Canny Edge Detection using Nvidia's CUDA Framework

Mori Ladu

Keywords: Gaussian Filter, Sobel Operator, Non-Maximum Suppression, Threshold Analysis, Hysteresis Analysis

1 Introduction

An edge in an image is defined to be a sharp local change in the intensity of pixels defined by some gradient function. That sharp change in pixels intensity is usually observed as discontinuities in the gradient of pixels in a local region of an image[2]. Edges are quite easy to identify for humans, but the problem of finding edges using programs is quite difficult; there is a a lot of subtle details that are not obvious to a computer. The two main difficulties are noise in the image and subtle intensity changes where discontinuity in pixel intensity is not present.

Canny edge algorithm, developed by John F. Canny in 1986, attempts to solve the two issues stated above. It extends on traditional edge detection mechanisms, Sobel and Laplacian operators for instance, and categorize edges in ways that continue to eliminate noise, but also create relationships between different edges. There are 5 steps in detecting an edge using this algorithm: gaussian filter, gradient operator (sobel in this case), non-maximum suppression, double threshold analysis, and edge tracking by hysteresis. The purpose of of the other steps, beside finding gradients, is to enhanced the edges present in the image: gaussian filter for example is to reduce the amount of noise in the image, while hysteresis analysis is to find relationship between suble edges and strong edges such that not-so-present discontinuities are preserved in the final edges found.

The paper will discuss the reasons for choosing CUDA framework, the general algorithmic description of Canny edge detection, and how each step pans out into the final image output. It will also discuss what convolutions are, and how CUDA can be used to optimize the multiple application of convolution over the image, as we proceed from one stage to another.

2 Algorithm Description

2.1 Gaussian Filter

The first step in the edge detection process is the removal of noise. Here, noise is understood as random variations in pixels in a local region of an image; that is, if there is a pixel with a very high intensity among a group of pixels that do not exhibit similar intensity, then it is likely the case that the pixel does not constitute any part of an edge since there is a discontinuity between its intensity and those of its neighbors. So, we need to eliminate such pixels from the final edge-detected image that we are aiming for. This is done by filtering the image using a blurring algorithm–Gaussian filter–to get an average value for every pixel. This averaging process eliminates discontinuity among pixels because every pixel is a weighted average of itself and its immediate neighbors.

The Gaussian filter used in this step is a discrete approximation of the continuous Gaussian normal distribution function. Normal distribution is given by the equation shown below in 1 dimension. Because our project involves images which are inherently 2-dimensional (can be linearized to 1D, but it takes a lot of effort), we also need to find an appropriate Gaussian function takes that into consideration. Given a standard deviation, weight distribution is given by the following formula in one dimension:

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((x-\mu)/\sigma)^2} \tag{1}$$

$$g(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((y-\mu)/\sigma)^2} \tag{2}$$

where :
σ = standard deviation.
μ = average value of the distribution
e = exponent function.

$$g(x,y) = g(x) * g(y) \tag{3}$$

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x-\mu)^2+(y-\mu)^2}{2\sigma^2}} \quad . \tag{4}$$

The discrete approximation of the Gaussian normal distribution function, that I used to calculate my kernel is given by:

$$H_{i,j} = \frac{1}{2\pi\sigma^2} exp^{-\frac{(i-(k+1))^2+(j-(k+1)^2}{2\sigma^2}} \tag{5}$$

where: $1 \leq i, j \leq (2k+1)$ and $k \in N^1$

Using this discrete approximation, we are able to calculate a 5x5 convolution mask, commonly used kernel in gaussian filters, which is then used in smoothing out noises in an input image. The catch in this step, however, is finding the right σ. For our purposes too, we use a commonly used standard deviation, $\sigma = 1.4$. The effect of adjusting results in different ways of blurring. Higher standard deviation approximates an unweighted intensity average of a group of pixels in an image, whereas a smaller standard deviation corresponds to a heavily weighted average that is biased towards the pixel we are considering and its immediate neighbors: the further you are, in this case, the less of an effect you have on the final blurred pixel.

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

Figure 1: A 5x5 gaussian mask generated from $\sigma = 1.4$

The effects of this blurring is quite subtle, not noticeable to the human eye. Ofcourse, we could make the size of the gaussian kernel a lot bigger to get a noticeable blurring effect, but for our purpose here, we are just trying to smooth out some noise in the image. The process of smoothing out edges has its own tradeoffs: not-so-present edges are smoothed out at the same time when eliminating noise in an image using the Gaussian. Finding the right kernel size or σ is dependent on the image, and there is no formula for correctly estimating this values.

2.2 Sobel-Feldman Operator

It is a gradient-based operator that assumes edges are situated at points where there are discontinuities in the gradient intensity function—a function that finds directional change in intensity of an image—or the gradient is very steep. It uses first order derivative to find the gradient values in the x and y direction. The gradient values in the two directions are then combined to get a gradient magnitude and angle. Because being an edge is a local

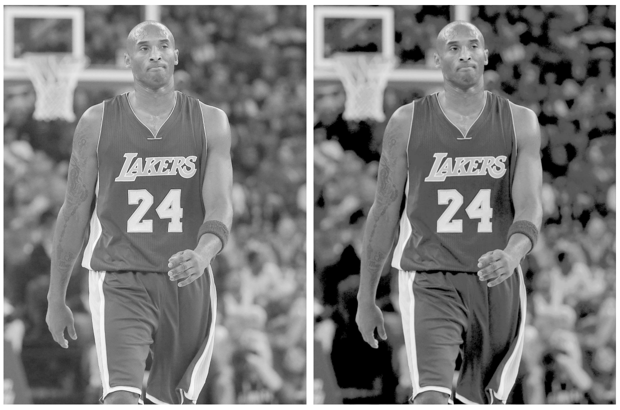

(a) gray image (b) after blurring

Figure 2: Gaussian filter effect is not obvious

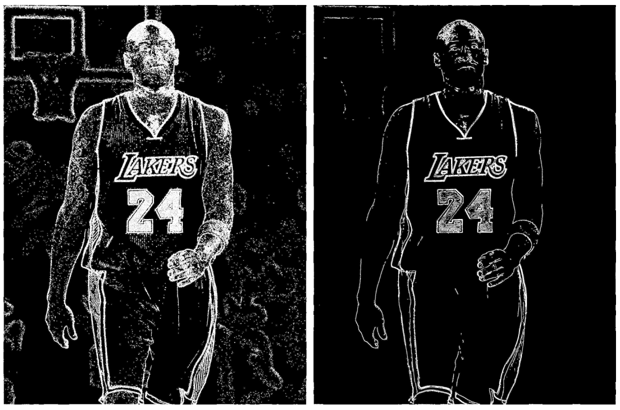

(a) without gaussian filter (b) using gaussian filter

Figure 3: Image noise is pronounced in the final output

[1]courtesy of Wikipedia's Canny Edge Detection page

property of a pixel and its immediate neighbors, the gradient function is usually approximated by a fixed n x n weight matrix, often a 3x3 matrix, and convolve over an image to find the gradient in the x and y direction[2]. The distance between the two gradients is used in estimating the gradient magnitude for an image. At the same, gradient magnitude is calculated using the arctangent function (See Figure 4 for the operators)

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2} \qquad \mathbf{\Theta} = \operatorname{atan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

(a) gradient magnitude (b) gradient direction

Figure 4: 3x3 Sobel-Feldman masks and gradients in the x and y directions, gradient magnitude, and gradient direction, respectively

2.3 Non-Maximum Suppression

Using gradient direction information computed with Sobel-Feldman operators, the arctan angles are categorized into four different angle categories ($0^o, 45^o, 90^o$, and 135^o) based on the proximity of a pixel angle to a line described by each of the four angles. Thus, a pixel with angle of 35^o is categorized as falling on the 45^o line, since it is much closer to this line than the straight line described by angle 0^o.

Collapsing the angles into these four categorizes help in encapsulating the immediate neighbors of the pixel we are examining into the decision of whether that pixel is an edge or not. That is, depending on the angle of the pixel, we consider the immediate neighbors that lay in the line described by the pixel angle, and check if the pixel is a local maximum or not. If that is not the case, then it is suppressed. Otherwise, it is kept for the next analysis phase.

2.4 Double Threshold Analysis

This is an arbitrary grouping of pixels into either strong or weak edges. Two thresholds are defined, low and high, and pixels are categorized based on whether their intensity is less than the low threshold, between low and high threshold, or higher than high threshold. The latter is referred to as a strong edge. A pixel with intensity less than the low threshold is eliminated in this phase. The remaining group of edges are classified as weak edges that may potentially be related to a strong edge or not. The result of this operation is the thining of edges found in the previous step.

2.5 Hysteresis – Edge Tracking

Here, relationship between weak and strong edges are established. The output of double threshold analysis is two sets of edges: strong and weak edges. Clearly, strong edges described actual edges of some sort. This decision is not clear cut for weak edges, however. A weak edge is pronounced as an actual edge if and only if one of its immediate neighbors (8 surrounding pixels, at max) is a strong edge. Weak edges without this property are eliminated and the final image is printed out with all the edges detected.

3 Design and Implementation

Canny Edge detection is an asynchronous multi-step algorithm with each step depending on the previous one. This makes the design of the algorithm as an ordered set of operations, with Gaussian filtering as the first and Edge tracking being the last. My implementation mirrors this behavior; every step in the algorithm is an isolated function that takes input from the functional implementation of the step prior to it.

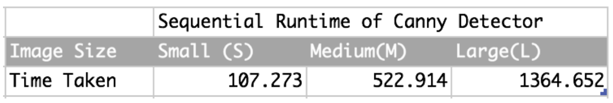
The goal of this project is to find optimizations, particularly compute-bound and memory-bound optimizations, that can effectively enhance the performance of the algorithm. The starting point is to get a frame of reference, which in this case is a sequential implementation running on the host side. Addressing the compute bound issue is the easiest of the two: I've decided to go with GPU and CUDA framework provided by Nvidia. GPUs consists of Streaming Multiprocessor (SM) that performs really well for computations such as image processing and others. The Single Instruction, Multiple Data (SIMD) model of SMs perfectly encapsulates the behavior of image processing: same instruction repeated multiple times over an image. Although GPU memory is a bottle-neck for most applications running on the device, fate of the DRAM technology, data locality and data parallelism from the organization of channels and memory banks on the DRAM, with the property of memory burst, can be cleverly exploited to cover up for the shortcomings. Among the memory-bound optimizations this project uses are the use of constant memory and shared memory using tiling[1].

Besides the sequential implementation, there is a group of "regular" device kernels that simply does computations on data provided without doing any sort of memory optimization. This is a base improvement over the sequential implementation. Among the steps of the algorithm, convolutions—gaussian filter and sobel operators—constitute the most computationally expensive of the entire algorithm, so I decided to optimize the two operations to see if performance would drastically increase. The first step is using constant memory to store convolution masks, with the expectation that using read-only constant memory should drastically decrease memory access time (˜5 cycles for constant memory, whereas for global memory its 500 cycles)[1]. The second memory optimization is to use shared memory in order to reduce the amount of traffic to the global memory. In this step, I also used constant memory to see how the interact interact together. This project also tests for the effect of varying block sizes for device kernel and how that affects the execution time of each kernel.

3.1 Results

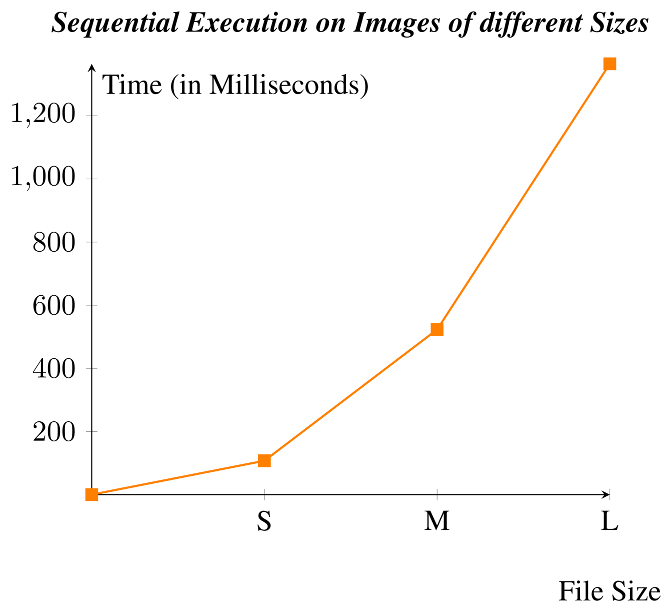
Figures starting from 5 and on are raw values or graphs of elapsed times (in milliseconds) calculated for each function/kernel called. The elapse time for device kernels is found using 'cudaEventCreate'. For the single processor implementation, I used the 'clock()' function to record the elapse time of the entire algorithm. The single processor (sequential implementation) is called by the same name, or in short form, seq. Figure 5 is a table of the execution times for a single processor implementation. The input to the algorithm are three different images, arbitrarily chosen based on their size. The three input images are gorge.jpg as large size, kobe.jpg as medium size, and lena2.jpg as a small size image—these can be found in the same repo as the source code. The increasing image size is to find how each implementation scales when the input gets bigger. The implementation that doesn't use constant memory or shared memory is called "regular" (in short reg). The last two usually have constant and shared declared explicitly such as "Constant Kernel" and "ConstantShared".

Running the sequential implementation with three different size images produces the expected result: increasing image size corresponded to a slower execution. An important observation in the line graph is that the execution time more than doubles as we go from small to medium and medium to large.



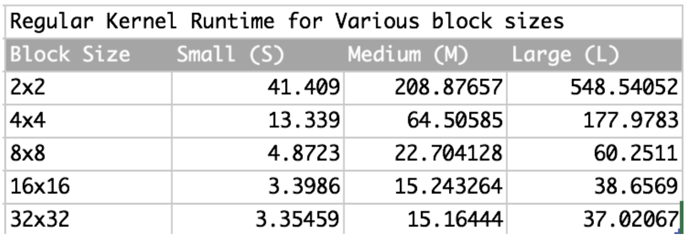
	Sequential Runtime of Canny Detector		
Image Size	Small (S)	Medium(M)	Large(L)
Time Taken	107.273	522.914	1364.652

Figure 5: Elapsed time in Milliseconds



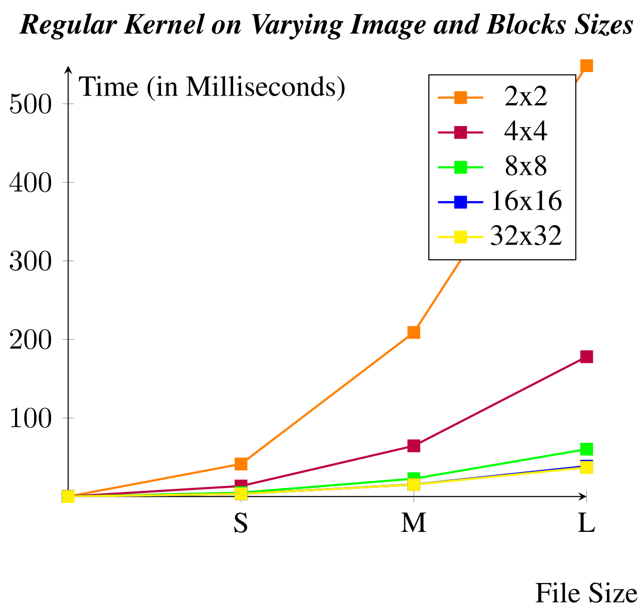
The result drawn from running the regular kernel, with no constant memory or shared memory, is intriguing, to say the least. Although increasing block size shows a drastic improvement in performance, we also start to see the negative effect of large thread size to on increasing performance, particularly going from 16x16 block size to 32x32. In general, more threads in a block corresponds to a better performance until we reach a high latency point whereby the overhead is greater than the performance benefits. This is because threads in a block are organized, scheduled, and executed in sets of 32 called warps. Increasing the number of warps means increasing the likelihood that there is atleast one warp ready to execute whenever others are in a period of latency due to memory access[1].

Similar results are found in the constant memory kernel, as well as the shared-constant memory kernel. It appears that 16x16 block size is probably the closest to the optimal block size. Increasing that further leads to no significant performance benefit because the overhead of scheduling threads of a greater number overshadows the performance benefits we are expecting. It may also be because of thread granularity. Although the convolution is the bottleneck of the entire algorithm, performance-wise, the amount of work each thread does is not that much. The work for each thread is redundant, but the huge number of streaming processors fetching for instruction may unnecessarily consume too much of instruction processing bandwidth[1]. For future references, it would be interesting to consider granularity as part of the optimization and actually see its effects.



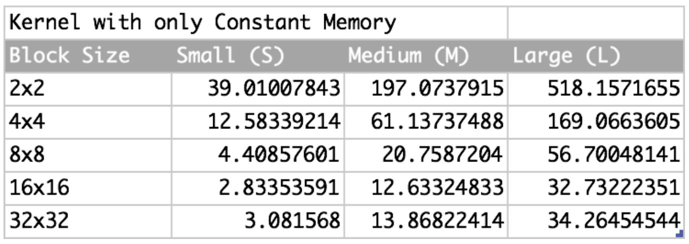
Regular Kernel Runtime for Various block sizes			
Block Size	Small (S)	Medium (M)	Large (L)
2x2	41.409	208.87657	548.54052
4x4	13.339	64.50585	177.9783
8x8	4.8723	22.704128	60.2511
16x16	3.3986	15.243264	38.6569
32x32	3.35459	15.16444	37.02067

Figure 6: Elapsed time in Milliseconds



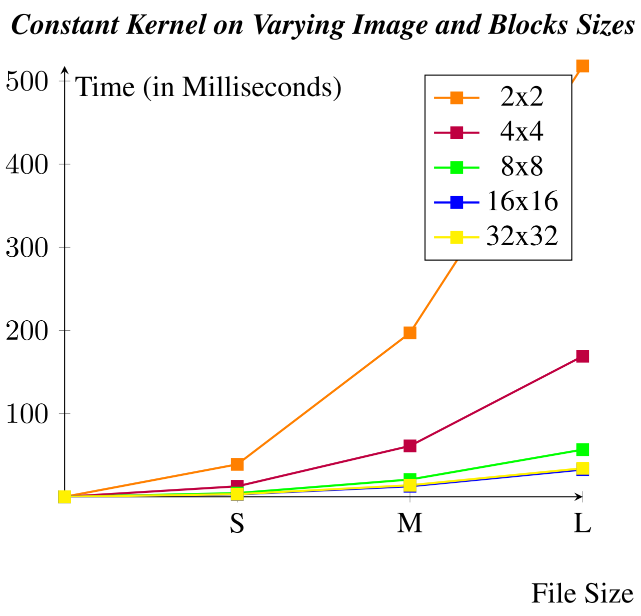
Among the four implementations, for the input images tested, constant memory kernel has the best performance for large, medium, and small size images. Access time for constant memory is obviously faster than the access time for global memory. So, it is expected for a better performance over regular kernel and sequential implementation. However, it is startling that the kernel that uses both shared memory and constant memory does not have the best performance.

Due to the way the relationship between a tile width and block width, when doing tiling, is defined, there is no data that could be collected on a 2x2 or 4x4 block

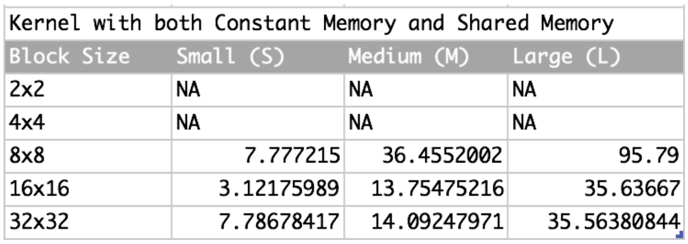


Kernel with only Constant Memory			
Block Size	Small (S)	Medium (M)	Large (L)
2x2	39.01007843	197.0737915	518.1571655
4x4	12.58339214	61.13737488	169.0663605
8x8	4.40857601	20.7587204	56.70048141
16x16	2.83353591	12.63324833	32.73222351
32x32	3.081568	13.86822414	34.26454544

Figure 7: Elapsed time in Milliseconds



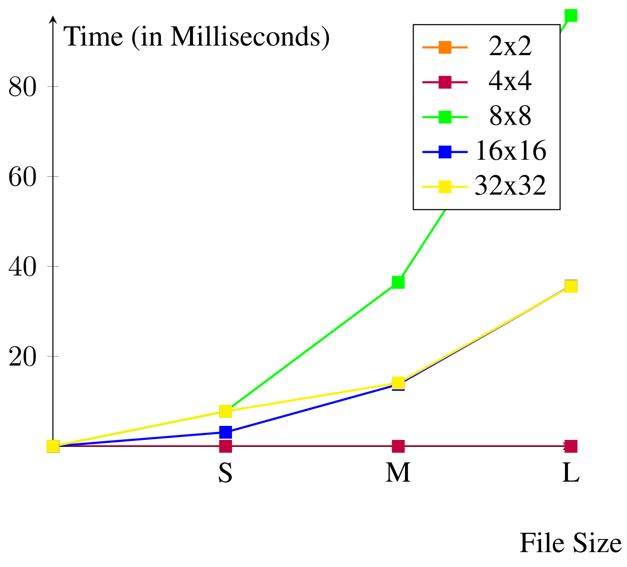
size. Plus, we are also limited by the maximum number of threads per block, so I can't draw much conclusion from it. The following graphs compares the performance of sequential implementation against the three optimizations, fixing block size in each case. From these results, we can see that there's relatively little performance difference between the three optimizations, besides the fact that the shared, constant kernel usually has a slow start to the rest when the block size is small. This decent performances from the kernels that retrieved all of their data from the global memory, despite that being a high latency request, points to one concept on data locality. Because arrays are usually allocated as contiguous chunks of data, subsequent requests to a block of data can result in coalescing, whereby memory requests for contiguous chunk of memory are combined to become one request for that chunk of memory. That'll explain why the other kernels who call on the global memory perform really well, in spite of the access time penalty.



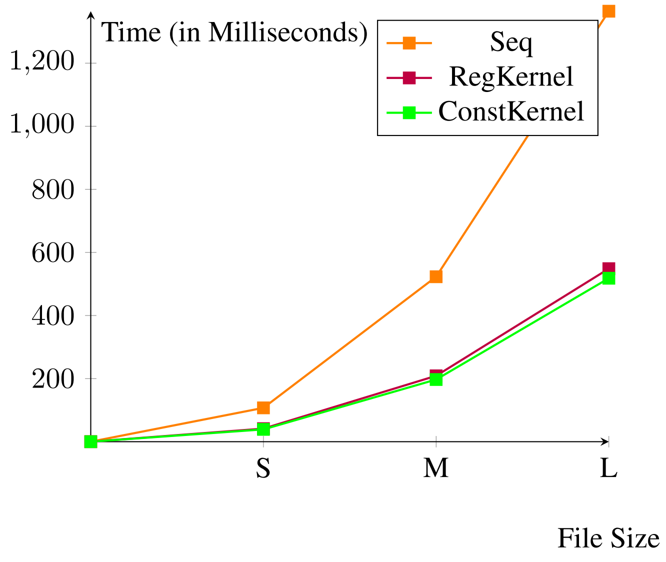
Kernel with both Constant Memory and Shared Memory			
Block Size	Small (S)	Medium (M)	Large (L)
2x2	NA	NA	NA
4x4	NA	NA	NA
8x8	7.777215	36.4552002	95.79
16x16	3.12175989	13.75475216	35.63667
32x32	7.78678417	14.09247971	35.56380844

Figure 8: Elapsed time in Milliseconds

Constant&Shared Kernel on Varying Image and Blocks Sizes



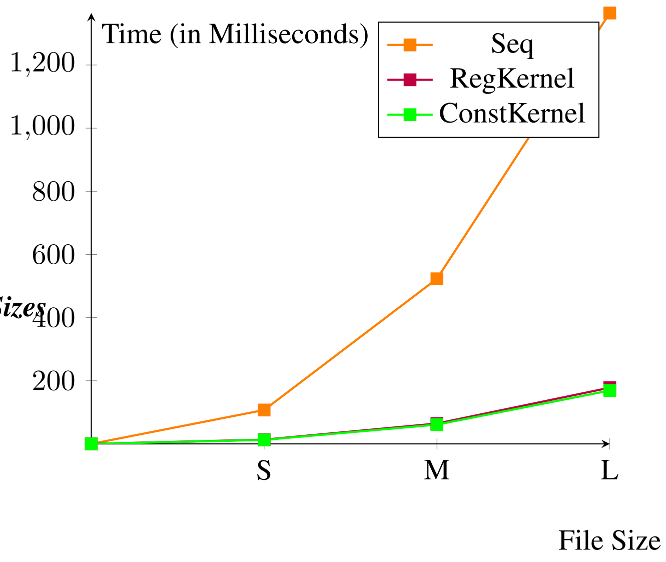
Sequential Against 2x2 Block Optimizations



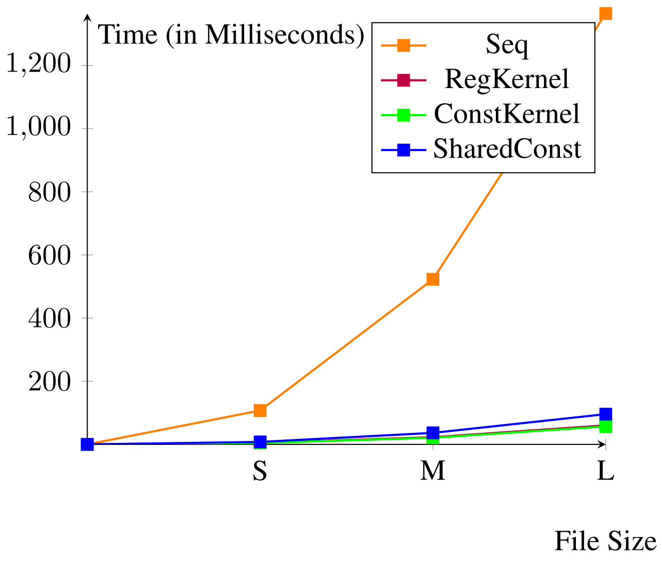
4 Conclusion

Theoretically, shared memory kernel should have the highest performance of all kernels. The results however point to an equal footing among the three device imple-

Sequential Against 4x4 Block Optimizations



Sequential Against 8x8 Block Optimizations

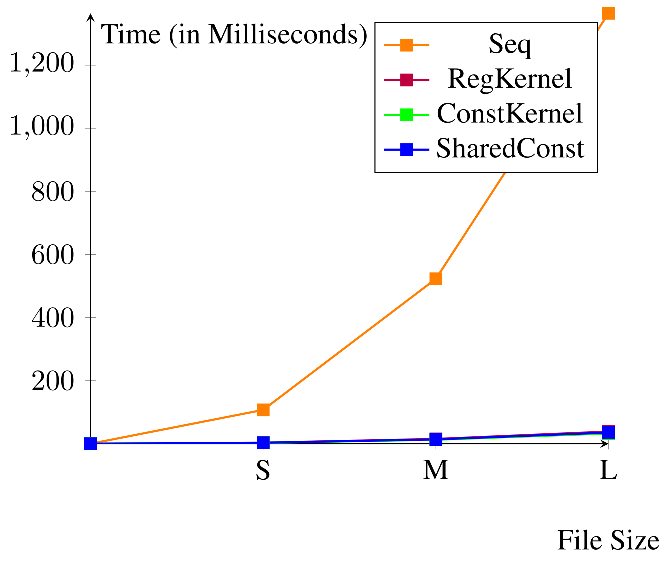


mentation. With big image sizes and block sizes, shared memory and tiling shows its optimization power as compared to the other kernels. For small images however, any other device kernel implementation can do a good, if not better, job of constructing an edge detected image.

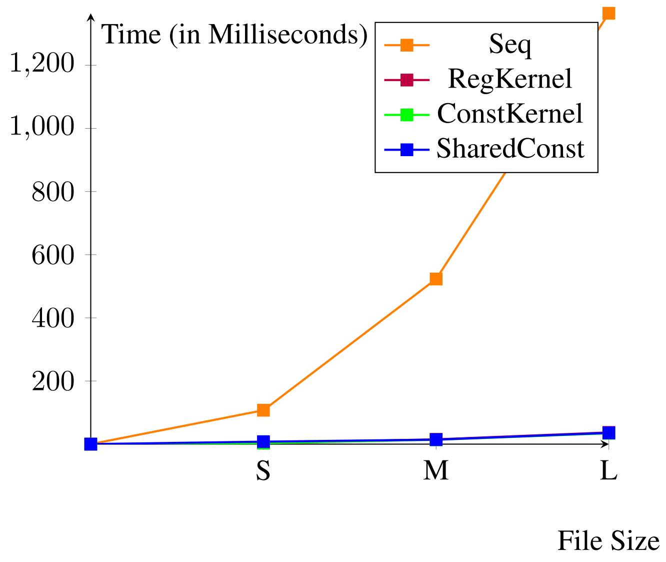
5 Future Work

Gaussian filter is amazing in removing noise from an image. But, it is also non-discriminatory; subtle edges that are obvious to the human eye can be treated as image noise, resulting in an image produce with many edges unpronounced edges left out. An improvement to the algorithm may include replacing Gaussian filter with an al-

Sequential Against 16x16 Block Optimizations



Sequential Against 32x32 Block Optimizations



ternative self-adaptive filter proposed by Bing Wang and ShaoSheng Fan[2].

Due to time constraints, some assumptions, I did not get to test the effect of removing if-statements out of for-loops. I've previously used wrap-arounds to handle the edge cases—when a pixel is in the border—to avoid using if statements for bounds checking. That started to add unnecessary edges to the bounds of the image, as some pixels from the wrap around have a high intensity that may flow into the pixel we are looking into. Because of that, I reintroduced bounds checking into for-loops. This is obviously bad due to thread divergence in a warp. I simply ignored this due to the negligible length of thread divergence in a warp (there is not a lot of instructions executed between threads divergence and convergence). It would be great to test how much of an improvement this optimization would make.

Sensitivity to different variables is also another issue in this project. Choosing the right σ, convolution mask size, low threshold, and high threshold is very difficult. One choice may have good result on one image but perform terrible on another image. For instance, an aggressive gaussian filter does well on a busy image such as gorge (the large image used in this paper). That aggressiveness eliminates a lot of fine edges on other images. The choice of thresholds, too, contributes a lot to the final output. A possible future work is to find the sweet spots to these variables so that they may perform good on average.

The last thing I wished the project would exploit is the property of memory bursts in GPUs. The matrices are copied directly from the host to the device, without padding with some inputs to adjust rows with the number of memory banks in a device, so that instead of doing 5 memory bursts to get a full row, we could use the same amount of memory bursts to get two rows. Sanitizing the data in that form would be tremendous to understanding how memory retrievals work in the GPU and particularly how it pans out for this project.

[1] and [1, 2]

References

[1] D. B. Kirk and W. W. Hwu. Programming Massively Parallel Processors. *Third Edition*, 2013.

[2] B. Wang and S. Fan. An Improved CANNY Edge Detec-tion Algorithm. In *Second International Workshop on Computer Science and Engineering, Qingdao*, Place (Country), 2009.