Canny Edge Detection Using an NVIDIA GPU and CUDA

Alex Wade CAP6938 Final Project

Introduction

This project is a CUDA based implementation of *A Computational Approach to Edge Detection*, by John Canny. In this paper, Canny presented an accurate, localized method of edge detection. Canny's method uses multiple parallelizable matrix and floating point operations, which makes it an algorithm that can potentially have major performance increases if implemented in CUDA and run on an NVIDIA GPU. High performance edge detection is useful for computer vision, as edge detection is an important step in many image processing algorithms. This project demonstrates that using general-purpose GPU computation does in fact result in a faster performing implementation of the Canny algorithm.

Algorithm Overview

Canny's algorithm consists of five major steps: image smoothing, gradient computation, edge direction computation, nonmaximum suppression, and hysteresis. This description of the algorithm assumes that the input image is already in grayscale format.

Image smoothing is performed on the image by convolving it with a Gaussian smoothing mask, such as the one in Figure 1. Smoothing serves the purpose of eliminating any sudden changes in intensity that may occur from phenomena such as image static, compression artifacts, camera problems, or bright reflections. Eliminating these sudden changes reduces the number of falsely detected edges in the output. Figure 2 below shows the result of the smoothing operation.

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

$\sigma=1.4$

Figure 1. Gaussian smoothing mask

Figure 2. Result of Gaussian smoothing operation

After smoothing has finished, the image gradient is computed. The gradient of the image is used to determine the edge locations, which lie inside areas of high changes in intensity. Gradient computation for each direction is performed through convolving the smoothed image once with a mask that produces a horizontal gradient and once with a vertical gradient mask (see Figure 3). The gradient magnitude for a location (X, Y) is obtained by adding the absolute values of position (X, Y) on the horizontal and vertical gradient image together, using the formula $|\nabla| = |\nabla_x| + |\nabla_y|$. The outcome of this operation can be seen in Figure 4.

-1	0	1
-2	0	2
-1	0	1

∇_x

1	2	1
0	0	0
1	2	1

∇_y

Figure 3. Horizontal and vertical gradient masks

Figure 4. Gradient magnitude image

Edge direction is computed using the horizontal and vertical gradient images computed in the previous step. The direction for an edge located at location (X, Y) is computed by using the formula $\theta_{x,y} = \tan^{-1}(\nabla_x/\nabla_y)$. Edges are then classified as being in one of four directions by snapping them to their nearest positive 45-degree angle. For example, edges that lie between 112.5 degrees and 157.5 degrees are snapped to 135 degrees. Figure 5 shows a visual depiction of the edge computation and classification process.

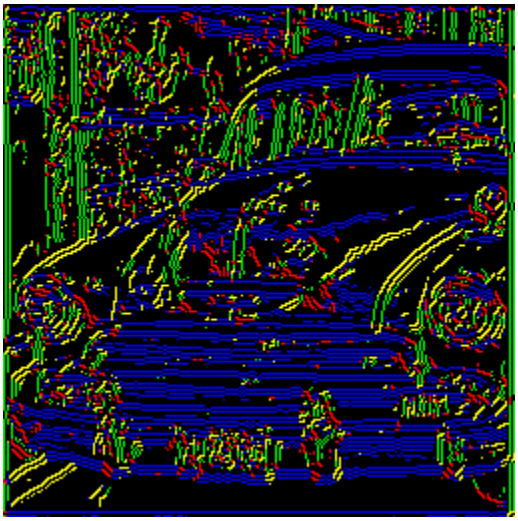

Figure 5. Edge classification results; blue lines are classified as 0°, yellow as 45°, green as 90°, and red as 135°

Nonmaximum suppression is used to localize the edges down to a single pixel. Each pixel in the gradient is visited and its magnitude is compared to that of each of its two perpendicular neighbors. Every pixel that does not have a higher magnitude than its neighbors has its value set to zero, and all pixels that are local maxima are retained. Perpendicular neighbor locations

are computed based on the edge directions that were previously computed. Nonmaximum suppression results in an image like the one in Figure 6.

Figure 6. Gradient magnitude image after nonmaximum suppression

At this point, all of the edges have been localized. However, many of the edges still present have very small magnitudes and are not proper edges. Hysteresis is used to eliminate these edges. A high and a low threshold are used for the hysteresis operation. The result image from performing nonmaximum suppression is scanned and all pixels that have magnitudes greater than the high threshold are added to the output edge image. Each of the neighbors of a newly added pixel are recursively scanned and are added if they fall below a low threshold. A list of previously visited pixels is kept to prevent pixels from being visited multiple times, which would cause an infinite loop. Each pixel in the final output image (see Figure 7) will have either an "on" or "off" value; intensity and magnitude values are not kept.

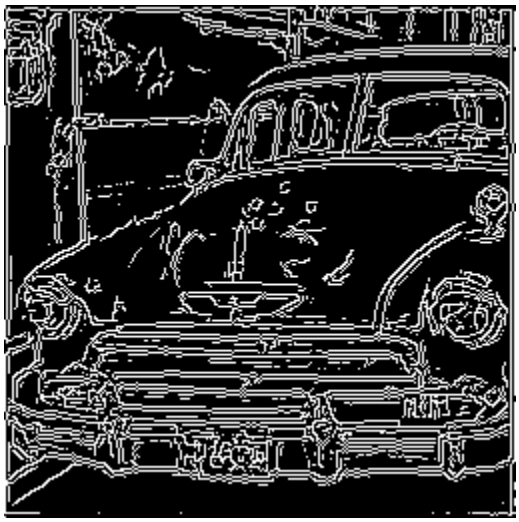

Figure 7. Final output image

Implementation Details

ImageMagick 6.5.1 was used to read the input file from disk and to write the output back to disk. CUDA 2.1 was used for the GPU computation.

In this implementation, all of the convolution masks are copied into constant device memory at initialization time. The input image is copied into texture memory, and a kernel function that performs smoothing is executed. The result is copied back to the host device and placed into texture memory, then a second kernel function is called that performs gradient computation, edge direction computation, and edge direction classification. The resulting gradient magnitude and edge direction information are copied back to the host. The remaining steps (non-maximum suppression and hysteresis) are performed by the CPU.

The Possible Improvements section of this document contains details about the reasoning behind these implementation choices.

Experiments

Performance of this project was conducted on a host system with an Intel Core 2 Quad processor running at 2.66 GHz and 3.25GB of RAM. The device used for testing was an NVIDIA GeForce 8800 GT with 512 MB of memory.

Evaluation was performed by running a CPU-only version of the edge detection algorithm and the aforementioned GPU based implementation using input images of sizes 256x256, 512x512, 1024x1024, and 2048x2048. For each size image, the output was checked for correctness and hysteresis threshold values were determined so that the output images for both implementations had the same number of pixels. Timing was collected for overall execution not including file I/O, the Gaussian smoothing operation, the direction classification computation, and the gradient computation operations. All timings were recorded five times and averaged for the results.

Results

The entire image detection algorithm performed faster for every size input image. For all image sizes, the performance increase was between 46 and 48 percent. Figure 8 shows a comparison of the CPU-only and GPU-based implementations. For the portions of the algorithm performed entirely with the GPU (image smoothing, gradient computation, edge direction computation, and edge classification), the improvement was much larger. The smallest input image was processed 88.8 percent faster by the GPU and the larger input images were processed between 94.1 and 94.8 percent faster. It is worth noting that the granularity of the timer may be course at small intervals, explaining the difference between the smallest image and the rest. Figure 9

shows a comparison of the GPU-only portion of the algorithm to the CPU-only version of that portion.

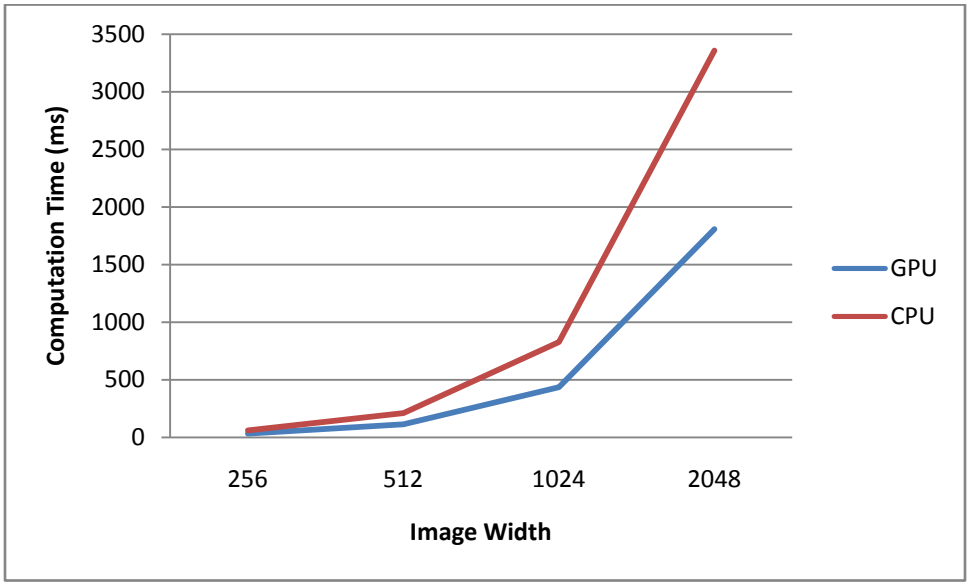

Figure 8. GPU vs. CPU performance comparison for Canny algorithm

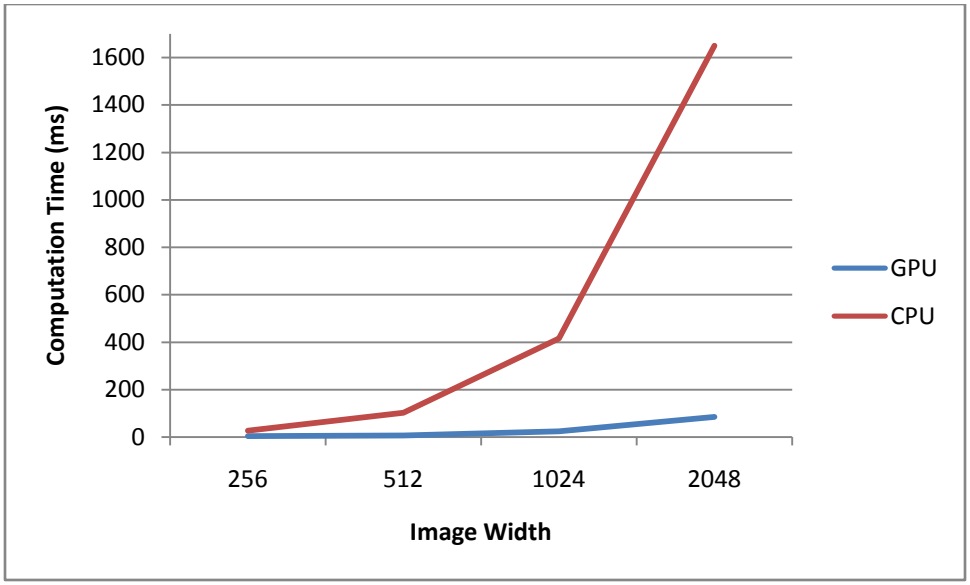

Figure 9. GPU vs. CPU performance for portions of the algorithm with GPU-only implementations

Possible Improvements

Given more time, the use of device memory can be improved resulting in better performance. Attempts at an implementation that used read/write accessible device memory properly resulted in crashes and incorrect output. Using the memory in this fashion would eliminate the need for read-only texture memory use and the round trip taken by the image between the

smoothing operation and the rest of the operations performed by the GPU. The current implementation choice is a workaround for the memory bugs that could not be fixed in the timeframe for this project.

Additionally, nonmaximum suppression and hysteresis could potentially be performed by the GPU. A GPU based version nonmaximum suppression was implemented, but no modifications to it resulted in faster performance than the CPU based version. This is mainly due to the large number of branches used to determine the location of perpendicular neighbor cells. No working GPU based hysteresis implementation was created, due to a lack of support for recursion in CUDA. Redesigning the logic used for both of these steps will likely make it possible to implement them on the GPU and result in a greater speedup.

Finally, researching more about CUDA and NVIDIA GPUs would result in further performance improvements because the code could be optimized further. Further optimization to the CPU code would also be beneficial because it would provide a more accurate baseline to which the GPU performance can be compared.

Conclusions

The results from this research indicate that there is a benefit to implementing the Canny edge detection algorithm. The increased performance improvement of the portions of the algorithm that had a GPU-only implementation versus the overall improvements indicate that further research to implement more of this algorithm using a GPU would be useful. This project has also served as an educational experience in CUDA and GPGPU programming as well as image processing.