Investigating Parallelization Techniques For Photo Editing

Alice Gusev
Email: agusev@usc.edu

Victor Hui
Email: victoryi@usc.edu

Joshua Williams
Email: joshuaw1@usc.edu

Abstract— Editing a photo is a computationally-intensive task that requires the formatting of large magnitudes of pixels, even for the simplest of changes. Although previous and current works are in progress, we study the efficacy of several parallelizing paradigms and techniques to find the fastest method. Through testing, we find CUDA Unrolling produces the best result for our implementation.

1. Introduction

Nowadays, photo editors can be seen integrated in many of our smartphone applications, from simple filter selection on social media applications such as Facebook, Instagram, and Snapchat, to professional photo-editing tools such as Adobe Photoshop. Many users on these platforms utilize these photo-editors to modify their image's appearance to achieve a specific look, making them more visually appealing or for a specific application. Instagram is one of the most popular platforms where photos are edited and shared, having more than 1 billion monthly users [1].

A common and simple photo editing tool is the filter. A filter is an automated photo editing tool where the colors of a photo are changed in a way that is defined by the filter. Photo filters are an easy way for amateur or hobbyist photographers to quickly edit their photos to make them more appealing. To demonstrate the popularity of photo editing and filters, a marketing firm analyzed a random sample of 40 million Instagram posts and found that approximately 18% of the posts use in-app filters [2]. Photo filters require processing of each individual pixel, so the popularity of photo filters requires better image processing technology.

As camera technology advances, the digital image file size produced increases as the number of pixels stored is increasing. Figure 1 shows that the number of megapixels of smartphone cameras has been increasing roughly linearly for the past decade. The number of megapixels of a digital image represents the total number of pixels that the image contains. Because of this, as megapixels increase, the pixels that a photo filter needs to operate on increases.

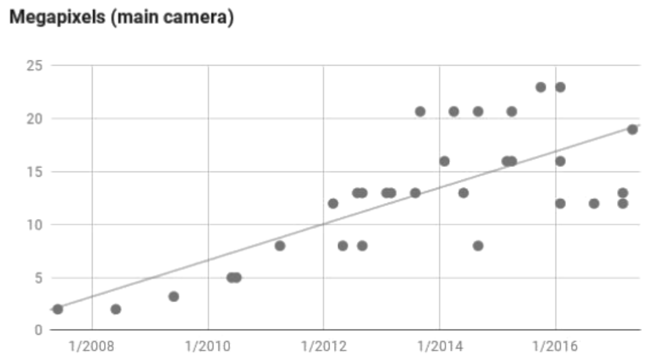


Figure 1. Yearly Trend Of Smartphone Camera Quality [3]

When a user utilizes a photo editor, they want filters to be applied nearly instantaneously, so increasing the speed that a filter can be applied contributes directly to user satisfaction and can potentially become a deciding factor on their decision to continue using an application. As images get larger, data parallelism can be used to improve filter application speed. Therefore, our group aims to parallelize image processing, especially regarding filter applications, to make this process more efficient. By parallel processing, we can apply the same filter on multiple parts of the image concurrently, therefore potentially cutting the execution time by a factor of the number of threads utilized for this application.

2. Background & Motivation

A digital photo is a file that can be broken down into a 2D array of pixels. Each pixel is represented by three values: red, green, and blue, which is known as

RGB. Each RGB element can have an integer value ranging from 0 - 255. The combination of the RGB values of the pixel creates the color at a particular location. A photo editing application applies a filter by modifying the RGB value of every pixel in an image file according to a predefined mathematical formula or algorithm.

Although the process of applying a filter is simple, there is a lot of mathematical computation that needs to occur. As the number of megapixels increases, the number of mathematical computations increases as well. Images can reach a resolution of 7680 x 4320 = 33,177,600 pixels, so applying an image filter sequentially would require millions of executions of the same mathematical formula. The calculation may be simple, but the sheer number of calculations necessary makes the computation extremely expensive. Because of this, utilizing data parallelism is a simple way to provide substantial execution speedup.

3. Proposed Ideas

As mentioned in the technical background, editing a photo with a filter requires a lot of computation especially for high resolution images, and parallelizing these computations would greatly increase the execution time. Therefore, our project goal is to develop an application to filter images and find a method that best decreases execution time (measured in seconds) of the filters in our application. Using what we have learned in the course, our method will include testing combinations of different programming paradigms and parallelization implementations on various input images with three types of filters: Mirror, GrayScale, and Color Inversion.
Overall, we will have several combinations for interpretation: our test programming paradigms will include CUDA and OpenMP. For OpenMP, we investigate the amount of threads for parallelization. For CUDA, we use a regular implementation, as well as loop Unrolling for different factors.

3.1. OpenMP

We choose OpenMP as a proposed paradigm because it allows for a relatively easy implementation that gives quick results. It is an API that supports shared-memory parallel programming in C/C++. OpenMP uses the Fork-Join Model with parallelization, where Fork: the master thread first creates a team of parallel threads, and Join: when the team threads complete their parallel region, they synchronize and terminate, once again leaving only the master thread. The OpenMP library creates an easy method for loop-level parallelism, which we will utilize for filters during the implementation phase of our project. Despite its simple implementation however, OpenMP lacks when it comes to programmer control. This may impact our result negatively.

3.2. CUDA

Our second paradigm for investigation is CUDA, a general-purpose parallel computing platform and programming model that leverages NVIDIA GPUs. A typical CUDA program consists of two parts, the host code that runs on the CPU, and the device code that runs on the GPU. At compile time, CUDA's NVCC compiler separates the device code from the host code. CUDA programming is well-suited to address data-parallel computations, where many data items are operated on at the same time. For this reason, we hope that CUDA's suitability with data parallelism will provide sufficient speed-up results, given that photo-editing is a staple data parallelism task.

4. Implementation

Following our Proposed Ideas, we have several implementations for testing on each of the filters. We also use four different image sizes to find the variability of timing for each method. Firstly, we have our baseline model that performs each filter serially. Second, we test OpenMP on different threads counts ranging from 1 to 10. Finally, we test CUDA on each of the filters, as well as CUDA Unrolling on each of them.

Therefore, we have 8 implementations for each of the four images:

- (1) Serial Test per Filter
- (5) OpenMP Tests (for threads of 2, 4, 8, and 16) per filter.
- (2) CUDA Tests (regular and unrolled) per filter. There are 4 different unrolling values (1, 4, 16, and 64).

This results in a total of 24 implementations.

4.1. Filters

For our testing of OpenMP, CUDA, and CUDA Unrolling, we chose three filter methods: GrayScaling, Mirroring, and Inverting. We specifically chose these filters because they vary greatly in their implementations, but still follow data parallelism. This way, we can see which of our paradigms and implementations are universally applicable to a greater variety of filters.

4.1.1. GrayScaling

There are many techniques for GrayScaling an image, such as taking an average of the individual R, G, B values of a pixel, RGB to YUV value conversion, HSV to GrayScale value conversion, and RGB Binarization. While all methods have its benefits and drawbacks, the method we used is known as the Luminosity Method. Studies show that the Luminosity method appears most pleasing to the human eye, as it takes into account the inconsistencies of the human retina when observing colors from different light spectrums. [4]

$$Gray = (R * 0.3 + G * 0.59 + B * 0.11)$$

4.1.2. Mirroring

In recent years, mirroring filters has been used extensively in the field of Computer Vision research. As a means to effectively double the size of an input dataset, images in a dataset could be mirrored before inputted into a Deep Neural Network for image classification, this method has been proven to improve classification rate significantly in certain cases. However, extra care must be taken when mirroring orientation-critical images such as ‘d’ and ‘b’ for text classification.

The mirroring filter is done by replacing the pixel at the current location with the pixel at (x_size - currloc) of the image.

4.1.3. Inverting

Color inversion is also known as turning an image negative, important for people with sight disabilities, color inversion preserves the relative contrast between different elements inside an image. In more recent years, color inversion also began appearing in our digital devices as Reading Mode, providing us with white text on black background which can be more comforting for our eye.

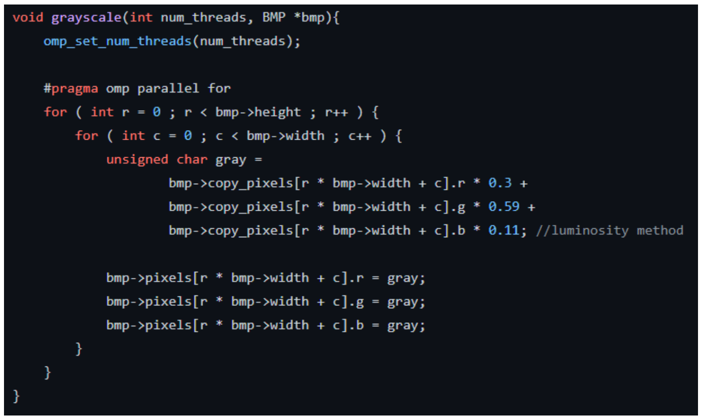
In order to turn an image negative, we would need to perform a subtraction from the max value for each individual RGB value:

$$New\_R = 255 - R \qquad New\_G = 255 - G \qquad New\_B = 255 - B$$

4.2. Programming Paradigms

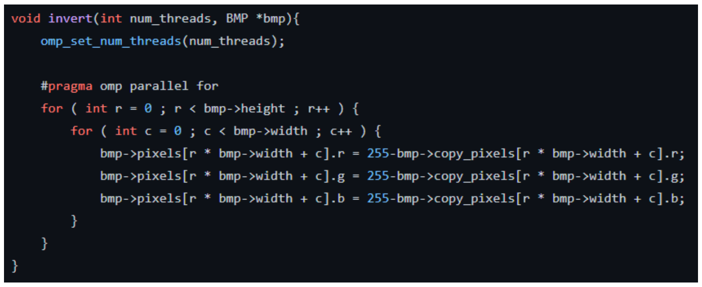
For all methods and measurements, we parallelized the computation of pixel values only, meaning that all read, write, and header decoding operations were performed in the main thread of the host function. Because CUDA programs cannot directly write the output into the same files that the host program is using, it can perform the mathematical formulas on the pixels but it cannot save those pixels into the file along with the necessary file headers. Although sequential and OpenMP would be able to parallelize the file output as well as the pixel manipulation, our parallel implementation only included the pixel modifications in order to allow direct comparisons between the different programming paradigms. For modularity, our implementation compartmentalized the many steps needed to read a .bmp image into different functions, which can be called sequentially before executing our filter functions. [5][6][7]

4.2.1 OpenMP



```
void grayscale(int num_threads, BMP *bmp){
    omp_set_num_threads(num_threads);

    #pragma omp parallel for
    for ( int r = 0 ; r < bmp->height ; r++ ) {
        for ( int c = 0 ; c < bmp->width ; c++ ) {
            unsigned char gray =
                    bmp->copy_pixels[r * bmp->width + c].r * 0.3 +
                    bmp->copy_pixels[r * bmp->width + c].g * 0.59 +
                    bmp->copy_pixels[r * bmp->width + c].b * 0.11; //luminosity method

            bmp->pixels[r * bmp->width + c].r = gray;
            bmp->pixels[r * bmp->width + c].g = gray;
            bmp->pixels[r * bmp->width + c].b = gray;
        }
    }
}
```

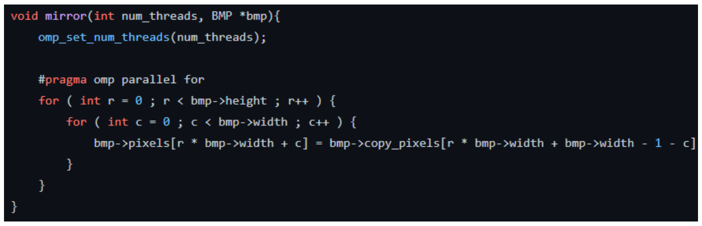
Figure 2. OpenMP Grayscale Implementation [4]



```
void invert(int num_threads, BMP *bmp){
    omp_set_num_threads(num_threads);

    #pragma omp parallel for
    for ( int r = 0 ; r < bmp->height ; r++ ) {
        for ( int c = 0 ; c < bmp->width ; c++ ) {
            bmp->pixels[r * bmp->width + c].r = 255-bmp->copy_pixels[r * bmp->width + c].r;
            bmp->pixels[r * bmp->width + c].g = 255-bmp->copy_pixels[r * bmp->width + c].g;
            bmp->pixels[r * bmp->width + c].b = 255-bmp->copy_pixels[r * bmp->width + c].b;
        }
    }
}
```

Figure 3. OpenMP Inverting Implementation

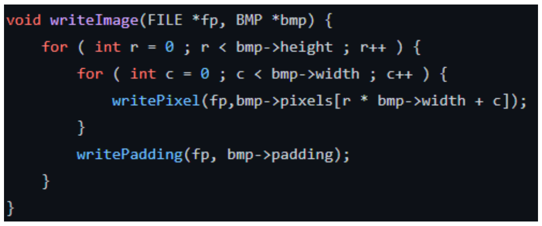
Our implementation for grayscaling and inverting is similar. Outside of the function, we created a Bitmap Class BMP, and read in the pixel values as a 2D pixel array into the Bitmap object. We then pass the bitmap object into the separate functions, and use the equations mentioned above in our implementation section to alter the individual RGB values.



```
void mirror(int num_threads, BMP *bmp){
    omp_set_num_threads(num_threads);

    #pragma omp parallel for
    for ( int r = 0 ; r < bmp->height ; r++ ) {
        for ( int c = 0 ; c < bmp->width ; c++ ) {
            bmp->pixels[r * bmp->width + c] = bmp->copy_pixels[r * bmp->width + bmp->width - 1 - c]
        }
    }
}
```

Figure 4. OpenMP Mirror Implementation

Our implementation for mirroring replaces the pixel of the current pixel array with the pixel at the opposite location as explained in the methods section.



```
void writeImage(FILE *fp, BMP *bmp) {
    for ( int r = 0 ; r < bmp->height ; r++ ) {
        for ( int c = 0 ; c < bmp->width ; c++ ) {
            writePixel(fp,bmp->pixels[r * bmp->width + c]);
        }
        writePadding(fp, bmp->padding);
    }
}
```

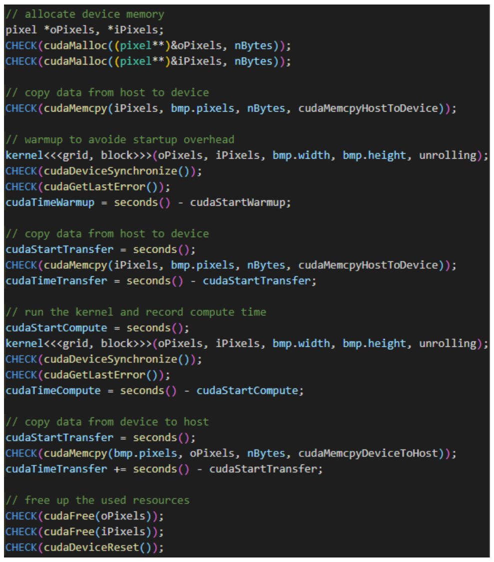
Figure 5. OpenMP Image Writing [5][6][7]

After returning from the function, we performed a sequential write operation to place the now altered pixel values into a newly created bitmap file.

4.2.2. CUDA

The CUDA implementation follows the same file setup steps as the OpenMP implementation. The file is opened, decoded, and parsed for pixel information. Once the pixels have been parsed into an array, it is ready to be used by the CUDA kernel. The most efficient division of blocks and grids was found to be when blocks were always of the size 32x32. Because the pixel manipulation requires only a small amount of data, each thread only requires a tiny amount of resources, so the improved parallelism within a block was found to outweigh the division of SM resources among the block.

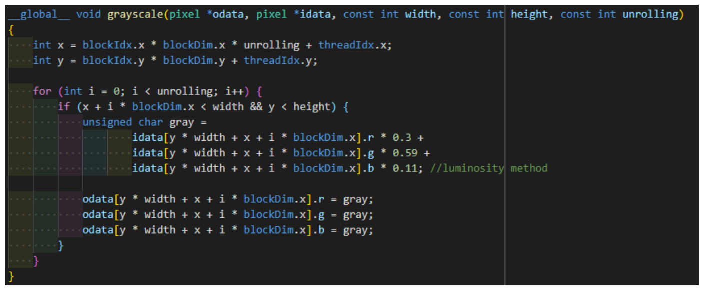
The CUDA portion of the implementation is split into three parts: warmup, data transfer, and computation. The warmup is the time that it takes for the GPU to be allocated and assigned to the host program and to run the kernel for the first time. Since the host program is being used on the Discovery Cluster, a GPU needs to be allocated to it, a process that requires time and can be seen to dominate the runtime of our algorithm. Because this is time that is not applicable to a normal device where the GPU would already be allocated, this warmup time can be ignored during comparisons of runtime. The data transfer part of the algorithm is the time that it takes to transfer the input pixel data to the GPU and for the output pixel data to be transferred back from the GPU. Since the GPU and host have separate memories, the data transfer process must occur and requires additional execution time that cannot be ignored since other implementations do not require this data transfer time. Finally, the computation time is the time that it takes for the kernel to fully run and execute the filter modifications on the pixels. The pixels are stored with three values in a 1-D array, and each kernel thread accesses the appropriate pixels and modifies them for the output. The CUDA host implementation can be seen in Figure 6.

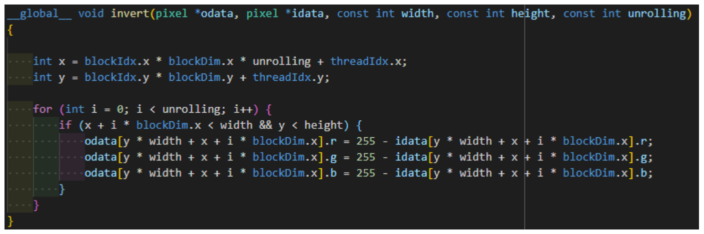


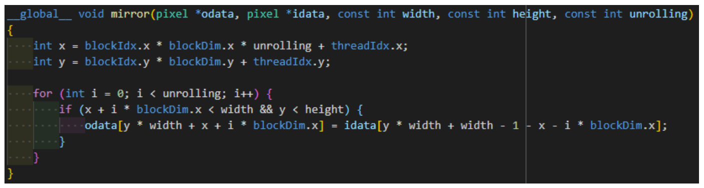
```
// allocate device memory
pixel *oPixels, *iPixels;
CHECK(cudaMalloc((pixel**)&oPixels, nBytes));
CHECK(cudaMalloc((pixel**)&iPixels, nBytes));

// copy data from host to device
CHECK(cudaMemcpy(iPixels, bmp.pixels, nBytes, cudaMemcpyHostToDevice));

// warmup to avoide startup overhead
kernel<<<grid, block>>>(oPixels, iPixels, bmp.width, bmp.height, unrolling);
CHECK(cudaDeviceSynchronize());
CHECK(cudaGetLastError());
cudaTimeWarmup = seconds() - cudaStartWarmup;

// copy data from host to device
cudaStartTransfer = seconds();
CHECK(cudaMemcpy(iPixels, bmp.pixels, nBytes, cudaMemcpyHostToDevice));
cudaTimeTransfer = seconds() - cudaStartTransfer;

// run the kernel and record compute time
cudaStartCompute = seconds();
kernel<<<grid, block>>>(oPixels, iPixels, bmp.width, bmp.height, unrolling);
CHECK(cudaDeviceSynchronize());
CHECK(cudaGetLastError());
cudaTimeCompute = seconds() - cudaStartCompute;

// copy data from device to host
cudaStartTransfer = seconds();
CHECK(cudaMemcpy(bmp.pixels, oPixels, nBytes, cudaMemcpyDeviceToHost));
cudaTimeTransfer += seconds() - cudaStartTransfer;

// free up the used resources
CHECK(cudaFree(oPixels));
CHECK(cudaFree(iPixels));
CHECK(cudaDeviceReset());
```

Figure 6. CUDA Host Implementation

Three device kernels were implemented to execute the three types of filters as discussed earlier. The three kernels can be seen in Figures 7, 8, and 9. By default, each kernel is assigned one pixel to perform the filter on. However, the user can input a value to specify how many pixels each thread will handle. When a thread handles more than one pixel, it implements a technique known as unrolling. Unrolling attempts to mitigate the overhead of additional threads by having each thread do more. There is a tradeoff between the parallelism achieved by more threads and the overhead required to use those additional threads. Unrolling attempts to find the most efficient division of pixels into threads so that parallelism is utilized but overhead is minimized.


```
__global__ void grayscale(pixel *odata, pixel *idata, const int width, const int height, const int unrolling)
{
    int x = blockIdx.x * blockDim.x * unrolling + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    for (int i = 0; i < unrolling; i++) {
        if (x + i * blockDim.x < width && y < height) {
            unsigned char gray =
                    idata[y * width + x + i * blockDim.x].r * 0.3 +
                    idata[y * width + x + i * blockDim.x].g * 0.59 +
                    idata[y * width + x + i * blockDim.x].b * 0.11; //luminosity method

            odata[y * width + x + i * blockDim.x].r = gray;
            odata[y * width + x + i * blockDim.x].g = gray;
            odata[y * width + x + i * blockDim.x].b = gray;
        }
    }
}
```
Figure 7. CUDA Grayscale Filter Implementation [4]


```
__global__ void invert(pixel *odata, pixel *idata, const int width, const int height, const int unrolling)
{

    int x = blockIdx.x * blockDim.x * unrolling + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    for (int i = 0; i < unrolling; i++) {
        if (x + i * blockDim.x < width && y < height) {
            odata[y * width + x + i * blockDim.x].r = 255 - idata[y * width + x + i * blockDim.x].r;
            odata[y * width + x + i * blockDim.x].g = 255 - idata[y * width + x + i * blockDim.x].g;
            odata[y * width + x + i * blockDim.x].b = 255 - idata[y * width + x + i * blockDim.x].b;
        }
    }
}
```
Figure 8. CUDA Inversion Filter Implementation


```
__global__ void mirror(pixel *odata, pixel *idata, const int width, const int height, const int unrolling)
{
    int x = blockIdx.x * blockDim.x * unrolling + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    for (int i = 0; i < unrolling; i++) {
        if (x + i * blockDim.x < width && y < height) {
            odata[y * width + x + i * blockDim.x] = idata[y * width + width - 1 - x - i * blockDim.x];
        }
    }
}
```
Figure 9. CUDA Mirroring Filter Implementation

5. Results

In all methods, we see a clear performance benefit moving from a sequential model to a parallel computation model. However, the speed up as well as optimal number of threads or amount of unrolling differs is dependent on the resolution of the input image.

5.1. OpenMP

OpenMP	Execution time with **grayscaling** filter in seconds				
Resolution	1 thread	2 threads	4 threads	8 threads	16 threads
640 x 426	0.001156	0.000597	0.000477	0.000392	0.025649
1280 x 853	0.004195	0.002408	0.001776	0.001387	0.026169
1920 x 1280	0.007657	0.004416	0.003055	0.002164	0.026957
5184 x 3456	0.04825	0.039712	0.01419	0.009992	0.031806

Table 1. OpenMP Results on Grayscale Filter

OpenMP	Execution time with **Inverting** filter in seconds				
Resolution	1 thread	2 threads	4 threads	8 threads	16 threads
640 x 426	0.000614	0.000344	0.000301	0.000427	0.025634
1280 x 853	0.002287	0.001298	0.000917	0.000735	0.026294
1920 x 1280	0.00367	0.002947	0.001585	0.001573	0.026378
5184 x 3456	0.018163	0.012665	0.005785	0.005769	0.027821

Table 2. OpenMP Results on Inverting Filter

OpenMP	Execution time with **mirroring** filter in seconds				
Resolution	1 thread	2 threads	4 threads	8 threads	16 threads
640 x 426	0.000755	0.000461	0.000448	0.000453	0.026265
1280 x 853	0.002945	0.001836	0.001149	0.000825	0.026357
1920 x 1280	0.005239	0.002813	0.002117	0.001806	0.027684
5184 x 3456	0.028022	0.025009	0.008852	0.00737	0.028157

Table 3. OpenMP Results on Grayscale Filter

In general, we can see that filter computation time speed up benefits most from having a small number of parallelization. In all instances, a point can be clearly seen where the speedup achieved by the parallelism is reduced due to the increased overhead required by the parallelism. This point is reached earlier in smaller resolutions than in bigger resolutions because the overhead of another thread quickly dominates the small data parallelism benefit that the thread provides. There is also a relationship between image sizes and computation time. As the image size increases, the filter complexity increases, and thus the filter execution time also increases. For the smaller images, we observed that parallelization between 2 to 4 threads resulted in the best execution time speed up. While for the 5184 x 3456 resolution images, parallelization between 4 to 8 threads would be optimal.

Given the results above, we find these trends in our data:

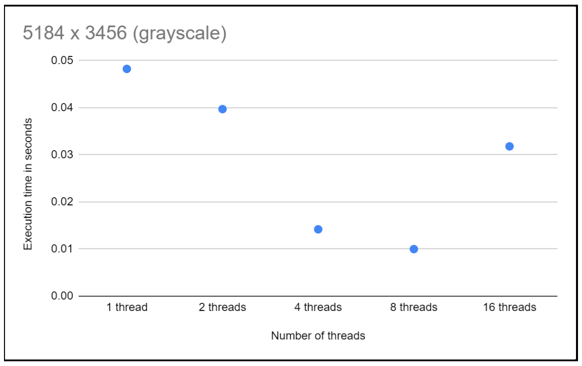


Figure 6. OpenMP GrayScale Results on different threads for image resolution of 5184 x 3456.

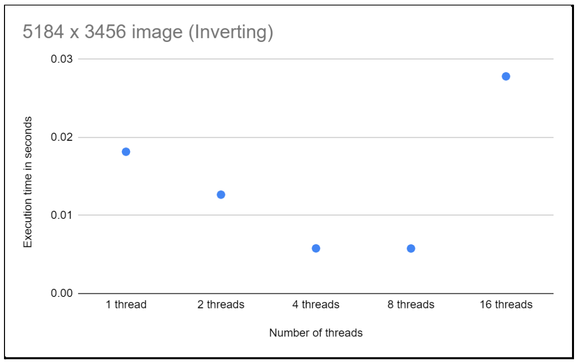


Figure 7. OpenMP Inverting Results on different threads for image resolution of 5184 x 3456.

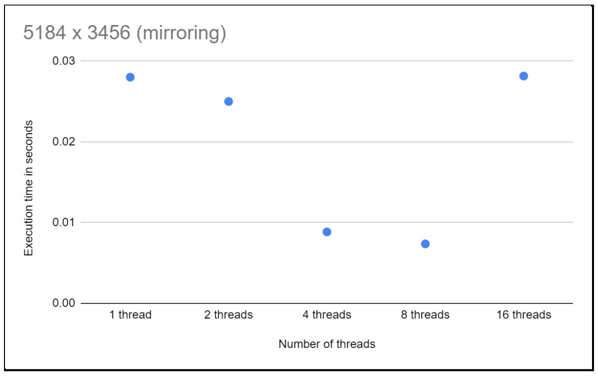


Figure 8. OpenMP Mirroring Results on different threads for image resolution of 5184 x 3456.

From the results in Figure 6, Figure 7, and Figure 6, we see a fairly complete trend for all filter methods. Although not shown explicitly, these trends also carry over for the other image resolution sizes (for each of the filters as well). This is especially interesting given that we chose filters that vary greatly in their implementations. However, these results make sense given the previous explanations of Table 1, Table 2, and Table 3.. To reiterate, although the point at which speed-up varies for each image resolution is different, each result cannot combat difficulties of the overhead. Essentially, each image benefits from a small amount of multi-threading that creates some speed-up, but the lack of programmer control over the increasing overhead makes OpenMP an unfavorable choice for our investigation.

More specifically, the trends in these figures reflect Amdahl's Law (also known as the Law of Diminishing returns), which states that maximum speed-up of a program that is only partially parallelized is limited to its serial portion and is depicted in Figure 9. In the case of OpenMP, although adding more threads could be benefiting the parallelizable portion of our work, the overhead generated becomes so great that it actually decreases the execution times eventually. Compared with the figures for OpenMP, it follows a similar trend. Except, as mentioned earlier, our execution times begin to degrade at later points (or would possibly flatten out with further investigation?), whereas the figure below shows a flattening of the execution time.

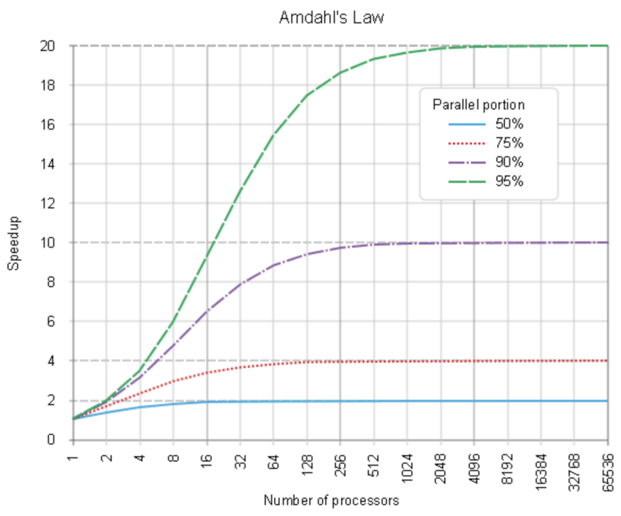


Figure 9. Depiction of Amdahl's Law [8]

5.2. CUDA

CUDA	Execution time with **grayscaling** filter in seconds				
Duration	Warmup	Data Transfer	Computation	Filter Time	Total Time
640 x 426	2.403044	0.000449	0.000112	0.000561	2.403605
1280 x 853	2.400405	0.001134	0.000201	0.001335	2.40174
1920 x 1280	2.402771	0.002434	0.000321	0.002755	2.405526
5184 x 3456	2.39636	0.016561	0.001556	0.018117	2.414477

Table 4. CUDA Results on Grayscale Filter

CUDA	Execution time with **inverting** filter in seconds				
Duration	Warmup	Data Transfer	Computation	Filter Time	Total Time
640 x 426	2.400061	0.00043	0.000109	0.000539	2.4006
1280 x 853	2.400222	0.001148	0.000194	0.001342	2.401564
1920 x 1280	2.405129	0.002344	0.000305	0.002649	2.407778
5184 x 3456	2.411985	0.016479	0.001552	0.018031	2.430016

Table 5. CUDA Results on Inverting Filter

CUDA	Execution time with **mirroring** filter in seconds				
Duration	Warmup	Data Transfer	Computation	Filter Time	Total Time
640 x 426	2.401361	0.000429	0.000107	0.000536	2.401897
1280 x 853	2.384221	0.001161	0.000214	0.001375	2.385596
1920 x 1280	2.407786	0.002336	0.000282	0.002618	2.410404
5184 x 3456	2.41222	0.016451	0.001283	0.017734	2.429954

Table 6. CUDA Results on Mirror Filter

As we can see from the above statistics, the total execution time of the CUDA image processor is heavily dominated by the warm up time, hovering around 2.4 seconds for all images. This is due to the fact that we are allocating a brand new CUDA device that is used by our program on the Discovery Cluster. In a real use of our application, the host device would already have the GPU allocated to the device and ready to be used, so the warmup time can largely be ignored when comparing with other filters. Because of this, the filter time is the best reference speed comparison. The filter time is the combination of the time it takes to transfer the data to and from the GPU, as well as the computation time which is the time it takes for the GPU to execute the kernel. Tables 4, 5, and 6 show the execution time when there is no unrolling. While data transfer and computation time increase according to the Resolution of the Image. Compared to OpenMP, we can see considerable performance improvement for all filters across all image sizes.

5.3. CUDA with Unrolling

Unrolling	Computation time with **grayscale** filter in seconds			
Unrolling factor	1	4	16	64
640 x 426	0.000112	0.000109	0.000111	0.000114
1280 x 853	0.000201	0.000178	0.00018	0.000191
1920 x 1280	0.000321	0.00025	0.000261	0.000284
5184 x 3456	0.001556	0.001219	0.001114	0.001161

Table 7. CUDA Unrolling Results on GrayScale Filter

Unrolling	Computation time with **inverting** filter in seconds			
Unrolling factor	1	4	16	64
640 x 426	0.000109	0.000115	0.000126	0.000136
1280 x 853	0.000194	0.000184	0.000191	0.000196
1920 x 1280	0.000305	0.000269	0.000275	0.000326
5184 x 3456	0.001552	0.001242	0.001217	0.001274

Table 8. CUDA Unrolling Results on Inverting Filter

Unrolling	Computation time with **mirroring** filter in seconds			
Unrolling factor	1	4	16	64
640 x 426	0.000107	0.000117	0.000118	0.000126
1280 x 853	0.000214	0.000171	0.000177	0.000191
1920 x 1280	0.000282	0.000236	0.000255	0.000284
5184 x 3456	0.001283	0.001048	0.00109	0.00116

Table 9. CUDA Unrolling Results on Mirroring Filter

The unrolling factor in the unrolling data is the number of pixels that each thread is responsible for. Unrolling attempts to mitigate the overhead of additional threads by having each thread do more. There is a tradeoff between the parallelism achieved by more threads and the overhead required to use those additional threads. Unrolling attempts to find the most efficient division of pixels into threads so that parallelism is utilized but overhead is minimized. The results of our test shows that the most efficient unrolling setting differs depending on the resolution of the image, which is directly proportional to how much data needs to be processed. For smaller images, the parallelism of more threads is more beneficial than the overhead reduced by unrolling, so an unrolling factor closer to 1 is seen to be most effective. However, for larger images, a larger unrolling factor is more helpful to limit the very large number of threads. However, the highest unrolling factor tested always resulted in increased overhead, showing that there is a balance that must be found for the unrolling tradeoff.

7. Conclusion

In conclusion, we showed that small levels of parallelism are beneficial to data computation time for OpenMP, as large parallelism can generate enormous overhead that ultimately worsened the performance of the filter computation compared to naive sequential methods. We also showed that computation time increases directly proportional to input data size, as well as that for larger images more parallelism is beneficial.

From our results, we can conclude that CUDA is truly an efficient way to parallelize independent data. The CUDA implementation was orders of magnitude faster than the OpenMP implementation for all resolutions and image sizes when only comparing the execution time of the filters. However, the overhead of data transfer that CUDA incurs made the OpenMP filter implementation overall faster when the data transfer was very large. As with many engineering approaches, there is no one-size-fits-all technique that can be used for a particular application. As demonstrated even within our small dataset, the techniques and approaches to solve a problem must be custom tailored to that particular situation.

8. References

[1] M. Iqbal. “Instagram Revenue and Usage Statistics (2021).” BuisnessOfApps.com. https://www.businessofapps.com/data/instagram-statistics/ (accessed Dec. 12, 2021).

[2] S. Pettersson. “Statistics: How filters are used by Instagram’s most successful users.” Medium.com. https://medium.com/@stpe/statistics-how-filters-are-used-by-instagrams-most-successful-users-d44935f87fa9 (accessed Dec. 12, 2021).

[3] S. Skafisk. “This is How Smartphone Cameras Have Improved Over Time.” PetaPixel.com. https://petapixel.com/2017/06/16/smartphone-cameras-improved-time/ (accessed Dec. 12, 2021).

[4] J. Cook. “Three algorithms for converting color to grayscale.” JohnDCook.com. https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/ (accessed Dec. 12, 2021).

[5] A. Rodriguez. “Read and write bmp files in c/c++.” Wordpress.com. https://elcharolin.wordpress.com/2018/11/28/read-and-write-bmp-files-in-c-c/ (accessed Dec. 12, 2021).

[6] “Read and write BMP file in C.” StackExchange.com. https://codereview.stackexchange.com/questions/196084/read-and-write-bmp-file-in-c (accessed Dec. 12, 2021).

[7] “Getting RGB values for each pixel from a raw image in C.” StackOverflow.com. https://stackoverflow.com/questions/1536159/getting-rgb-values-for-each-pixel-from-a-raw-image-in-c (accessed Dec. 12, 2021).

[8] “Amdahl’s Law.” Wikipedia.org. https://en.wikipedia.org/wiki/Amdahl%27s_law (accessed Dec. 12, 2021).