here. However, processing every single pixel involves 9 memory fetches. For large sized images, cache misses would be unavoidable and memory bandwidth becomes a bottleneck.

* 1. **TEST CASES**

Since convolution filters can be applied extremely fast on any common image sizes, it was important to establish concrete test cases with enough samples to generate meaningful results. We decided to test each memory layout 25 times for each filter, for a total of 75 samples per memory layout (for each value of the dependent variable). We then averaged all of these in order to get a good value for how long it takes to apply a convolution filter for each memory layout. Using this testing method, we were able to get consistent results across multiple runs.

* 1. **SERIAL PERFORMANCE**

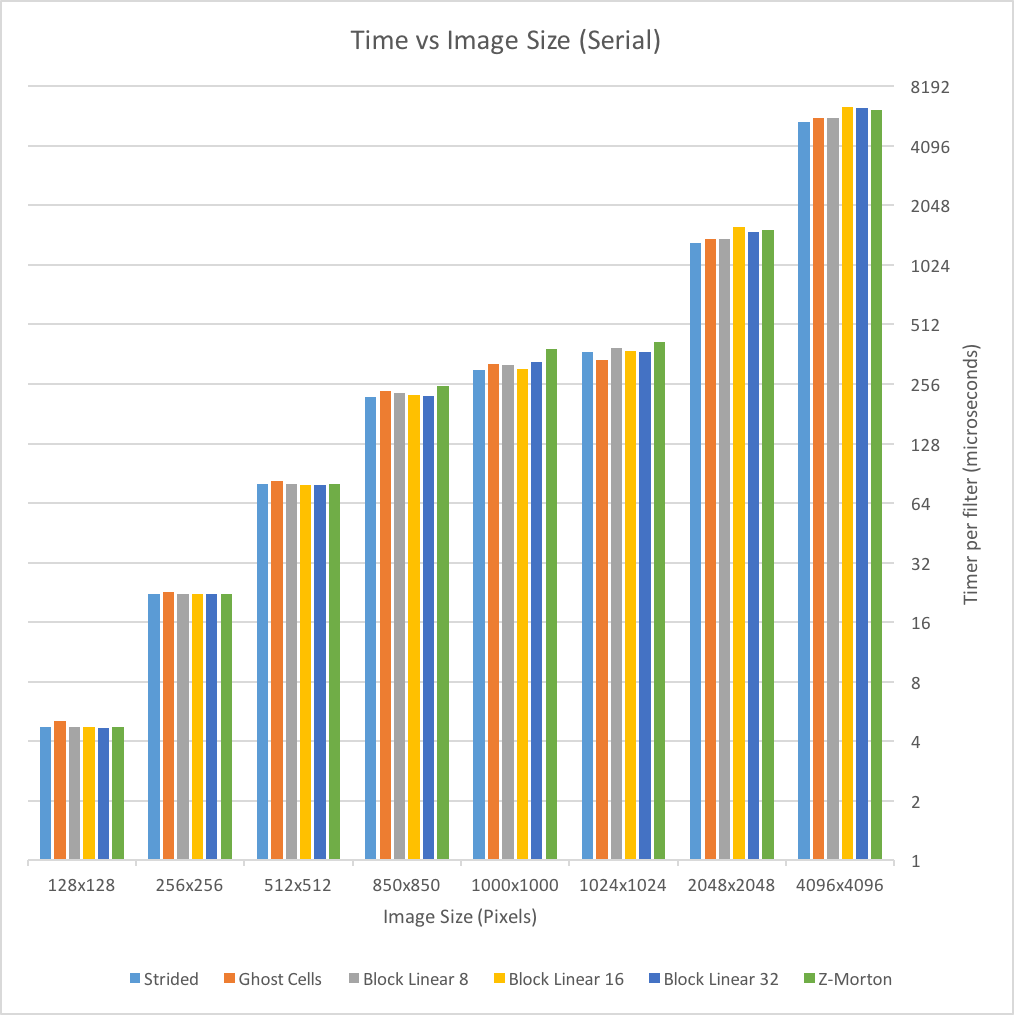


Fig 6. RGB Serial Graph

Our first experiment involved testing the time per filter, measured in microseconds, of the six memory layouts across eight different image sizes. This is a weak scaling experiment since we are varying the amount of work needed to be done and measuring the time. We sought to compare not only the performance of the different memory layouts as size increased but also the performance of RGB versus grayscale images. Figure 6 details the results of RGB images, and Figure 7 shows the results of grayscale images. As you can see, the vertical axis is measured in microseconds (a ten-thousandth of a second) and scaled logarithmically by a power of 2. The time to apply a filter increases exponentially as image width/height increases, which is not very surprising. The exponential increase is actually a linear increase based on the number of total pixels in the image. For example, a 2048x2048 image has only a quarter of the pixels in a 4096x4096 image, and the time per filter in the 2048x2048 image (1300-1500microsec) is just about a quarter of the time per filter in the 4096x4096 image (5300-6300microsec).

Another aspect of this graph is that the strided row-major layout performs better than expected. For most of the image sizes, it is either equivalent to or faster than the other memory layouts. For smaller image sizes, the block linear and Z-Morton layouts perform very well, but they begin to perform worse as image size increases. The ghost cell layout seems to have the opposite effect, starting out slower and speeding up as image size increases.

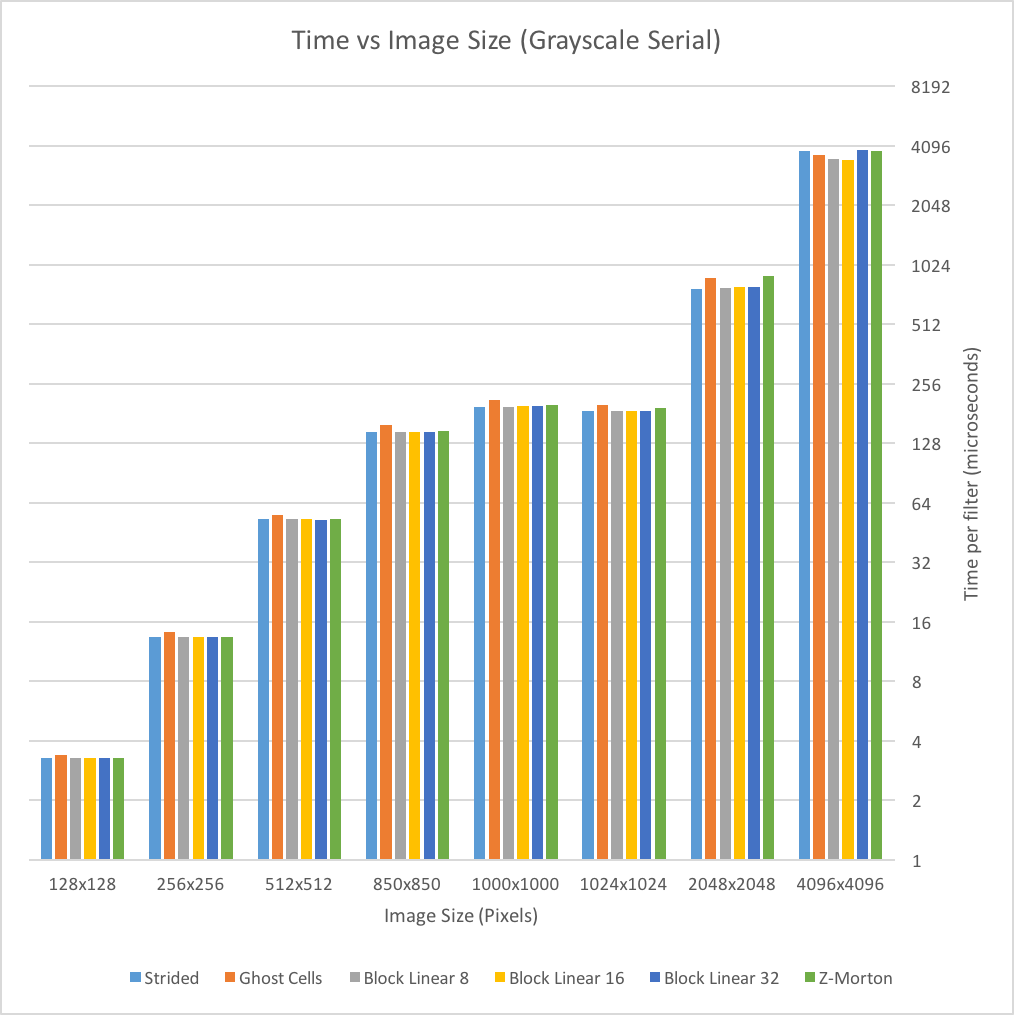


Fig 7. Grayscale Serial Graph

Perhaps the more surprising takeaway from this experiment is the time per filter of the grayscale versus RGB images. Since the RGB images have three times more values per pixel than their grayscale counterparts, one could possibly expect to have around a three times speedup when applying filters to the grayscale images. However, this is not the case. There is some speedup in the grayscale graph, but the filters are applied less than twice as fast as in the RGB case.

The different memory layouts perform mostly the same across the different image sizes, with a few exceptions. The ghost cell layout performs worse than all the others until the 4096x4096 image, while the block linear layouts are consistently the best layouts.

* 1. **PARALLEL PERFORMANCE**

In our next experiment, we picked a standard image size – 1024x1024 pixels – to do a strong scaling study with. We varied the number of threads from 1 to 24 and graphed the results of the various memory layouts. We chose this image size because it is one of the larger ones, but not too large. It is a good middle ground that we thought would give us clear results. In figures 8 and 9, we show the results of this experiment.

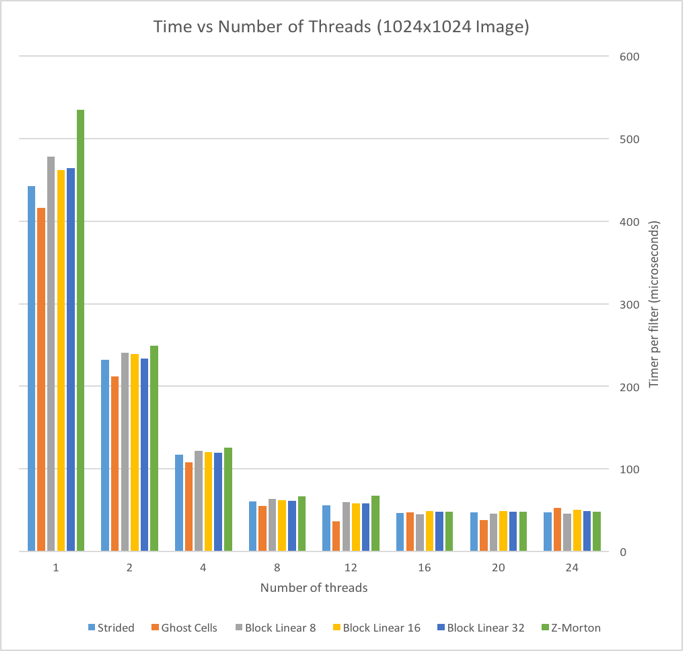


Fig 8. Time vs Number of Threads

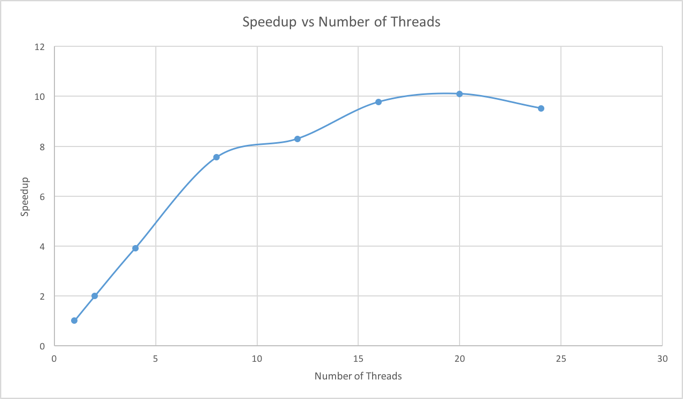


Fig 9. Speedup vs Number of Threads

Figure 8 shows the time per filter (still in microseconds) of the memory layouts using a varying number of threads. In order to make more sense of this data, we created figure 9, which averages the time of the six filters at each thread count and shows the speedup for the different number of threads. Using this, we found that 20 threads seemed like the ideal number of threads to test on all image sizes. The speedup from using 20 threads was just over 10 times (10.098x). We then conducted the same experiment in Section 4.3 for RGB images, except we ran the parallel code with 20 threads.

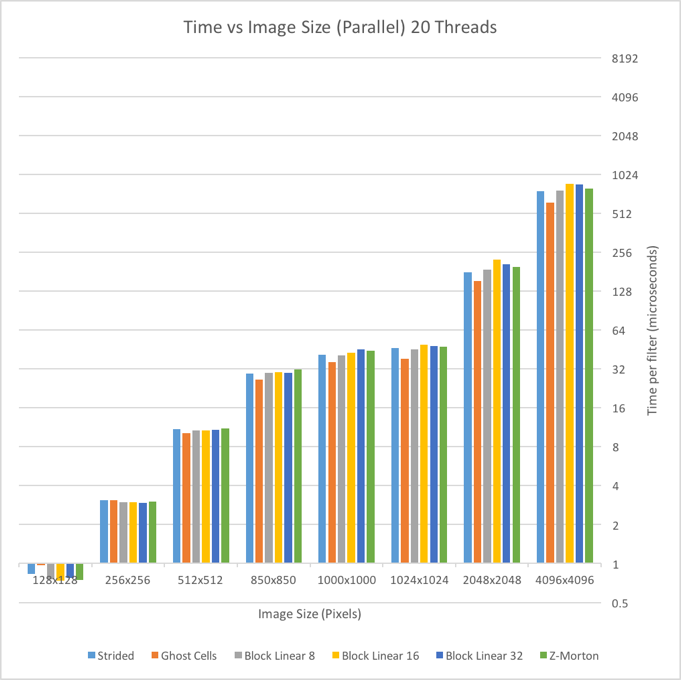


Fig 10. RGB Parallel Graph

As figure 10 shows, the speedup of parallelization with 20 threads is very significant, especially at larger image sizes. The most interesting part of this graph is that the ghost cell layout consistently performs better than all the other memory layouts, and the gap between it and the next fastest layout increases as image size increases. All the other layouts vary in relative speed across image sizes, but the ghost cell layout gets relatively better and better for larger image sizes.

1. **CONCLUSION**

Our entire study came with some surprises and some expected results. First, our speedup from parallelization was almost exactly as expected. Our graph for speedup and the the speedup graphs from other previous studies in Section 1 are nearly identical. As the number of threads increased, we saw a speedup of similar shape in figures 1, 2 and 9.

However, the memory layout adjustments that we made did not improve performance as much as we had hoped. While there was a speedup from the layouts in some cases, and our overall fastest model was the ghost cell layout with 20 threads, the ghost cell layout was actually slower in some serial cases (and the other layouts were slower in some serial and parallel cases!). It is clear that the ghost cell layout benefits the most from parallelization and larger image sizes. We expected the special layouts to improve for larger image sizes because the strided row-major layout should only worsen as rows get longer. Perhaps the reason that the ghost cell layout improved the most from parallelization is that any 5x5 block can be picked up by any thread and there are no subdivisions – the blocks are simply in column-major order. Overall the speedup of our best model (ghost cells with 20 threads) over the serial row-major implementation is 11.5 times – a rather significant increase in speed.

While we are not entirely sure why our other memory layouts did not show quite the same speedup over the row-major implementation, we do have some theories. While there are more cache hits and better data locality for the other layouts, there are a lot more computations that must be done to know where a given pixel is in memory. Therefore, these computations seem to hurt the performance more than the data locality improves it. This is evidenced by the performance at larger image sizes: there are more pixels, meaning more computations, and the performance of the block linear and Z-Morton layouts only gets worse. In addition, the time required to apply a filter is extremely short, so any extra computation or even variable assignment could affect the performance.

The performance of grayscale versus RGB images is also interesting but not all that surprising. While there are three channels per pixel, storing a pixel’s data in memory as R\_value, G\_value, B\_value means that doing computations on all three channels is fairly simple. Thus, it does not require three times as much time to compute.

There are a few different extensions that would make for interesting future work. One such extension is testing larger image sizes. Larger images take exponentially longer to run, but we wonder if the trend of the parallel ghost cell layout getting “better and better” would continue as image sizes got even bigger. Another extension would be other layouts, like different block sizes for a block linear layout. It is clear that the block linear performance dips for larger image sizes, but it is possible that larger block sizes would remedy that.

**REFERENCES**

[1] Parallel Image Processing – Lei Cao and Yan Wang (2013-04-19)

https://engineering.purdue.edu/~eigenman/ECE563/ProjectPresentations/ParallelImageProcessingCao\_YanWang.pdf

[2] Performance Evaluations for Parallel Image Filter on Multi-Core Computer using Java Threads – Devrim Akgün (July 2013)

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.1706&rep=rep1&type=pdf