Canny Edge Detection on GPU using CUDA

Matthew Horvath Jr.
Electrical and Computer Engineering Department
Oakland University
Rochester, MI, USA
mhorvath@oakland.edu

Michael Bowers
Electrical and Computer Engineering Department
Oakland University
Rochester, MI, USA
mkbowers@oakland.edu

Shadi Alawneh, Ph.D., (Senior Member, IEEE)
Electrical and Computer Engineering Department
Oakland University
Rochester, MI, USA
shadialawneh@oakland.edu

2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC) | 979-8-3503-3286-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/CCWC57344.2023.10099273

***Abstract* - Edge detection is a crucial step in many of today's computer vision applications. Canny edge detection in particular involves several steps to achieve real-time results. Many systems currently deployed leverage the compute capability that a graphics processing unit (GPU) can achieve. This paper covers the implementation and testing of a Canny edge detection algorithm using CUDA C. The results cover a comparison of the naive implementation in sequential C, a parallelized implementation using OneAPI Threading Building Blocks (TBB), and a tiled, shared memory approach using CUDA C. A comparison between the NVIDIA GTX 1060 and NVIDIA RTX 3090 are also performed. The CUDA C implementation shows an improvement of up to 100 times that over the naive sequential implementation for an RGB image at 4k resolution, and an improvement of 10 times when compared to the TBB approach. Additionally, the RTX 3090 showed roughly a speed up of 1.5 times that of the GTX 1060, demonstrating the advances made between the generations of GPUs. These results overall show the benefits of using a GPU accelerated approach to edge detection, with further improvements left to achieve.**

Keywords - CUDA, Kernel, Compute, Edge Detection, Parallelism, Shared Memory, Tiling, Real-Time

I. INTRODUCTION

The Canny Edge Detector is an edge detection method used to highlight and isolate edges within an image, and is widely used as a pre-processing tool in computer vision [1]. Canny Edge Detection follows a multi-stage algorithm: Stage 1 consists of taking the RGB image frame and converting it to grayscale. Stage 2 performs a 2D Gaussian blur convolution on the grayscale image to reduce noise. In Stage 3, the Sobel operation is performed on an image, which consists of convolving the image with 2D Sobel kernels to find the intensity of image gradients. Two convolutions are performed, with one to highlight derivatives in gradients in the horizontal X direction and another for highlighting derivatives in the vertical Y direction. The absolute magnitude and the direction of the gradient for each pixel are found from the X and Y gradient components. In Stage 4, non-maximal suppression is performed, which will suppress pixels whose absolute magnitudes are not the local maximum with respect to its immediate neighboring pixels along the direction of the gradient, which visually thins edges. Stage 5 consists of double thresholding, which will classify edges as strong, weak, or non-edges according to a high and low threshold. This is complemented by a hysteresis stage which will reclassify weak edge pixels as strong edge pixels if they are neighbored by at least one strong edge pixel.

One notable property of the Canny edge detection algorithm is that within a given algorithm stage, each pixel can be calculated independently. Thus, the Canny edge detection algorithm can highly benefit from parallel computation. Cheikh [2] implements the canny-edge detector utilizing various standards such as OpenMP and CUDA to employ loop-level parallelism on CPU and GPU architectures, respectively. Domain decomposition can also be utilized by splitting the input image into smaller tiles and distributing the computation of individual tiles across a multi-processing architecture. Rahamneh [3] utilizes a heterogeneous CPU-FPGA architecture and load balancing to queue image tiles so each processor handles separate image tiles, achieving speedups compared to CPU-only and FPGA-only implementations.

GPUs are often leveraged in computer vision and other high-throughput applications for their ability to buffer large amounts of data (i.e. video frames) and compute in parallel by distributing data elements across numerous GPU cores. NVIDIA GPUs utilize the CUDA programming model, which provides a compute platform for developing efficient parallel applications for general-purpose computing on GPU hardware. CUDA utilizes compute kernels as routines that are executed in parallel within a group of threads. CUDA's programming model defines the grouping of threads into multi-dimensioned thread blocks, which are further organized into grids. Thread blocking aids in data mapping and enables barrier synchronization and the use of shared memory between threads within the same thread block.

Prior implementations of the Canny algorithm on CUDA-enabled systems include an implementation by Luo and Duraiswami [4], which features a significant speedup compared to CPU-based OpenCV and Matlab implementations. Shared

979-8-3503-3286-5/23/$31.00 ©2023 IEEE

memory aprons were utilized for the convolutional stages during the non-maximal suppression stage. Major differences are seen in the implementation of the hysteresis thresholding heuristic implementation, as blob analysis for weakly-classified edges was extended beyond a typical view window in Canny edge detection, which resulted in the hysteresis component of the implementation occupying 75% of the total processing time. Traditional implementations of this stage involve reclassifying weak neighbors based on the strength of its immediate neighbors.

II. Methodology

A. Implementation

Three implementations were developed and tested against each other: a sequential implementation was developed in C, where each pixel was iterated over and calculated individually. A multithreaded CPU version was implemented in C++ using TBB [5], and was achieved by wrapping the calculation for an individual pixel within a tbb: parallel for, which was employed in all calculation stages where the order of iteration did not affect the final result. The GPU implementation was developed using CUDA C and utilizes loop-level parallelism, as a CUDA Kernel was developed for each computation stage.

Fig 1. RGB Image of Female Red-Winged Blackbird

B. Grayscale-to-RGB Kernel

In the sequential implementation, each pixel is iterated over the total image size to compute the grayscale value for each pixel using the RGB luminosity method. In the GPU implementation, the input color image is copied to the GPU global memory, and each GPU thread handles the luminosity calculation for each element so long as the calculated indices for each point is within the image boundaries. These grayscale results are copied to a pre-allocated global array within the GPU global memory. Fig. 2 illustrates the pseudocode of the RGB-to-Grayscale CUDA kernel.

```
1: Get thread indices for X and Y relative to the image
2: If thread indices are within image X and Y boundaries
3:         Get thread index and grab RGB pixel value
4:         Compute the RGB to Grayscale converted pixel
5:         Output Grayscale pixel to global memory
6: End
```

Fig 2. RGB to Grayscale Pseudocode

C. Gaussian Blur Kernel

In the second stage, a blur is applied to the grayscale image result via a convolution operation. A gaussian distribution is used for the convolutional mask, which is used to reduce image noise.

$$B = 1/159 * \begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix} * A$$

EQUATION I. Gaussian Blur Kernel

```
1: Get load thread indices to load shared tile
2: Get output tile thread indices to output
3: If load indices are within image X and Y boundaries
4:         load in image pixel value to input tile
5: Else
6:         load in zero to input tile
7: Wait for barrier synchronization within the block
8: If thread index is within output tile X and Y
9:         Perform flipped mask convolution
10:        If thread index is within output image X and Y boundaries
11:                Output blurred pixel to memory
12:        End
13: End
```

Fig 3. Gaussian Blur Pseudocode

In the GPU implementation, shared memory was used to load the input elements necessary to perform the convolution calculation for each output pixel. Each shared memory tile is sized 16 by 16, which is identical to the block size. Thus, each member thread of each block participates in the loading of an input pixel from global memory into shared memory. For thread blocks that extend past the image boundaries, any elements that extend beyond the image boundaries will be zero-padded.

Fig 4. Gaussian blur applied to grayscale image

Each thread within the output tile range of each thread block participates in the calculation of an output pixel, and performs a flipped convolution operation with respect to the unique 5x5 section of input elements that are loaded in from the shared memory tile. Threads that participate in calculating an output pixel will commit the result into a pre-allocated global array within the GPU global memory. Fig. 3 shows the pseudocode of the setup and execution of the Gaussian Blur CUDA kernel.

D. Sobel Filter Kernel

In the third stage, the Sobel operation is performed on the blurred image results from the previous stage. The Sobel operator is performed in both the X and Y directions via the convolution operation, which emphasizes rapid changes in image intensity and approximates image gradients, which produces an image with highlighted edges.

$$Sx = \begin{pmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{pmatrix} * A \qquad Sy = \begin{pmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} * A$$

EQUATION II. Sobel Filter Kernels

$$S = \sqrt{Sx^2 + Sy^2} \qquad \Theta = atan2(Sy, Sx)$$

EQUATION III. Sobel Magnitude and Direction

The final Sobel magnitude for each pixel is found by taking the hypotenuse of the X and Y results, and the direction of the gradient is found by taking the arctangent of the X and Y results.

In the GPU implementation, a similar tiling scheme is employed in the Sobel CUDA Kernel as was in the Gaussian CUDA Kernel, where the shared tile size is equal to the thread block size, and each member thread of a block participates in the loading of an element into shared memory. Each thread within the output tile range within a thread block will perform a flipped convolution operation for both the x and y direction, using the X and Y Sobel convolution masks that were previously loaded into constant memory during the initialization stage. The magnitude and direction are found via the hypotenuse and arctangent, respectively, and are committed into pre-allocated global arrays within the GPU global memory. The Sobel filter kernel pseudocode can be seen in Fig. 5.

E. Suppression Kernel

In the fourth stage, non-maximal suppression is performed on the Sobel result from the previous stage. Non-maximal suppression consists of observing the immediate neighboring pixels lying along the direction of the gradient about a center pixel.

```
1: Get load thread indices to load shared tile
2: Get output tile thread indices to output
3: If load indices are within image X and Y boundaries
4:          load in image pixel value to input tile
5: Else
6:          load in zero to input tile
7: Wait for barrier synchronization within the block
8: If thread index is within output tile X and Y
9:          Perform flipped mask convolution for Sobel X component
10:         Perform flipped mask convolution for Sobel Y component
11:         If thread index is within output image X and Y boundaries
12:                   Clamp the Sobel X and Y value to 175
13:                   Calculate direction gradient with arctan
14:                   Calculate magnitude via hypotenuse
15:                   Output values to global memory
13:         End
14: End
```

Fig 5. Sobel Filter Pseudocode

Fig 6. Sobel operator applied to image

The center pixel is retained if it is the local maximum amongst the selected neighbors. Otherwise, it is rejected and saturated to 0. The pixels that are remaining are then classified according to two specified “high” and “low” thresholds. Pixel magnitudes above the higher threshold are designated as strong edge pixels and are saturated white. Pixel magnitudes below the lower threshold are designated as non-edge pixels and are saturated to black. Pixel magnitudes residing between the two thresholds are designated as weak edge pixels. Based on these classifications, weak pixels may be reclassified as strong pixels if they neighbor

at least one strong pixel - otherwise, they are reclassified as non-edge pixels and are saturated accordingly.

In the GPU implementation, a similar tiling strategy is used as was used in the Sobel CUDA Kernel, where each thread in a thread block participates in the loading of an input element from the Sobel magnitude results from the global memory into the shared tile.

Fig 7. Non-maximum suppression applied to image

Each thread within the output tile range within a thread block performs non-maximal suppression by loading in the direction of the center pixel from global memory, and checking the relative magnitudes of the neighboring pixels along that direction. As all magnitude values range from 0 to 255 (unsigned), the direction results previously calculated from the Sobel kernel are received as values ranging from 0° to 90° in radian.

This scheme is used to determine the two neighboring pixels that are used in the local maximum testing. Afterwards, the center pixel is classified based on its strength relative to the two predefined thresholds - non-edge pixels are brought to 0, weak pixels are brought to 127, and strong pixels are brought to 255. These results are placed within an intermediate shared tile (of identical dimensions to the first shared tile), as a search is subsequently performed around each weak pixel for any immediately-connected strong pixels so they may be reclassified as either strong or non-edge pixels.

Additionally, output pixels are only written out to the global memory if they reside within a two-pixel border width around the final image, which effectively saturates image borders to zero. This is done to remove the darkening side effects on the borders of the image resulting from the gaussian blur, which results in the image borders being detected by the Sobel Kernel and passing through as detected edges in the final Canny results. These exclusion parameters result in an invariable and imperceptible black border around the final image result. Fig. 8 shows the pseudocode of the setup and execution of the Suppression CUDA kernel.

F. Development Environments

These implementations were developed in two separate environments. The first environment (Linux) was used to build and test the convolution-based CUDA kernels and to develop the CPU implementations. Testing on individual data frames was done via reading and outputting PPM files. This environment proved the simplest for building and quickly developing and testing the base kernels against the results from the CPU implementations.

```
1: Get thread indices for X and Y relative to the image
2: Get load thread indices to load shared tile
3: Get output tile thread indices to output
4: Create shared input tile
5: Create shared output tile
6: If input within image boundary
7:          Load shared input element
8: Else
9:          Load shared input element with 0
10: Wait for barrier synchronization within block
11: If thread indices for X and Y are less than tile width
12:         If output indices are within exclusive border in image
13:                   Keep pixel if it is local maximum along gradient direction
14:         Else
15:                   Suppress pixel to zero
16:         Classify pixel strength as strong, weak, or non-pixel via double thresholding
17:         Reclassify weak pixel as strong if it neighbors a strong pixel
18:         Write pixel to global memory
19:         End
20: End
```

Fig 8. Suppression Kernel Pseudocode

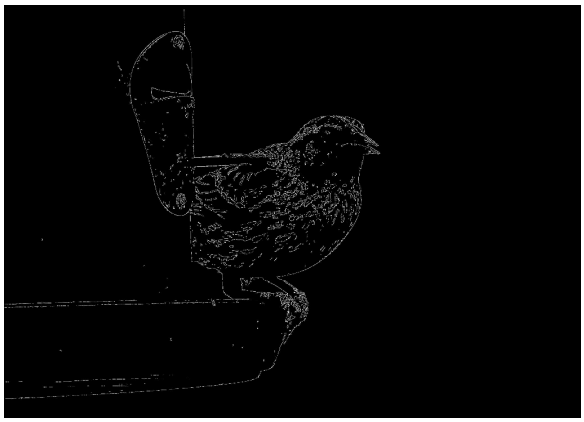

Fig 9. Final Canny edge detector result

The standard timing library, cudaEvents and NVIDIA NSIGHT were used to gather final timing measurements, which compared the sequential CPU implementation against a multi-threaded CPU implementation using TBB, and the CUDA-accelerated GPU implementation. This system utilized a GTX 1060 and a Ryzen 9 7950X. The GTX 1060 features a boost clock of 1.8 GHz, 1280 CUDA cores, and a 256-bit memory bus width [6]. Specifically, the Ryzen 9 proved beneficial for testing the TBB implementation, as it allows up to 32 threads for use in multi-threaded applications [7].

The second environment (Windows) was used to set up an OpenCV camera feed for more user-friendly final results, as well as the building and testing of the suppression kernel. The use of OpenCV allowed rapid interfacing with camera feeds for live video testing. Visual Studio Community was used, which allowed for the integrated use of the NVIDIA NSIGHT debugging toolkit [8]. NSIGHT Systems provided additional timing information and visualizations for each kernel

call and NSIGHT Compute tools provided information pertaining to GPU resource allocation and usage per kernel call, optimization recommendations, and profiling of each individual kernel. A RTX 3090 and an Intel i9 13900k were utilized in this environment. The RTX 3090 features 10496 CUDA cores, a boost clock of 1.7 GHz, and a 384-bit memory bus width [9]. A TBB implementation was not tested on this system, but NSIGHT Systems allowed for a comparison of the RTX 3090 running on the Windows environment vs the GTX 1060 running on the Linux environment.

III. Results and Discussion

A. Overall Results

The results were as expected, with the CUDA C implementation performing much better than that of a sequential C method. Tables I, II, and III were populated from the Linux environment, and a singular PPM image depicting a geometric pattern was looped for 100 iterations to get an average for the timings. Multiple resolutions were also tested, ranging from a 40x30 to 4K resolution.

For the sequential implementation, the timings are clearly the worst, with a 4K image taking up to roughly 1.5 seconds to be fully computed. Both the TBB and CUDA C implementations can take less than 200 ms, with the CUDA C taking less than 15 ms. Looking at the gains over others, the CUDA C implementation over sequential C is by far the most advantageous improvement, with over 100 times speed up. Interestingly, the gain of CUDA C over TBB vs the TBB gain over sequential are converging, with the TBB implementation seemingly scaling better than the CUDA C when compared to each other. This may be caused by the massive increase in resolution, resulting in much bigger memory transfers onto the GPU per image, whereas the CPU-based TBB implementation does not feature this memory transfer overhead. A technique that may have resulted in better scaling may have been to utilize CUDA streams to break the 4K into regions, and "pipeline" the kernel calls. This will be discussed further later on.

Sequential Implementation (ms)					
Resolutions	**40x30**	**640x480**	**1280x720**	**1920x1080**	**3840x2160**
Grayscale Total	0.005	0.870	2.510	5.050	17.900
Gaussian Blur Total	0.097	23.900	72.700	162.500	644.700
Sobel Filter Total	0.098	23.700	71.300	158.300	626.400
Suppression Total	0.020	7.100	22.100	47.000	193.500
Total	**0.220**	**55.600**	**168.700**	**372.900**	**1482.500**

TABLE I. Sequential Implementation Kernel Timings

TBB Implementation (ms)					
Resolutions	**40x30**	**640x480**	**1280x720**	**1920x1080**	**3840x2160**
Grayscale Total	0.050	5.080	0.600	1.020	2.820
Gaussian Blur Total	0.033	5.080	12.900	19.100	55.300
Sobel Filter Total	0.084	7.790	18.300	27.100	71.100
Suppression Total	0.050	6.190	14.100	21.100	48.600
Total	**0.220**	**19.330**	**45.910**	**68.350**	**177.720**

TABLE II. TBB Implementation Kernel Timings

CUDA Implementation (ms)					
Resolutions	**40x30**	**640x480**	**1280x720**	**1920x1080**	**3840x2160**
Grayscale Total	0.021	0.140	0.321	0.659	2.410
Grayscal Kernel	0.008	0.025	0.056	0.104	0.360
Gaussian Blur Total	0.017	0.051	0.130	0.271	1.050
Gaussian Blur Kernel	0.009	0.042	0.122	0.261	1.040
Sobel Filter Total	0.026	0.340	0.948	2.150	8.570
Sobel Filter Kernel	0.013	0.330	0.936	2.130	8.570
Suppression Total	0.021	0.120	0.290	0.610	2.220
Suppresion Kernel	0.008	0.063	0.163	0.363	1.440
Total	**0.085**	**0.650**	**1.670**	**3.690**	**14.250**

TABLE III. CUDA Implementation Kernel Timings

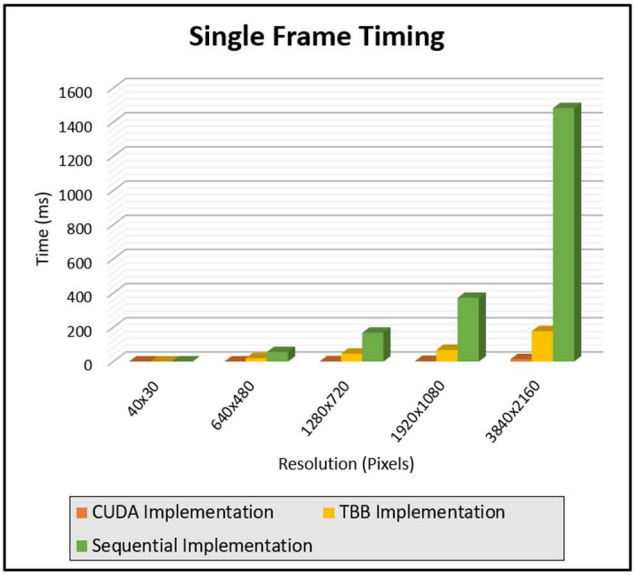

Fig 10. Single Frame Performance Metrics

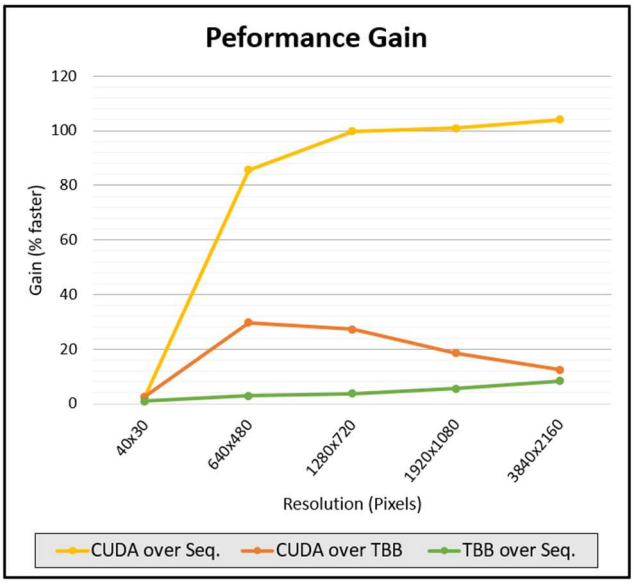

Fig 11. Performance Gain of Implementations

B. NSIGHT Systems and Kernel Breakdown

Further testing was done using NSIGHT Systems and NSIGHT Compute on the Windows environment. NSIGHT systems allowed for a full profiling of the application running, accurately detailing each CUDA API call, along with each individual kernel call. Systems also presents a detailed breakdown of the average timing for each CUDA API call. NSIGHT Systems was used for profiling on both systems, so a direct comparison between the RTX 3090 and GTX 1060 was performed as seen in Table IV.

Table IV shows the RTX 3090 performing at roughly 1.8 times faster than the GTX 1060. Noticeably, the suppression kernel gets the biggest speed up when looking at each kernel, with a performance gain of roughly 3.0 times. Fig. 10 and 11 show the total time for a frame to compute as well as the scaling of the performance. It should be noted that this build of the application has only the RGB cudaMemcpy() input, followed by each subsequent cudaSynchronize() and each kernel, and lastly the cudaMemcpy() output from the final kernel. For debugging purposes, a cudaMemcpy() output for each kernel was used to evaluate and visualize each stage.

GPU Nsight Systems Analysis (ms)			
Kernel	**RTX 3090**	**GTX 1060**	**3090 Speed Up (%)**
Grayscale	0.015	0.096	604.6
Gaussian Blur	0.048	0.268	557.7
Sobel Filter	0.629	2.09	333.4
Suppression	0.064	0.377	585.2

TABLE IV. NSIGHT Systems GPU Comparison Data

NSIGHT Compute Launch Statistics for Kernels				
Kernel	**Grayscale**	**Gaussian Blur**	**Sobel Filter**	**Suppression**
Grid Size	8,160	8,160	8,160	8,160
Block Size	256	400	324	234
Threads	2,089,960	3,264,000	2,643,840	2,642,840
Registers / Thread	56	31	56	32
Static Shared Mem. (Bytes)	324	400	324	648
Theroretical Occupancy (%)	100	81.25	68.75	91.67
Achieved Occupancy (%)	90.6	64.12	26.23	76.11

TABLE V. NSIGHT Compute Analysis Data

Lastly, NSIGHT Compute was used to report the GPU resource allocation and utilization. NSIGHT Compute allows for further debugging by showing bottlenecks in memory transfers, warp stall states, under utilization of streaming multiprocessors (SM). The statistics for the kernels can be seen in table V.

Starting with the RGB kernel, it can be seen that the theoretical occupancy for a given SM is 100%. This is due to the block size being 16x16 (256) and each SMs thread limit being 1536 threads, which allow a perfect ratio of six full thread blocks per SM. The actual occupancy checked in around 90%, limited due to the image sizes not being perfect multiples of 256, thus, along the edges of the image were partial thread blocks that had unused threads.

For the gaussian blur kernel, the grid size is now bigger than the RGB one of 256, as this implementation uses all threads to load input data, but only some to output data. Because of this difference, the theoretical occupancy was reduced to about 82%. Similar to before, as the image was not a multiple of 256, the theoretical occupancy was reduced to 60%.

C. Further Improvements

The last two kernels both suffer from similar implementations, with the block size being bigger than the output tile due to the loading of kernel data. From previous testing, as well as the data from the NSIGHT Compute analysis, the Sobel filter is by far the worst performing kernel. With this in mind, a look into the warp scheduling was done using Compute. NSIGHT Compute showed that lines 9-10 in Fig. 5 were the biggest bottleneck in the Sobel kernel. This is likely due to the convolution mask data reads from constant memory being uncoalesced due to the mask flipping during the convolution operation, which reverses the read order for the mask.

The calculations to determine final magnitude (hypotenuse) and direction (atan2) of the Sobel inverse tangent function were determined to be a major bottleneck within the Sobel Kernel. Replacing these computations with simple constant assignment reduced the processing time of the Sobel Kernel by a factor of 8.0 times.

From the previous section, the Sobel filter kernel was determined to be a major bottleneck CUDA C loop. One potential improvement would be to exclude the calculations for the row of 0's within each Sobel kernel, which would reduce the number of shared memory accesses for each Sobel convolution from 9 down to 6. Approximating atan2 with an n-order polynomial versus the original atan2 function would have significantly reduced the processing time associated with direction approximation [10]. Alternatively, a simpler method to approximate direction by taking a ratio of Y/X could have been used.

Additionally, using OpenCV as the high-level API for frame grabbing proved to not work as ideally as hoped. OpenCV took a big overhead in terms of how fast it could grab a frame, limiting the throughput of the total application. While not explicitly an improvement to the CUDA C implementation, it would incorporate some of the same elements. Utilizing some pipelining, ideally, all at the same time an image can be grabbed from the camera, another one being loaded into the GPU, another image is being computed by the GPU, another image is being output to the host, and finally an image is being shown to the user. This would involve a combination of utilizing both CUDA Stream elements, along with pipeline techniques on the host CPU. More research would be needed to better implement the host pipelining. For the CUDA C, one idea would be to have one kernel total instead of different kernels for each stage. Reviewing the NSIGHT Systems profiling, it can be seen that the first RGB cudaMemcpy() takes in total longer than the RGB, gaussian, and suppression kernels combined including

the cudaSynchronize(). It is relatively similar to the Sobel kernel as well. If utilizing CUDA streams allows for full use of the PCIe lanes, then relatively little downtime in the pipelines would happen, so long as the Sobel filter can be sped up. The pipelining of the CUDA C would have a smaller impact that pipelining the host CPU OpenCV frame grabber, it would still improve the theoretical frame compute time in CUDA C, which is the ultimate goal.

IV. Conclusion

The implementation of a CUDA-accelerated Canny Edge Detection was a success. Not only did the CUDA C speed up the application over the standard sequential C, it also sped up over the TBB implementation. With improvements of up to 100 times speed for a 4K image over the sequential implementation, CUDA C proved to be an effective strategy for accelerating this application. Future improvements including CUDA Streams, a refined Sobel filter and a more focused approach towards ensuring coalescing for global memory accesses are expected to increase the gains in the GPU implementation. Additionally, the use of the Nvidia NSIGHT debugging toolkit provided meaningful insights into bottlenecks and performance issues. In future work, the knowledge of how to use these tools will provide a better development period, along with the ability for more robust testing and profiling.

References

[1] J. Liang, Canny Edge Detector, N.d. [Online]. Available: https://justin-liang.com/tutorials/canny/

[2] T. L. Ben Cheikh, G. Beltrame, G. Nicolescu, F. Cheriet and S. Tahar, "Parallelization strategies of the canny edge detector for multi-core CPUs and many-core GPUs," *10th IEEE International NEWCAS Conference*, Montreal, QC, Canada, 2012, pp. 49-52.

[3] S. Rahamneh and L. Sawalha, "An OpenCL-Based Acceleration for Canny Algorithm Using a Heterogeneous CPU-FPGA Platform," *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, San Diego, CA, USA, 2019, pp. 322-322.

[4] Y. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, Jul. 2008.

[5] OneTBB documentation. (2022) *parallel_for - oneTBB documentation.* [Online]. Available: https://oneapi-src.github.io/oneTBB/main/tbb_userguide/parallel_for_os.html

[6] Specifications. (2021). Nvidia GTX 1060. [Online]. Available: https://www.nvidia.com/en-gb/geforce/graphics-cards/geforce-gtx-1060/specifications/

[7] Specifications. (2022). AMD Ryzen R9 7950x. [Online]. Available: https://www.amd.com/en/product/12151

[8] CUDA toolkit documentation V11.8.0. (Jan 2023). Nvidia. Available: https://docs.nvidia.com/cuda/index.html

[9] Specifications. (2023). Nvidia RTX 3090 & 3090 Ti. [Online]. Available: https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/

[10] Gironés, Xavier & Julià, Carme & Puig, Domenec. "Full Quadrant Approximations for the Arctangent Function," IEEE Signal Processing Magazine. 30. pp. 130-135, Jan. 2013.