ECE588: Parallelization of the Canny Edge Detection Algorithm using CUDA (March 2017)

Dan Collins, Anil R. Peddolla, and Alex G. Varvel

***Abstract* — Edge Detection is a topic of particular importance for computer vision and in general, image processing. Detection of edges allows distinct characteristics of 3-D space to be identified with reduced storage and processing overhead. Processing speed is critical in popular technical disciplines which require real-time image analysis, such as automotive driving assistance systems and facial recognition. The Canny Edge Detection algorithm is unique within the edge detection field because it drastically reduces the amount of data that needs to be processed relative to other first-order image processing algorithms. This report gives an overview of the Canny Edge Detection algorithm and the CUDA programming model. Serial CPU and parallel GPGPU implementations of this algorithm are then discussed, including the parallelization methodology. Finally, results are analyzed for both performance and correctness. A parallel speedup proportional to the size of the image is realized.**

I. INTRODUCTION

EDGE detection is a mathematical method of determining the "edges" within an image. An "edge" is defined as a point which the brightness of the image has discontinuities. Sharp changes in pixel intensity allow the edge detection algorithm to map an outline of the image being observed, and these edges can correspond to changes in depth, orientation, materials, or lighting. Mapping the edges of an image is a non-trivial task, as many false edges can be mapped, or segments of a real edge can be left off if the change is not drastic enough for the algorithm to pick up.

The algorithm implemented in this paper is the Canny Edge Detection algorithm. This algorithm is a first-order image processing algorithm that was developed by John F. Canny in 1986 and follows a 5-step detection algorithm: Apply a Gaussian filter, find the intensity gradient, apply non-maximum suppression, apply a double threshold, then track the edges using hysteresis. The size of the Gaussian filter and the double threshold can be manipulated in order to affect both the computation time and the effectiveness of the algorithm. Based on the image being processed, different combinations of the two may yield more "accurate" results.

Edge detection using a program that runs serially causes the program to run very slow, especially as the number of pixels increases because the amount of work is directly proportional to the number of pixels in the image. In real-time image processing analysis, a serial implementation will be much too slow. Edge detection can be parallelized in order to drastically speed up computation time, so this report details how the algorithm can be parallelized using the parallel computing platform CUDA. CUDA was chosen as the parallelization platform because it has been designed/optimized for graphics and image processing falls into that category. Additionally, this is a very common platform in industry, so the results of this report can be used as a baseline for future studies.

II. CANNY EDGE DETECTION ALGORITHM OVERVIEW

Before discussing how the algorithm can be optimized, it is important to first understand how the algorithm is applied to the image. The Canny Edge detector algorithm contains five filters that are serially applied to the image on a "per-pixel" basis. Since pixels have a red, green, and blue component, each field is manipulated independently. Some of the filters require information to be exchanged amongst the pixels while other filters only apply to each pixel individually. The exchange of information between pixels can significantly hinder performance when implementing a parallelized version of the edge detection algorithm, so intelligent passing of information is of utmost importance.

A. Gaussian Blur

The first filter needed in the Canny Edge Detection algorithm is the Gaussian blur. The purpose of this filter is to slightly blur the image and reduce the noise in it. Reducing the noise in the image allows for the actual, distinct edges to be detected while avoiding spurious noise which could have been detected as a false edge. This filter works by applying a square "convolution kernel" to each pixel over the image. The "convolution kernel" is a square matrix of scalars following a Gaussian distribution that gets applied to the center pixel and the surrounding pixels.

Once the kernel is applied, the sum of the scalars multiplied by the values of the pixels within the kernel are divided by the sum of the scalars in the convolution kernel and that becomes the new value for the pixel being blurred. For example, if a 5x5 convolution kernel is applied to a pixel in the center of the image, the new red, green, and blue value for that pixel will be the sum of the red, green, and blue values multiplied by the convolution kernel divided by the sum of the values in the "convolution kernel," respectively.

What makes this filter Gaussian, is the "convolution kernel" does not follow an average distribution, where all scalars contribute equally, but the scalars within the matrix represent a Gaussian distribution, meaning values of pixels closer to the pixel being blurred contribute more. The size of the convolution kernel determines the amount that the image gets blurred. The larger the kernel, the more the image is blurred. See figure 1 below for a sample calculation of the process:

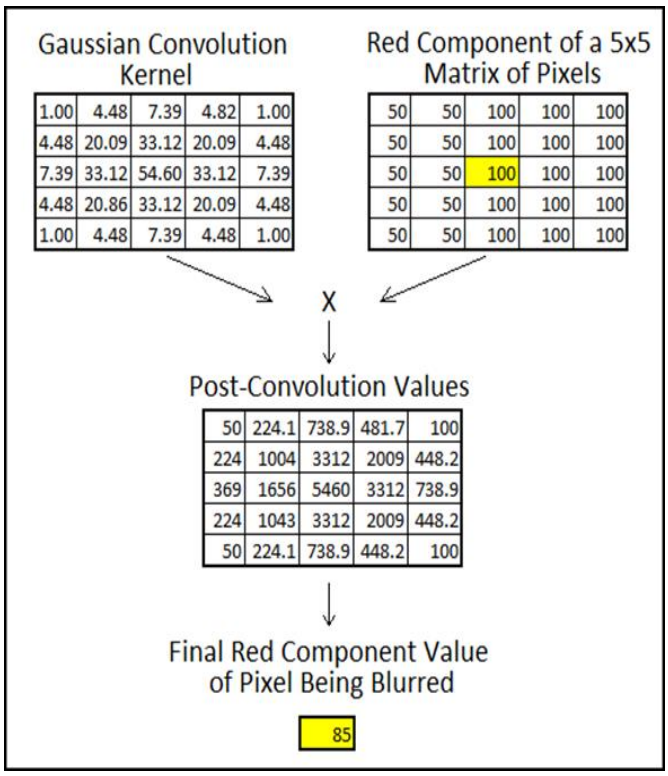

Fig. 1: Convolution Kernel Example

B. *Intensity Gradient*

The input to the intensity gradient step is the blurred image produced by the Gaussian Blur step above. Similar to the Gaussian blur, this filter also needs to communicate with the pixels around it. This filter takes the first derivative of the contrasts between color channels across the image in the horizontal and vertical directions in order to determine the magnitude and direction of each possible edge. The horizontal and vertical gradients are known as the "deltaX" and "deltaY" arrays. These derivatives are converted from a red, green, and blue three channel gradient to single, one channel gradient by converting the gradients to grey scale. From this, the edge gradient is determined by finding the hypotenuse of deltaX and deltaY. The magnitude of the gradient can be determined using the following formula: sqrt(deltaX2 + deltaY2). Edge direction is evaluated as "alpha," which is the ratio of deltaX and deltaY. "Alpha" is used in the following filter.

C. *Non-Maximum Suppression*

Since the image was blurred in the Gaussian Blur step, edges detected by the intensity gradient filter may appear thicker than they actually are. Non-Maximum Suppression thins the edges that were produced by the horizontal and vertical gradients generated by the Intensity Gradient filter. Because the image was blurred, the values of the pixels towards the outside of the edge have less magnitude than the pixels in the middle of the edge. The non-maximum suppression filter takes the pixels along an edge and suppresses the values that are not at the maximum value to thin the edges. This is accomplished by first determining the direction of the edge using "alpha." The pixels perpendicular to the edge denoted by "alpha" are compared and all the gradient values are suppressed to 0 except the local maxima, which indicates location with the sharpest change in intensity value.

D. *Double Threshold*

The previous three filters were all applied to the image to detect the true edges, but along the edges, there still may be noise or spurious edges. The double threshold filter applies a high and a low threshold across the image to determine the strong and weak edges. If a pixel's value is greater than the high threshold, it is marked as a strong edge, if a pixel's value is less than the high threshold but greater than the low threshold, it is marked as a weak edge, and lastly, if a pixel's value is less than the low threshold, the pixel is suppressed completely. This filter is used to eliminate the last bit of noise.

E. *Edge Tracking Hysteresis*

In addition to detecting the edges, all four of the previous filters have filtered noise that is present in the image. Because there may be some edges that were filtered too severely (for example, edge information could have been blurred away in the image during the Gaussian blur), the edge tracking hysteresis filter makes one final pass over the image and connects edges that should have been connected. Using the strong edges that were determined by the double threshold filter, the edge tracking hysteresis filter passes over the image using a 3x3 matrix that connects weak edges adjacent to strong edges. Doing so re-establishes edges that should have been detected but were dropped by aggressive filtering. After this filter finishes, the edges in the image have been properly detected.

III. CUDA Overview

There are many tools (libraries, devices, etc.) available to parallelize the filters which make up the Canny Edge Detection algorithm, but for this project, NVIDIA's CUDA API was used. GPUs are well suited for image processing and filtering because much of the work approaches being embarrassingly parallel. The CUDA programming model abstracts away the GPU device and presents the programmer with a co-processing capable of handling coarse-grained tasks. This enables programmers to write thread-level code that executes in many independent, concurrent threads. This model is perfect for data-parallel processing, such as applying the same filter to many pixels simultaneously.

The code was written in C++ with the CUDA extension and compiled using NVIDIA's proprietary complier, NVCC. The problem space maps well to GPUs, so it was anticipated that a large speedup between the serial and parallel implementation would be realized by using CUDA.

In order to achieve the data-parallelism provided by the CUDA API, each filter had to be realized in the constructs provided by CUDA. NVIDIA graphics cards have many levels of abstraction that can be exploited to achieve massive amounts of parallelism. Any task that is to be parallelized can be split into a combination of grids, blocks, and threads. These constructs will be discussed in more detail in section 4.C:

Parallel Algorithm Implementation and for further information on how these constructs are arranged in hardware, please consult reference number 5 in section 8: References.

IV. Implementation

In order to determine the speedup achieved due to parallelization, a baseline for timing had to be established. A standard, single thread implementation was written in C++11. This algorithm was then parallelized and the speedup was analyzed. The following sections detail this process.

A. Execution Environment

A standardized environment which utilized the FAB computers provided by PSU was used to develop the edge detection code. The FAB computers have the CUDA toolchain pre-installed and are equipped with at least one NVIDIA Quadro K620. In order to manipulate independent pixels, the Magick++ image processing library was used. Magick++ allows images of standard encodings such as .jpg, .png, or .bmp to be imported and decomposed into a flat buffer of pixels. Lastly, GitHub was used for version control and code sharing amongst the team. Please consult reference 6 in Section 8: References for the link to the GitHub repository used for this project.

B. Serial Algorithm Implementation

The serial implantation of each filter used a single for loop to iterate over the flat buffer of pixels provided by Magick++. Each filter detailed in section 2: Canny Edge Detection Algorithm Overview was implemented in a separate function and each function used a nested for-loop to iterate serially through the pixel buffer and transform the pixels one at a time. Implementing these filters serially made the fine-tuning and debugging of the algorithms much smoother since serial debugging is much easier than parallel debugging. However, this was a limited implementation because the execution time of the images scales linearly with image size. A faster image processing approach was needed to simulate real-time analysis.

C. Parallel Algorithm Implementation

Parallelism in CUDA allows for for-loops to be "unrolled" by mapping pixels to threads. If a thread can be mapped to each individual pixel, then it is as if a for-loop has applied the same filters to each pixel; the difference is the computation across all pixels happens in parallel.

The Canny Edge Detection problem can utilize domain-decomposition to partition pixels or groups of pixels to each thread. As discussed above, CUDA abstracts the GPU hardware resources into grids of thread blocks, each of which can be referenced using a global thread ID. The thread IDs consist of one, two, or three dimensional coordinates. Since the Canny Edge detection filters did not require complex mapping, one dimensional grids, blocks, and threads were used. The version of CUDA used (and thus the underlying hardware) allowed for a maximum of 1024 threads in a block. The size of the image was divided by 1024 to obtain the number of threads blocks, and all of these blocks were within one grid. Effectively one thread was launched per pixel during each kernel execution, each doing the work of the innermost loop of the serial implementation. A separate kernel was created for each step of the algorithm, with the exception of the hysteresis step.

To allow the Edge Tracking Hysteresis filter to be parallelized using CUDA, the serial implementation was broken into two parts (due to a data dependency), each of which was implemented as a separate CUDA kernel. The first CUDA kernel goes through the image and pushes any pixel that is above the high threshold value to its max value (a strong edge is equivalent to the max pixel value, or "white"). This step also creates a bitmask of these strong edges which need to be revisited for later processing. The second CUDA kernel looks at each pixel that was previously found to be a strong edge and applies a low threshold to its eight immediate neighbors. If any of these neighbors are above the low threshold, they are also marked as strong edges (effectively connecting the weak edges). This resulted in slightly different edge detection which can be seen in section 5: Results.

Within the memory hierarchy of an NVIDIA graphics card, "Shared Memory" can be used to pass information quickly between threads. This memory is local to each block, thus "Shared Memory" cannot be shared between blocks without explicit reads and writes to global memory (which is available to all threads, blocks, and grids). The convolution kernel used in the Gaussian blur and any other kernels or variables that were applied to the whole image were also moved to Shared Memory, since it is significantly faster than using global memory.

The last optimization that was used in the parallel implementation of the Canny Edge Detection algorithm was streams. Streams allow for task level synchronization by forcing each CUDA kernel to execute tasks (kernels and memory operations) serially within a stream. All of the filters used in the algorithm were chained together using a stream, so that no two filters were being applied at the same time. This allowed us to ensure correctness and consistency across execution cycles.

V. Results

Below is a subset of the images tested in this project. Figure 2 is the reference image that we used to compare the accuracy of our edge detection algorithm to that of one used in the field. Figure 3 is the output of the reference edge detection algorithm. To validate the accuracy of our edge detection algorithm, we took post-edge detection images from benchmark suites and qualitatively inspected them for similarity. A qualitative inspection was used instead of a binary differential calculation because the output wouldn't be representative of the actual accuracy of the algorithm. There are too many insignificant and minute differences in implementations which could result in invalid findings. Figures 4 and 5 were the output of our serial and parallel edge detection algorithms, respectively. Figure 3 was compared to figures 4 and 5 to determine the accuracy of the edge detection algorithm implemented in this project.

We were satisfied with the results of our implementation of this algorithm as every important edge was detected by our algorithm. Upon closer inspection, it can be seen that our algorithm actually detects edges with a higher degree of accuracy than the reference algorithm. The edges presented by the pipes on the right side of the image are detected by our algorithm, but not by the reference edge detection algorithm. This specific area was highlighted using a red circle in figures 3, 4, and 5. Based on these findings, we concluded that our algorithms were working, and working well.

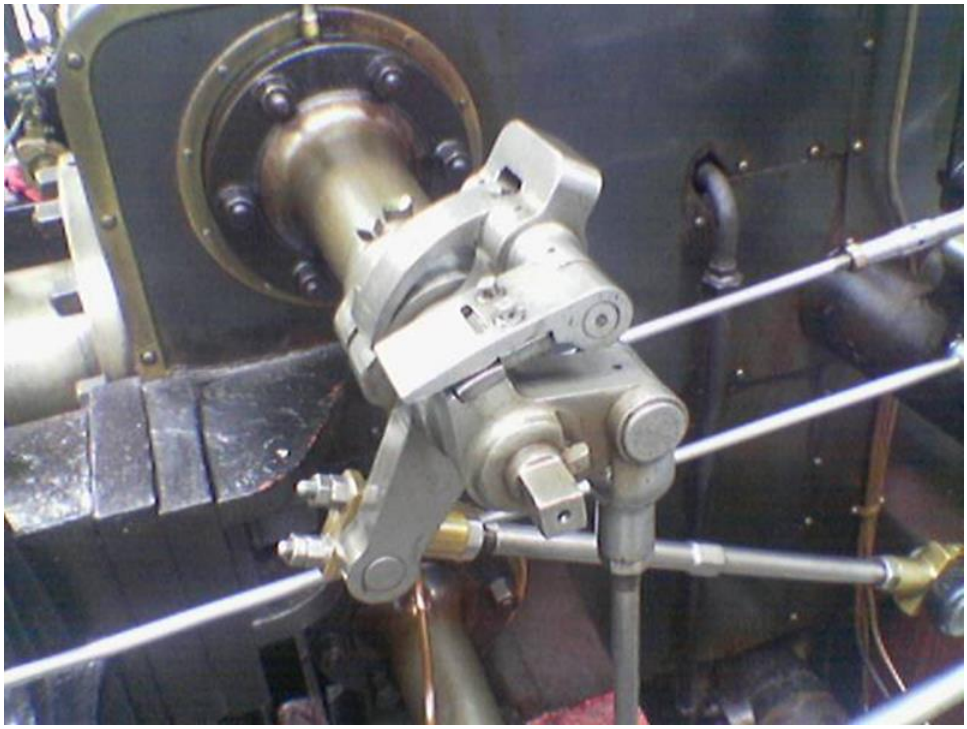

Fig. 2: Reference Image

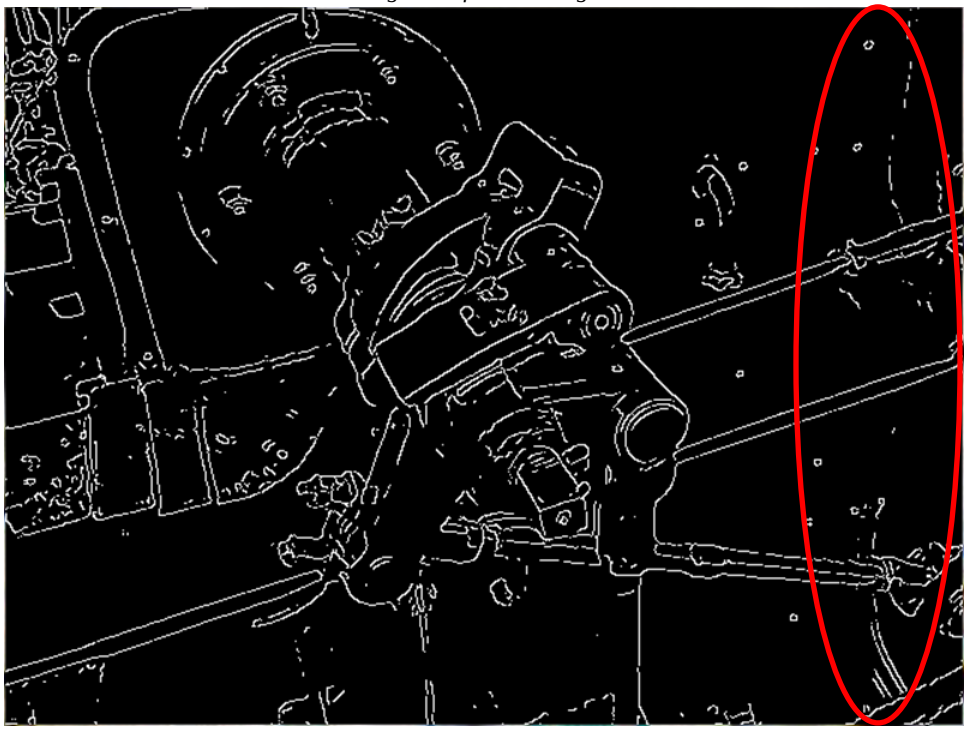

Fig. 3: Reference Edge Detection

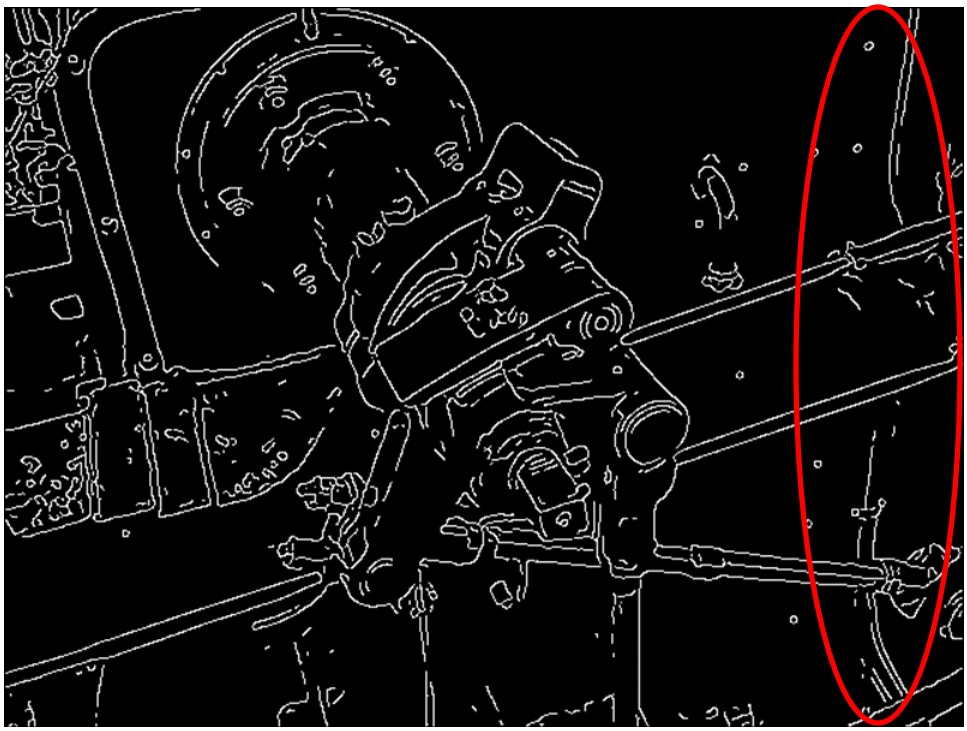

Fig. 5: Edge Detection via Serial Implementation

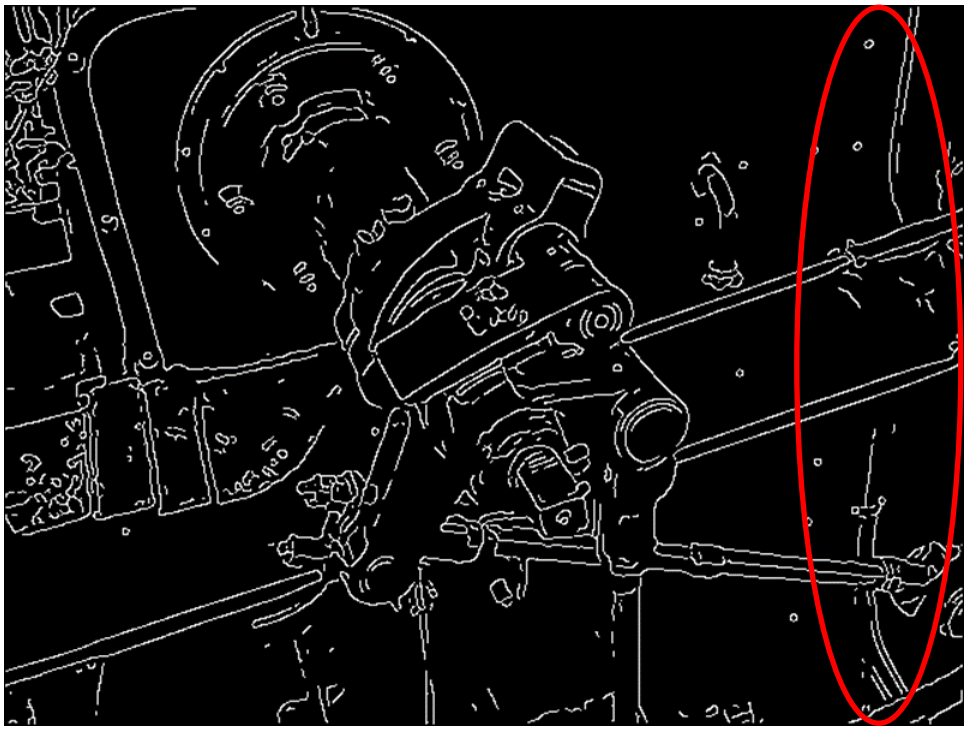

Fig. 4: Edge Detection via Parallel Implementation

VI. Analysis

Two analyses were performed in order to test the effectiveness of parallelization: Image size manipulation and convolution kernel size manipulation. Thread count was not manipulated because we mapped each thread to each pixel of the image, so as the image size increased, so did the number of threads. For the image size manipulation, a very small image (70x70), a medium sized image (640x480), a 1080p image (1920x1080), a 4k image (4096x2160), and an 8k image (7680x4320) were all filtered using the serial and parallel implementation of the code. The times and relative speedups between the serial and parallel implementations were then recorded and calculated, respectively. For the convolution kernel size manipulation, the size of the kernel was changed from 3x3, 5x5, 7x7, 15x15, and 31x31. The 15x15 and 31x31 sized convolution kernels are too large to provide any relevant edge detection as they blur the image too much, but the speedup when computing that much data was still worthwhile to test. The results from each analysis can be seen in the following sections.

A. *Image Size Manipulation Result*

Tables 1 and 2 and figures 6 and 7 show the results of the image size analysis. The conclusions which can be drawn from this analysis will be discussed in section 7: Conclusions.

B. *Convolution Kernel Size Manipulation Results*

Tables 3 and 4 and figures 8 and 9 show the results of the convolution kernel size manipulation analysis. The conclusions which can be drawn from this analysis will be discussed in section 7: Conclusions.

Table 1: Image Size Manipulation Data

Image Size Manipulation		
Image Resolution	Serial Speed (s)	Parallel Speed (s)
70 x 70	0.003811	0.086464
640 x 480	0.270741	0.10651
1920 x 1080	1.642513	0.16379
4096 x 2160	7.332309	0.405638
7680 x 4320	26.205247	1.233142

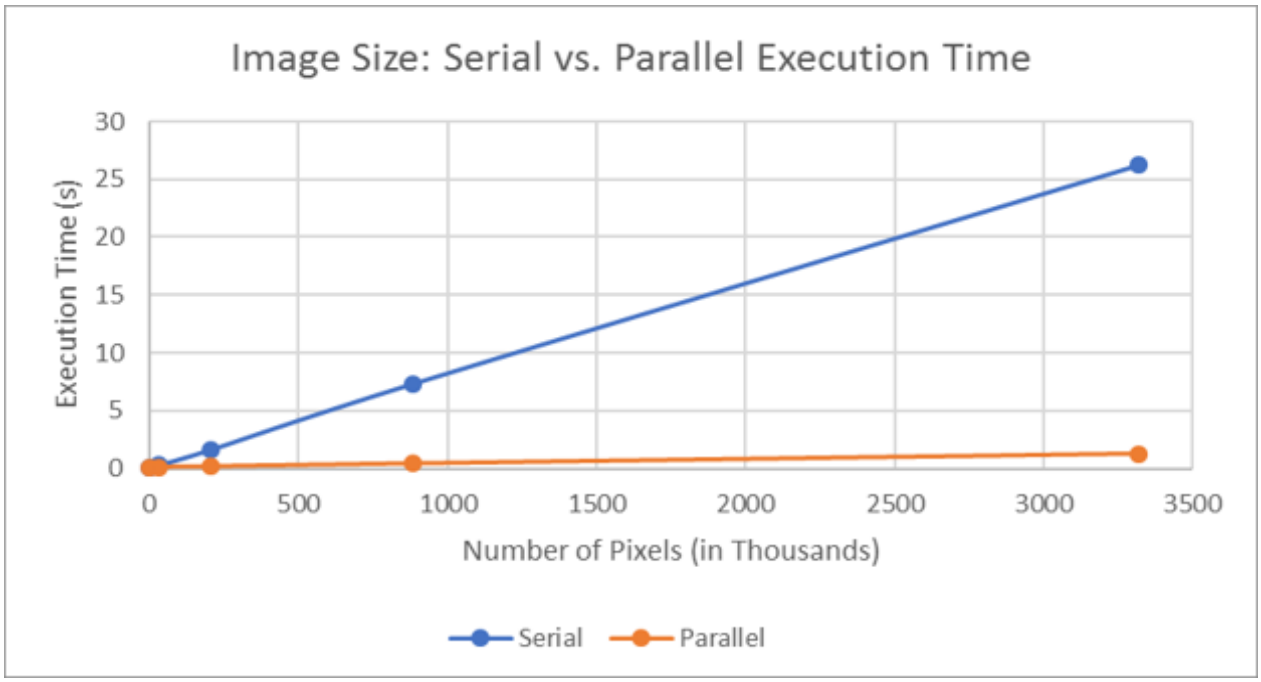

Fig. 6: Image Size vs. Execution Time

Table 2: Image Size Manipulation Speedup

Image Size Manipulation	
Image Resolution	Parallel Speedup
70 x 70	-22.6880084
640 x 480	2.541930335
1920 x 1080	10.02816411
4096 x 2160	18.0759914
7680 x 4320	21.25079431

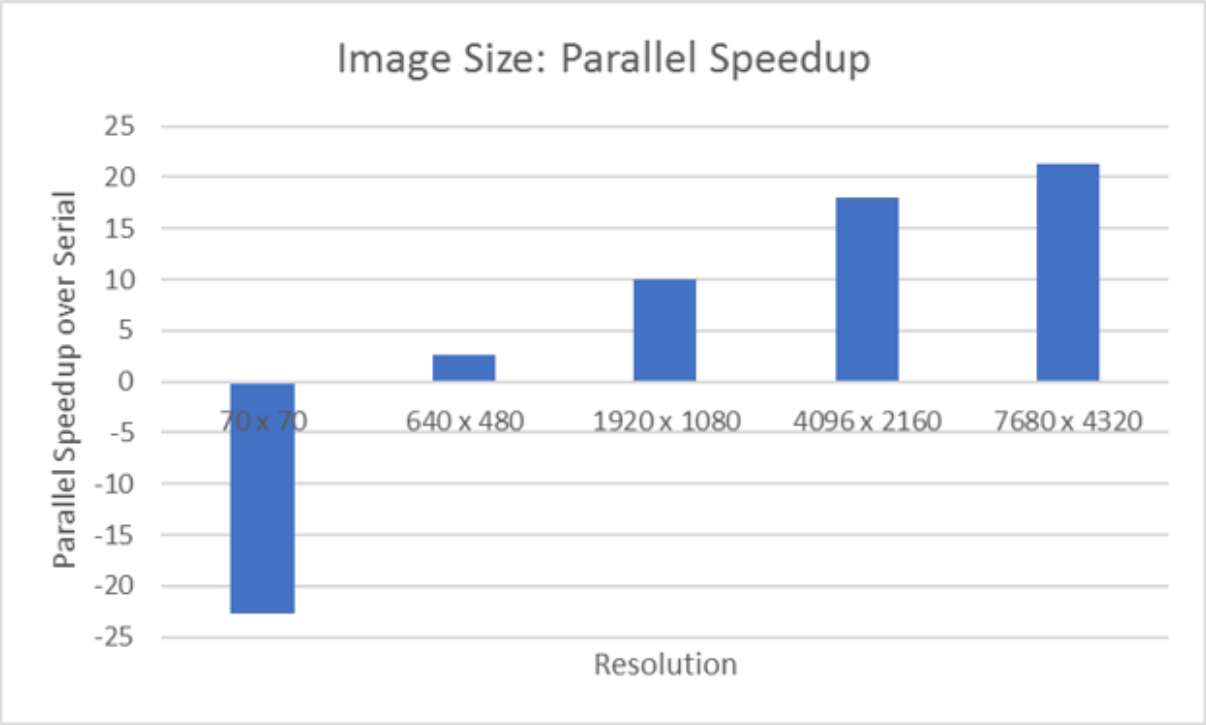

Fig. 7: Image Size Manipulation Speedup

.

Table 3: Convolution Kernel Size Manipulation Data

Convolution Kernel Size Manipulation		
Kernel Size	**Serial Speed (s)**	**Parallel Speed (s)**
3	0.439012	0.118802
5	0.937175	0.136428
7	1.642513	0.16379
15	8.013535	0.418164
31	33.578706	1.28104

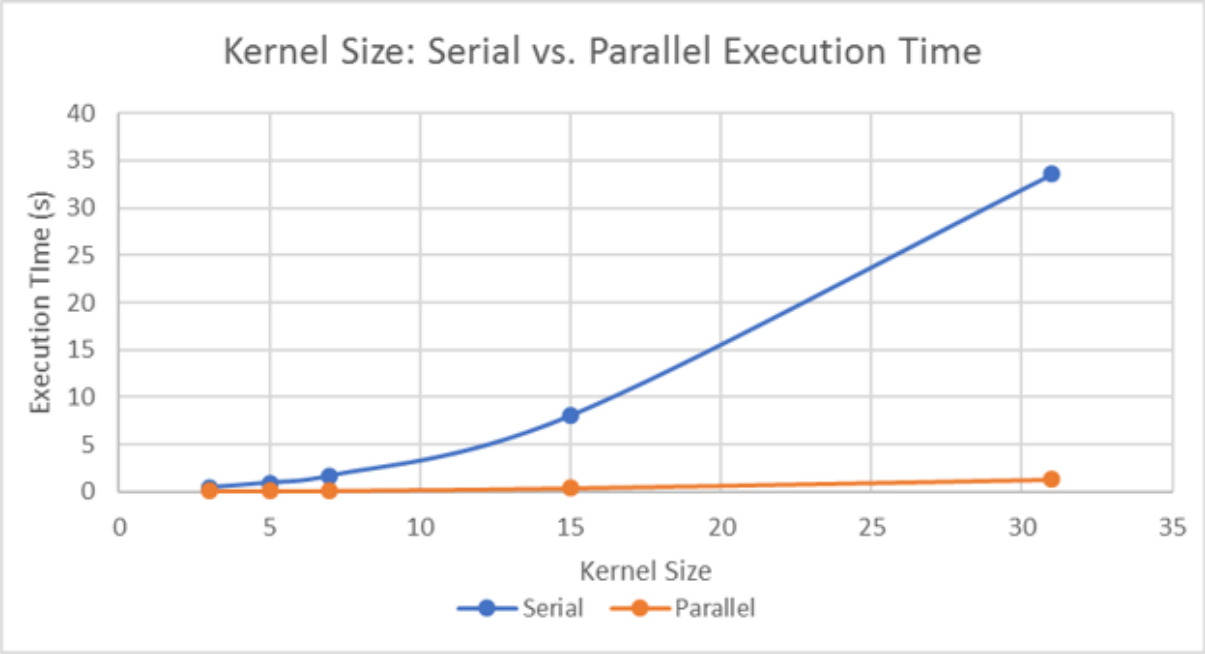

Fig. 8: Convolution Kernel Size vs. Execution Time

Table 4: Convolution Kernel Size Speedup

Convolution Kernel Size Manipulation	
Kernel Size	Parallel Speedup
3	3.695324995
5	6.869374322
7	10.02816411
15	19.16361762
31	26.21206676

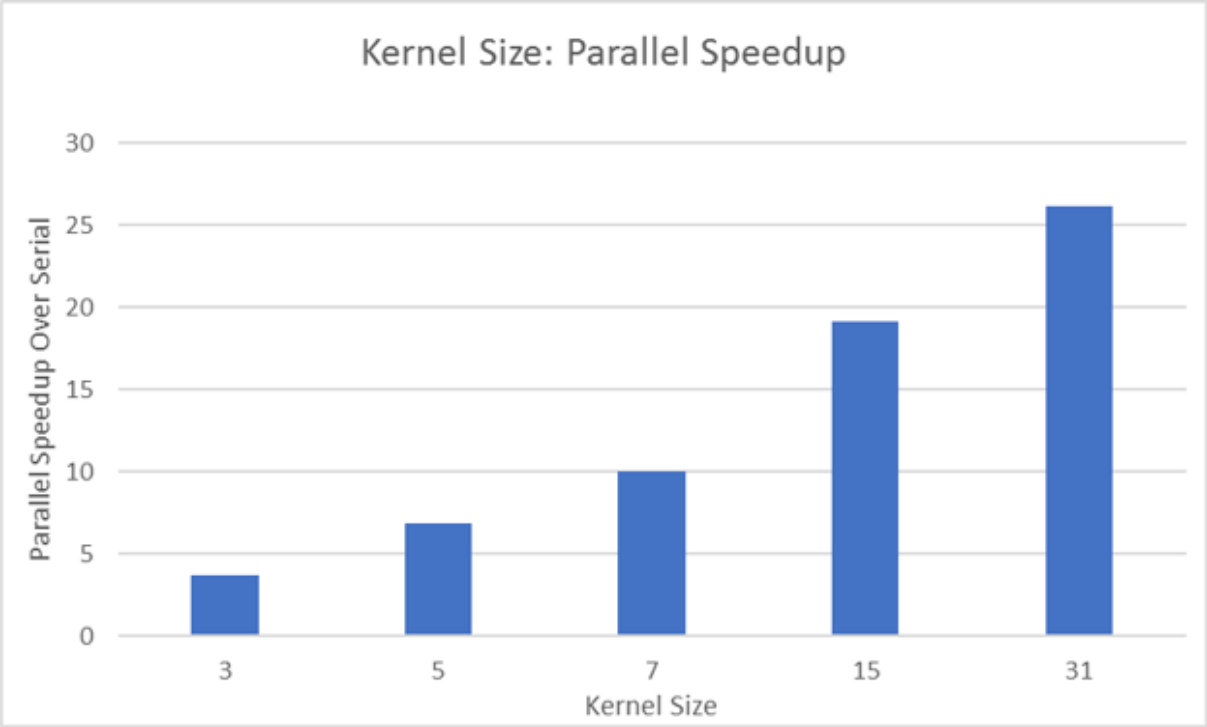

Fig. 9: Convolution Kernel Size Manipulation Speedup

VII. Conclusion

Based on the information provided in the results and analysis sections, we feel as though this project was a great success. Section 5: Results detailed the accuracy of our edge detection algorithm: the output image from our edge detector had less noise and more detail than the reference edge detector so we are very satisfied with the quality of our edge detection implementation.

On an even more impressive note, the speedup we saw when we parallelized the algorithm using CUDA was incredible. Tables 2 and 4 show that at a minimum, the parallelization generated a speedup of about 2.5x when in regards to image size, and almost 4x in regards to changing the convolution kernel size. At the most extreme, the image size manipulation showed a speed up of 21x and the convolution kernel size manipulation showed a speed up of 26x. As can be seen in figures 6 and 8, if larger images and kernel sizes were tested, this trend would continue as well because the serial implementation increases linearly with the size of the image/kernel whereas the parallel implementation speed increased at a much lower rate. The only time this did not hold true was for the smallest image of 70x70 pixels. This image was small enough that the serial implementation was faster due to the overhead involved with device memory allocation, data transfer, and PCIe bus transmission between the CPU and GPU.

In hindsight, it was wise to use the standardized development environment because it allowed us to reproduce behavior in a consistent manner between team members. Since there is a wide variety of performance and feature deltas between different generations of NVIDIA cards, it could have gotten complicated if our timings and execution paths were dependent on the hardware the program was running on. Using GitHub as a revision control and collaboration platform was also beneficial, because it allowed team members to make independent progress, share and audit progress, and combine deliverables.

We felt as though this project was informative and edifying. The execution time of the parallel implementation remained relatively flat compared to the serial implementation, which increased linearly with the number of pixels to be processed. This is a best-case scenario for parallel speedup: the execution time is nearly independent of the number of iterations of processing that need to be done on the pixels. It shows that the Canny Edge Detection algorithm is highly parallelizable, and that GPUs, if enough hardware resources are present, are able to do this work concurrently.

References

[1] J. Canny. "A computational approach to edge detection." IEEE Transactions on pattern analysis and machine intelligence 6 (1986): 679-698.

[2] K. Ogawa, I. Yasuaki, and K. Nakano. "Efficient Canny edge detection using a GPU." Networking and Computing (ICNC), 2010 First International Conference on IEEE, 2010.

[3] Y. Luo, and R. Duraiswami. "Canny edge detection on NVIDIA CUDA." Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on. IEEE, 2008.

[4] L. H. A. Lourenço, D. Weingaertner, and E. Todt, "Efficient Implementation of Canny Edge Detection Filter for ITK Using CUDA," 2012 13th Symposium on Computer Systems, Petropolis, 2012, pp. 33-40. doi: 10.1109/WSCAD-SSC.2012.21

[5] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, “NVIDIA Tesla: A Unified Graphics and Computing Architectre,” IEEE Micro 2008, pp. 39-55

[6] To access the GitHub Repository used for this project, please visit the link below: https://github.com/kinap/canny-edge-dectector