Christopher Pack, Alexander Vilesov, Alex Yin
EE/CSCI 451
Professor Prasanna
6 December 2019

Parallel Canny Edge Detection on CPU and GPU

Introduction to Edge Detection

Edge detection involves finding boundaries in images by looking at areas with discontinuity in brightness. For canny edge detection, the images must be in grayscale so finding edges is based simply on the brightness of each pixel (0-255) relative to the values of its neighboring pixels and differences in color such as red vs blue, are not directly taken into consideration.

Canny Edge Detection has several real world uses. It can be used to segment images into different pieces, but some more interesting uses are in the field of machine learning. These include being one step of computer vision on autonomous vehicles to detect the locations of other cars and the relative positions of lanes. It is also used in the medical field to detect “fingerprints” of diseases, such as tumors, by looking at images or scans of a patient. And of course can be used in detecting actual fingerprints by looking at the ridges in the photo of a person’s finger.

Edge detection is a good topic for parallelization because it is not feasible to do real time edge detection on a single core CPU. In our testing, running serial code gave us less than 2 FPS, which is nowhere near good enough for use in a real time situation such as a car or a robot. By using CUDA and parallelizing the problem on an Nvidia GPU we were able to get a 100x speed up and run at 200 FPS. So by parallelizing the code the algorithm becomes more useful since it can be used in many more situations.

The Canny Edge Detector Pipeline

The Canny Edge Detector begins with an input of a gray-scale image. The first step in the process is to apply a Gaussian blur. The Gaussian blur operates as a low pass filter and attempts to keep the most salient edges/features of the image. The Gaussian blur is calculated through the use of a two-dimensional Gaussian probability density function. The kernel size and standard deviation that is used can be variable, but the kernel size has to be an odd number that is greater than one.

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right); 1 \leq i, j \leq (2k+1)$$

The next step of the image-processing algorithm is called the gradient calculation. This involves using the classical Sobel operator. The Sobel operator detects the edges in the X and Y direction of the image with convolutional kernels Kx and Ky.

$$K_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, K_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

Afterwords, for each pixel, the magnitude($|G|$) and angle (θ) of the edge are calculated using the equations to the right. This operation provides the advantage of being able to analyze a pixel's edge in any direction and not being limited to the X and Y axis.

$$|G| = \sqrt{I_x^2 + I_y^2},$$

$$\theta(x, y) = arctan\left(\frac{I_y}{I_x}\right)$$

The third step of the algorithm is called Non-Maximum Suppression. This stage tends to thin thick edges and remove unnecessary noise. It is done by first analyzing the angle (θ) of each edge in the image. The angle is then rounded to the nearest angle that is a multiple of 45 degrees since surrounding pixels can only appear at those angles (0, 45, 90, 135). Canny will then look at pixels that are perpendicular or orthogonal to the edge's angle. For example, if pixel (i,j) had an angle of 45 then the orthogonal angle of 135 would contain pixels (i-1, j+1) and (i+1, j-1). If either of the orthogonal pixels has a magnitude that is greater than the original pixel, it is set to a value of zero.

The fourth step is called thresholding. In this step Canny will group all pixel edges into 3 groups: Strong edge, Weak edge, and Ignored edge. It also requires two inputs from the user: upper-bound and lower-bound. For each pixel, if their value exceeds the upper-bound it is labeled as a Strong edge. If the value is in between the values it is labeled as a Weak edge. If the value is below the both bounds, the edge is set to zero automatically and has no chance of becoming an edge in the final image. Finding the correct bounds to get a 'good' edges in an image is tedious, so algorithms such as the Otsu method are implemented to find these bounds automatically for the user (this algorithm has not been applied in this project)

The final step of the Canny algorithm is called hysteresis. This step applies a Breadth First Search type approach to reach the final output of the algorithm. Instead of beginning with a single source node in the que, all Strong edges are put in the que. Then, for each Strong edge, Canny looks if there are any adjacent Weak edges (in a 3x3 or 5x5 search grid with the pixel in the center). Any Weak edges that are found are added into the que and the search repeats until no more Weak edges can be added. As a result all pixels/edges in the que are set to a value of 255 (white) and any Weak edges that were not reached are set to 0. The resulting image is the final product of the Canny Edge Detector algorithm.

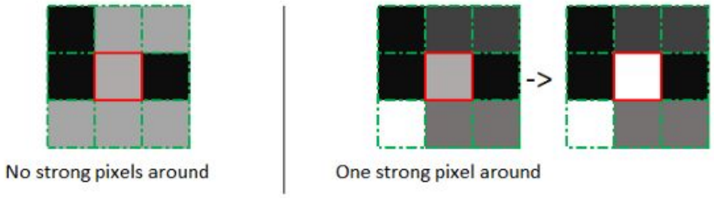
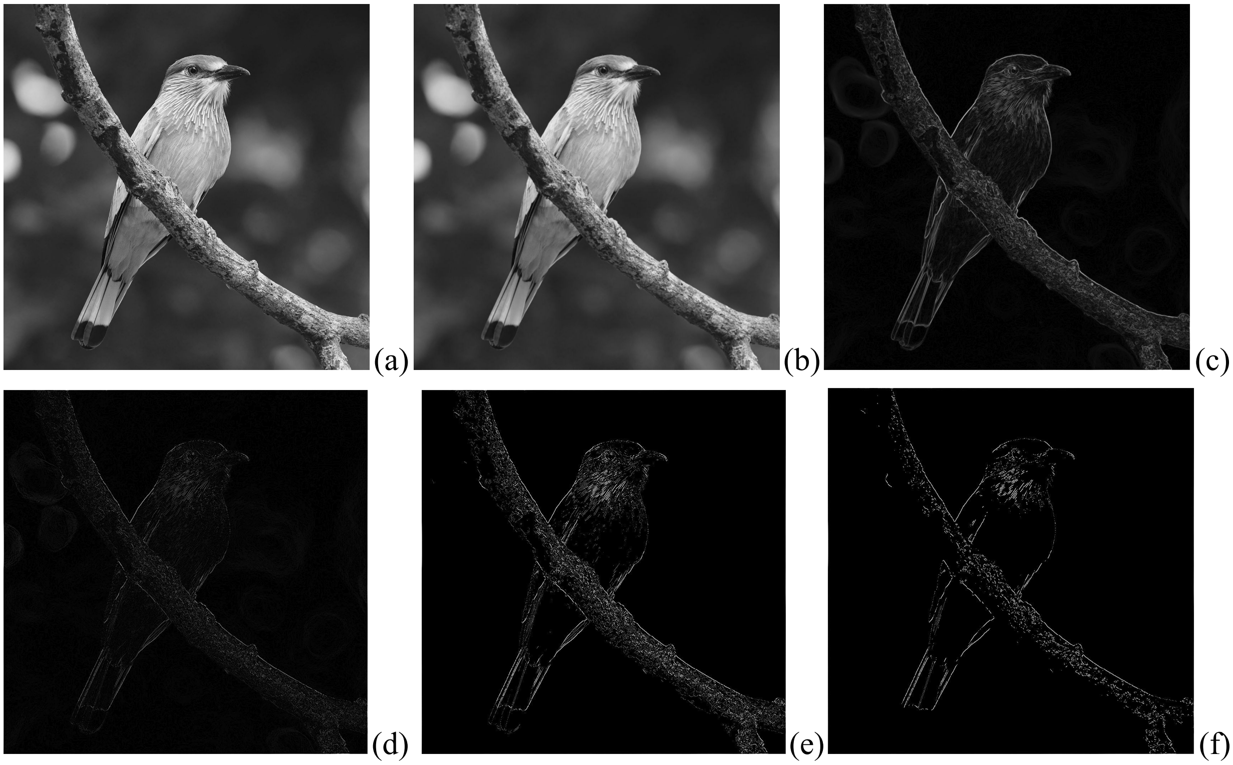


Image After Each Phase of the Algorithm

(a) Original Image (b) Gaussian Blur
(c) Gradient (d) Non-Maximum Suppression
(e) Thresholding (f) Hysteresis



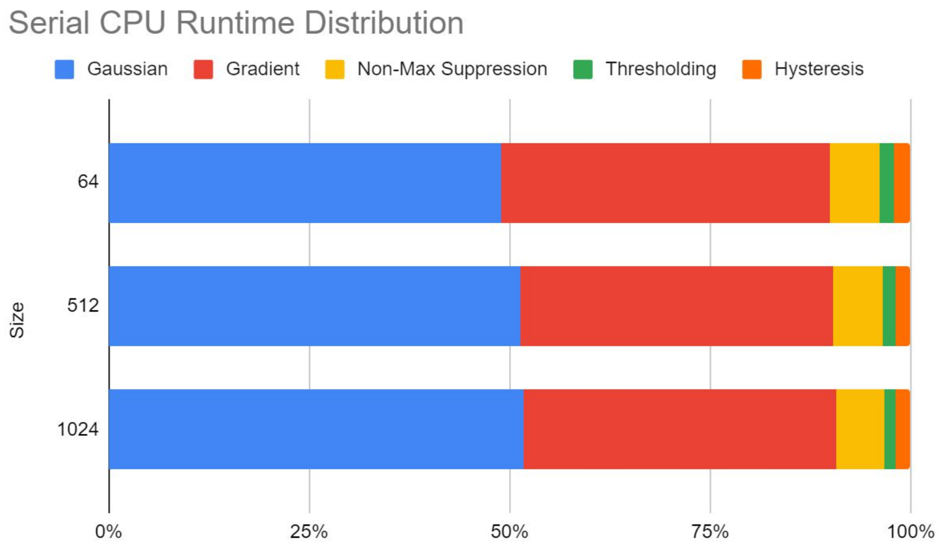
Experimental Setup

We used a Dell Workstation with a 16 core Intel Xeon Gold CPU (Base clock 2.1 Ghz, Max Turbo 3.7 Ghz) equipped with 128GB of DDR4 RAM. This was paired with an Nvidia P5000 GPU that has 16GB of onboard memory, 288 GB/s bandwidth, 16GB/s PCIe to the motherboard and 2560 CUDA cores running at 1.2 Ghz. We choose this setup because of the options we had available to us, it was the most powerful and would best approximate what might be used in a setup for a simple AI application that might also need to use edge detection. We were able to get our FPS up to 400 on 1024x1024 images while running on this computer using our CUDA code. We used 100-200 online images for each of size 64x64, 128x128, 256x256, 512x512, 1024x1024 of flowers, animals, berries and nature.

Objective and Project Hypothesis

Canny Edge detection is one of the most computationally expensive out of all other edge detectors. Our objective is to reach at least high 60 FPS for high resolution images (1024x1024) through the use of parallel computation in a CPU and GPU setting. We wanted to see how the speed of both the CPU and GPU would be affected by the problem size. We believed from the beginning that the GPU would outperform the CPU but were able to back up that idea with statistical results.

Analysis of Serial CPU C++ Code



In running the serial C++ code we saw that Gaussian blur and gradient calculation were the largest portions of the runtime. The proportion of time spent on each part of the pipeline remained constant with image size. From this, it can be deduced that all parts are all proportional to the image size by the same factor, meaning that the time spent in each part of the pipeline remains the same, relative to other parts. We took this analysis and because of Amdahl's law focused the most effort on improving the times that took the longest. However, we parallelized in the entire pipeline by the end of the project so there are nearly equal improvements across all phases of the algorithm.

Specifically, the serial C++ code depends upon openCV for reading and visualizing images, and the cmath library for mathematical functions like exponential and arctan2. The algorithm currently works with the assumption that images from the same dataset are fixed size but can be easily modified to accommodate datasets with variable sized images. Both our serial and parallel CPU implementations require enough memory to hold two 2D float arrays of size (height,width), because canny edge detection cannot be done entirely in place, due to the nature of convolution operations. As a result, the two allocated arrays swap places as input and output arrays of the pipeline parts. Each part of the pipeline is implemented in a function, and all parts share a similar structure to the below code.

```
int padd = kernel_size / 2;
for(int i=padd; i<h-padd; i++){ // height
    for(int j=padd; j<w-padd; j++){  // width
        // do some operation, below is conv, could be nonmax, grad, etc.
        for(int k_i=-padd; k_i<=padd; k_i++){
            for(int k_j=-padd; k_j<=padd; k_j++){
                b_img[i][j]+= img[i+k_i][j+k_j]*gauss[k_i+padd][k_j+padd];
            }
        }
    }
}
```

In terms of runtime, each part of the pipeline can be shown to be in the approximate runtime of the below table.
n refers to image size in one dimension.
k refers to kernel size in one dimension
c refers to some scalar value that depends on the content of image

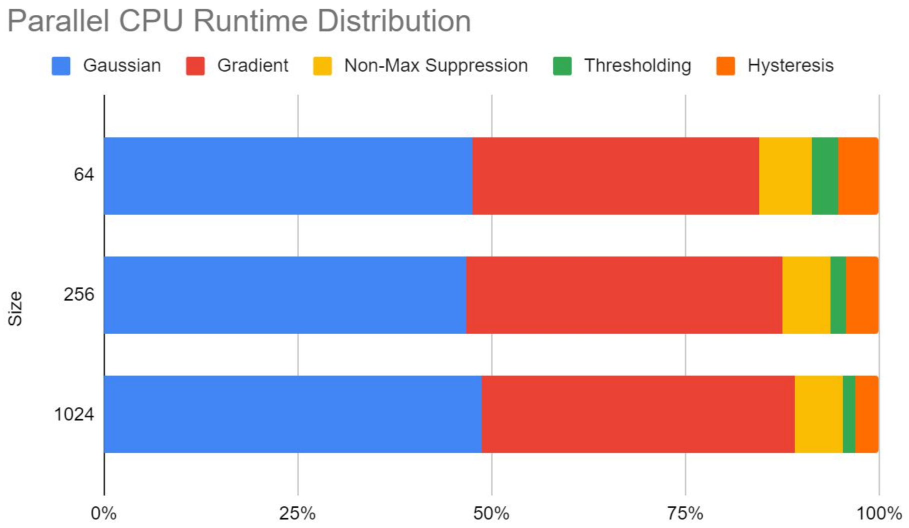
Part	# of Operations	Upper-bound Time Complexity
Gaussian Blur	$2n^2k^2$	$O(n^2k^2)$
Gradient Calculation	$4n^2k^2 + 4n^2$	$O(n^2k^2)$
Non-max Suppression	$4n^2$	$O(n^2)$
Double Thresholding	n^2	$O(n^2)$
Hysteresis	cn^2	$O(cn^2)$
Total	$6n^2k^2 + (9+c)n^2$	$O((k^2+c)n^2)$

It should be noted that there are some algorithm edge case simplifications made on our part to reduce the complexity of the problem that technically should be accounted for. First, the edge pixels of the image are ignored as opposed to the more common zero-padding or reflection-padding approaches that are used in all the common computer vision libraries like OpenCV. Second, there are some data dependencies in the hysteresis portion of the parallel CPU implementation that are not handled but affect only a very small number of pixels a very small percentage of the time. In the hysteresis part, there are data dependencies that can occur that will lead to unpredictable and unintended behavior in some low threshold pixels for CPU implementation. This problem seemed very difficult to resolve and ultimately not so critical to the main focus of this project, so we decided to ignore this edge case.

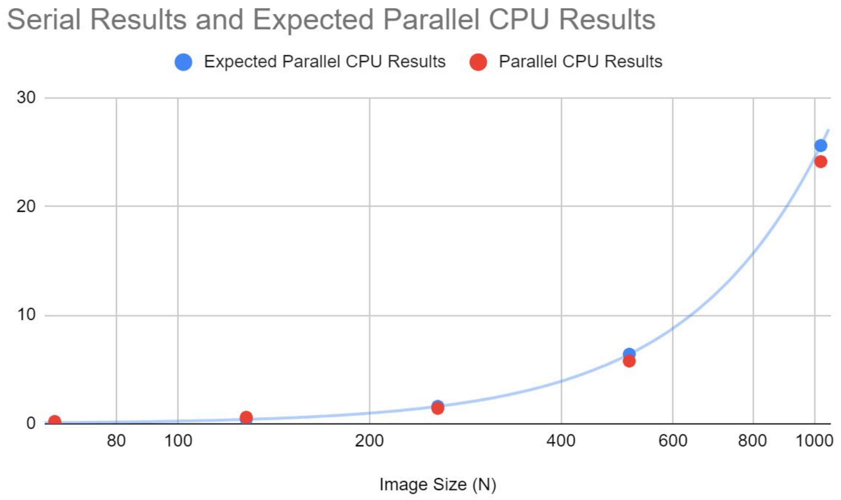
CPU Implementation

As mentioned in the serial implementation, the canny edge detection algorithm splits into 5 parts that iterate through the image and apply some operation with few data dependencies. As a result, the CPU parallelism can be implemented very intuitively using OpenMP.

Each of the five parts of the pipeline can be split into thread blocks with each thread receiving rows of the shared image array. This parallelism strategy fits strongly into the CREW paradigm.



As hinted by the time distribution, all parts of the pipeline scale equally with each other as N increases. In fact, the distribution is nearly identical with the serial distribution, indicating that all parts are sped up by a shared speedup factor, which given the structure of the code and the parallelism strategy used, makes sense. Although in theory the gradient calculation should have taken longer since there are more operations done in that part, the ‘0’ values in the sobel kernels leads to multiplications with ‘0’, which are computed very quickly. The exponential operation in the gaussian blur part may also be especially costly. Additionally, it appears that the kernel size factor term $\mathbf{4n^2k^2}$ heavily outweighs the square root and arctan term $\mathbf{4n^2}$, despite the fact that square root and arctan are more expensive operations than simple addition or multiplication.



The expected parallel CPU result points are calculated by the equation $T(N) = N^2 / 40960$, which fits both our actual results and the runtime calculations made earlier. Additionally, it did not seem feasible to further optimize the gaussian blur and gradient calculation bottlenecks, simply because those two parts were already operating at near maximum efficiency from a CREW-PRAM perspective. There are no race conditions or synchronization points within the bottleneck parts, which makes it difficult to further parallelize beyond the current status.

GPU Implementation

The implementation of the Canny algorithm can vary widely from program to program so the following gives a detailed description of how Canny was implemented on a GPU and some of its limitations. The program also has certain biases that come from being programmed with the given machine. Our GPU program was written in C++ and used CUDA to interface with the GPU.

The convolutions have two implementations where one uses solely global memory and the other optimizes memory latency by using shared memory. The convolution operations in the naive method using only global memory are implemented in a similar manner for both the Gaussian Blur and Gradient Calculation. A thread is allocated to each pixel where a convolution will be calculated. The dimensions for the grid and blocks are calculated as follows for an image of size *M* by *N* with kernel size *K*:

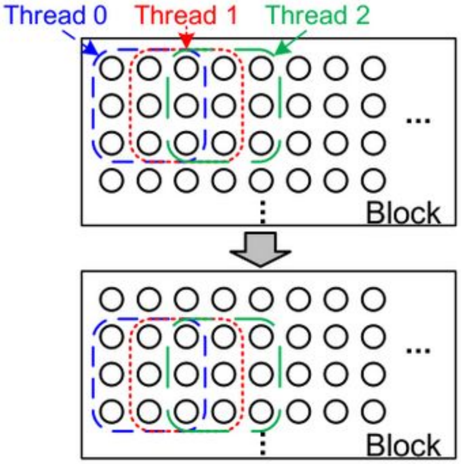
Thread Block = M-K+1   *Grid = N-K+1*

The additional constant of |*-K+1*| equals the padding of the image during the convolution calculation, since pixels on the edge of the image do not have necessary boundary pixels to complete the convolution operation. While the Gaussian Blur only executes one convolution per pixel, the Gradient Calculation kernel executes two Sobel convolutions and then calculates the magnitude and angle of the resulting edge.

The shared memory implementation differs in several ways (Chen). First of all, the dimensions for the grid and blocks are:

Thread Block = N-K+1   *Grid ≥ number of streaming multiprocessors = SM*

Each thread block calculates a convolution for a whole row of the image. During the initialization step, a shared array of size KxN is created to store pixels the convolution is dependent on. However, instead of each thread executing one convolution per pixel, the thread calculates *M/Grid.dim* convolutions. After each row of convolution, the entire thread block moves down one row. Therefore, the top row of the shared array is not needed for the next convolution. A cyclic shift of the rows in the shared array is performed and the last row of shared memory is replaced with new pixel values



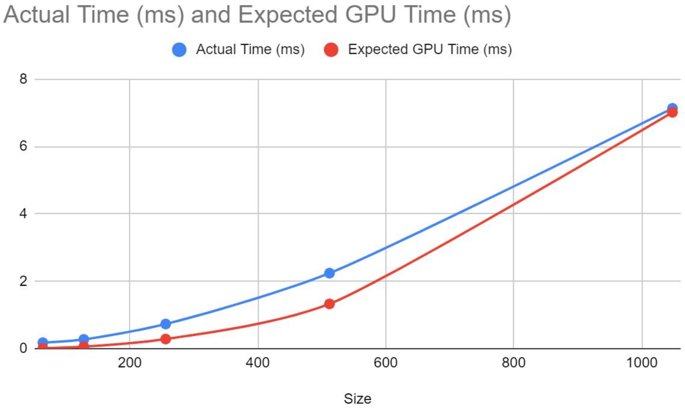
from global memory. Additionally, the grid dimension has to be greater than the number of streaming multiprocessors or else not all CUDA cores will be active in the convolution steps.

The Non-Max Suppression kernel implements similar Block and Grid layouts as in naive GPU convolution. It is worth noting that when a thread in this part of Canny reads and writes pixel values, data races may occur if writing of pixel values is done in place. To avoid the need for synchronization, all writes occur in a separate 'copied' array of the image.

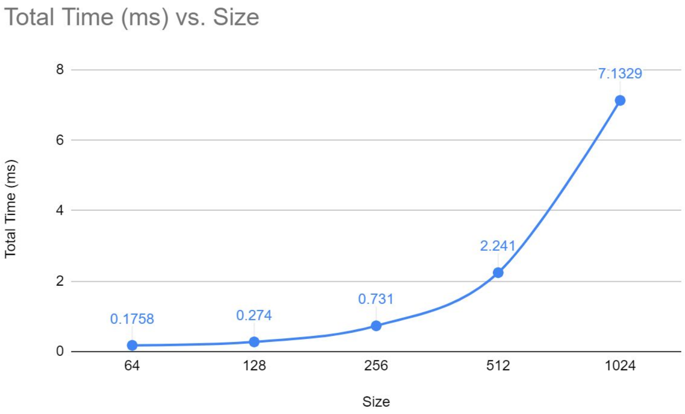
The final kernels in Canny, thresholding and hysteresis also use the same Block and Grid layouts as in naive GPU convolution. Thresholding is an embarrassingly parallel process and each thread straightforwardly analyzes if its pixel is a strong, weak, or an ignored edge. Many implementations of the hysteresis portion are done by splitting the image into blocks and assigning each block to a thread which would then do a mini-BFS search on the graph assigned to it. However, this poses the issue of how to connect weak edge segments that are located across various blocks. This can be solved by having the divided image blocks overlap with each other, however, this increases the runtime. Instead, a different implementation runs the BFS abstractly (Emrani). Each thread is assigned to a vertex (pixel), with the same block and grid configuration as the naive GPU convolution kernel. During each iteration of the BFS, the thread checks if it is a weak edge and if it should be changed to a strong edge. However, instead of completing all the iterations of the BFS in one kernel, each time the kernel is launched it does one iteration of the BFS. Therefore, the new hysteresis kernel will keep launching until there are no changes in the image.

GPU Results vs Expected

Size	Total Time (ms)
64	0.1758
128	0.274
256	0.731
512	2.241
1024	7.1329



Our GPU code ran in approximately N^2 time which matches our expectations as the parallel algorithm we implemented does a number of memory accesses for each output pixel, and thus is dependent on N^2, the size of the image. We calculated our expectations by adding together 3 different equations, one for memory

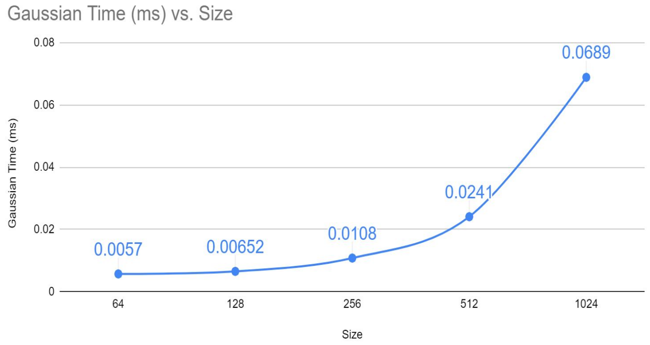


access time, one for computation time, and one for CPU↔GPU data transportation.
*Memory Time = N^2 * 4 * (47 + 10*Average BFS Depth)*
There are N^2 outputs, we used ints which are 4 bytes, and then there are 47 accesses to memory for the image.
*Computation = N^2 * (60 + 500 + 50 + 30 + 50*Average BFS Depth + 180)*
Again, N^2 output and the numbers are all based on the number of computations that are done in each step. However, the 180 is added because of the use of two arctan functions in the code and they take about 90 times as long as a basic operation like addition, thus we had to account for that.
*CPU/GPU communication = N^2 * 4 * 2 / 16 GBPS*
This is based on communicating N^2 ints in both directions and then dividing by the bandwidth of a PCIe 3.0 x16 slot of 16 GBPS. This CPU/GPU communication only exists in the Non-Maximum Suppression stage.

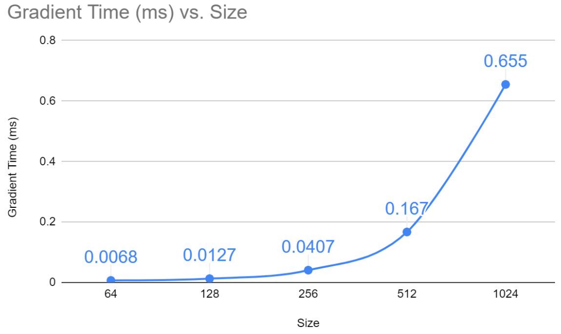
Timing Analysis for each Stage of Canny

We used Nvidia's built in timing functions to profile the different sections of our code and see how long each section of code took and also how it changed as the input size changed. The results of the profiling are below, for Gaussian, Gradient, Non-max Suppression, Thresholding, and Hysteresis.

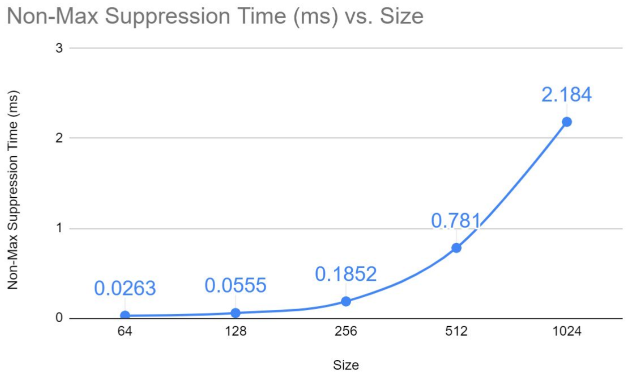
Here we see that the time for the Gaussian portion of the code grows closer to linear than it does quadratic. This is likely due to the quickness of the algorithm and thus overhead is a relatively high portion of the total time.



The next phase is for calculating the Gradient, and this is much closer to quadratic as we would expect. Gradient calculation takes on average less than 10% of the total runtime. One would expect this to take twice the time than the Gaussian stage, but the extra expensive magnitude and angle edge calculations make it longer.

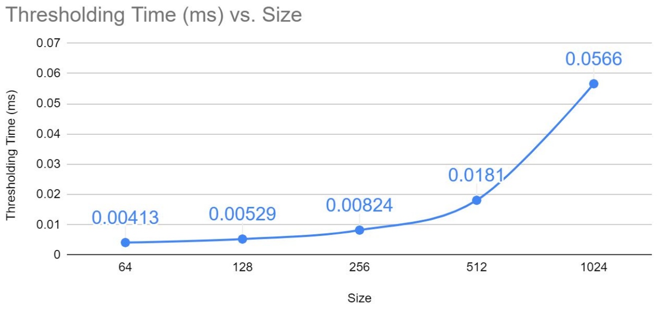


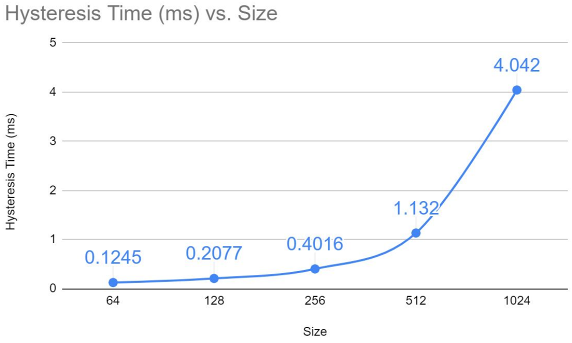
The third step is the Non-Max Suppression which is the second longest phase and the longest that we have seen so far. It took on average about 30% of the total runtime. Non-Max Suppression is not computationally heavy, but requires creating and



copying an extra array of the image to resolve data dependencies mentioned in the GPU section. The extra memory latency of transferring data from GPU to CPU and back makes this stage of Canny execute longer than expected.

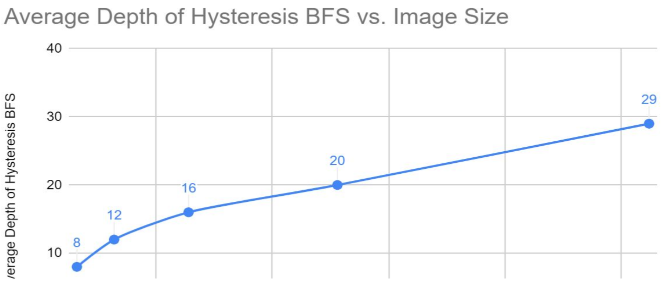
We then move into the thresholding phase where the growth rate appears roughly linear until N = 512 and then the growth rate slightly increases. Overall this is very close to linear and again is likely caused by it being such a short time that the overhead is dominating the runtime. If we were running on 2K, 4K, 8K, or higher images then we would probably start to see more of a parabolic trajectory to the timing.

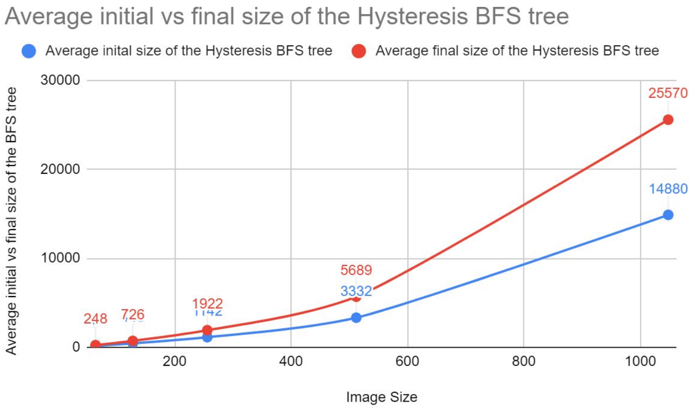


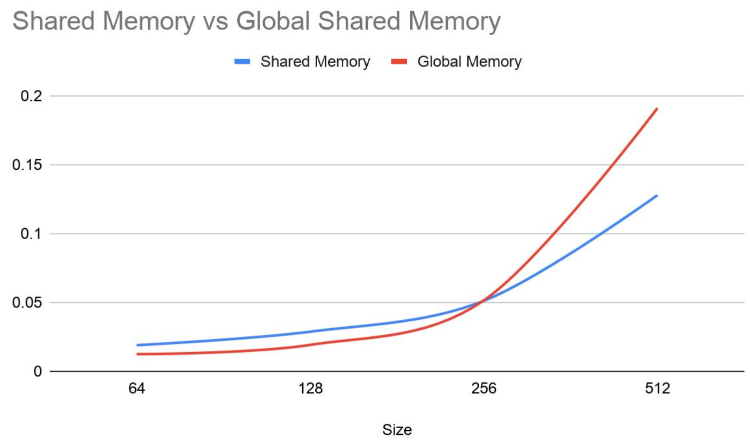


The final stage of Canny Edge Detection, Hysteresis, is also the longest, accounting for between 40% and 60% of the total time depending upon the input size. We also see that this follows a quadratic growth as it should.

We also calculated the average depth of the hysteresis BFS as well as the total average initial and final BFS trees for the Hysteresis and these are shown to the side. The average depth of the hysteresis is increasing logarithmically. As a consequence, the hysteresis kernel will have to be fired a non-constant amount of times as the image varies unlike the other stages in Canny. The result is that the hysteresis portion should increase as a function of $N^2 log(N)$. Unfortunately, we do not have enough data points to see this clearly in the results.







Finally, a graph to the right shows a comparison between using shared and global memory in convolution operations. In global memory implementation with kernel size K and 3 convolutions, the amount of data fetching from global memory is $3 * K^2$. In a shared memory implementation the number of data fetches from shared memory is $3 * K^2$ and from global memory is 2. Accordingly, if global memory access is a 10 times slower. The speedup in memory access per pixel is $\frac{global}{shared} = \frac{10*3*K^2}{3*K^2+10*2}$. While this result does not manifest in lower resolution sizes, one can see a jump in speed at a higher resolution of 512x512.

Future Work/Research

Our work focused on improving the latency from input to output. We also were limited to just one GPU with 2,560 CUDA cores, so we were already using every core available with the 64x64 sized images. But if we consider something like a supercomputer we might be dealing with many more CUDA cores and there are efficiencies that we could implement to exploit this. First we could pipeline the algorithm so that there is one image in each phase. However, since some phases take much longer than others, this would only be able to reduce the time per image to be equal to the longest phase, but not to the average phase length. We could also implement Otsu's method as an extra feature in the thresholding phase. Another area for research would be to explore other more efficient hysteresis implementations as this was the longest part of the algorithm.

Works Cited

Emrani, Zahra, et al. "A New Parallel Approach for Accelerating the GPU-Based Execution of Edge Detection Algorithms." *Journal of Medical Signals and Sensors*, Medknow Publications & Media Pvt Ltd, 2017, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5394804/.

Chen, Xiaoming, et al. "Optimizing Memory Efficiency for Convolution Kernels OnKepler GPUs". *The 54th Annual Design Automation Conference 2017*, June 2017. https://arxiv.org/abs/1705.10591

Size	CPU FPS (16 cores)
64	4807
128	1683
256	705
512	173
1024	41

Size	GPU FPS (P5000)
64	5688
128	3649
256	1367
512	446
1024	140