Final Project
CUDA Edge Detection

Daniel Woldegiorgis

Keywords: Edge Detection, Gaussian Blur.

Abstract

Edge detection is concerned with finding the points in an image where there are sharp changes/discontinuities in the brightness or intensity of an image. Edge detection has numerous applications as an integral part of image processing and computer vision algorithms. This project implements an edge detection algorithm using CUDA and shows different optimization strategies such as the use of shared memory to reduce the number of global memory accesses in one of the expensive operations involved in edge detection: convolution. The results are applicable not just to the 3x3 sobel kernels that were used in this particular project but also to different kernels of different types and sizes.

1 Introduction

Over the years, a number of different ways of performing edge detection have been developed; however, most of these methods can be grouped into one of two categories: search based or zero-crossing based.[1]

The search based methods detect edges by first computing a measure of edge strength, usually a first order derivative expression like the gradient magnitude, and then search for local directional maxima of the gradient magnitude using a computed estimate of the local orientation of the edge, usually the gradient direction. [1]

The zero crossing based methods search for zero crossings in a second order derivative expression computed from the image in order to find edges, usually the zero crossings of the Laplacian is used. [1]

This project makes use of the search based strategy by using the sobel operator to obtain the first order derivative gradient magnitude as well as the gradient direction.

In addition, similar to most other edge detection algorithms, the images are smoothed using a Gaussian blurring function in order to remove noise points that might reduce the quality of the output of the edge detection operation.[2]

2 Methodology

The edge detection algorithm implemented in this paper can be summarized as follows:

1. Convert Input Image to Grayscale
2. Apply Gaussian filter
3. Perform Convolution with Sobel operators (filters)
4. Obtain the gradient magnitude and direction
5. Apply non-maximum suppression
6. Apply double thresholding
7. Perform edge tracking by hysteresis

I chose to use the CUDA programming environment for my implementation because the biggest bottlenecks in the algorithm above are the convolution operations needed for performing Gaussian filtering as well as applying the sobel kernels, which are both compute-intensive arithmetic (floating point) operations. And GPUs are good for performing such tasks with massive parallelism.

Each one of the steps in the algorithm is explained and illustrated below.

2.1 Convert Input Image to Grayscale

The goal here is to convert each color pixel that has 3 components (Red, Greeen and Blue) to a pixel with just one component (Gray). There are three common ways of creating grayscale images.[3]

1. Lightness Method
   This method involves averaging the most prominent and least prominent component colors. That is,

$$\frac{max(R,G,B) + min(R,G,B)}{2} \tag{1}$$

   The disadvantage with such a way of calculating gray pixels is that it is susceptible to extreme values. (Same issue as an outlier affecting the mean of set of numbers).

2. Average Method
   This method involves averaging the three components of the pixel to obtain one final pixel value. That is,

$$\frac{(R,G,B)}{3} \tag{2}$$

   This method is able to create a good grayscale approximation; however, it assumes that humans have equal perception to Red, Green and Blue colors.

3. Luminosity Method
   This method also involves averaging the three components of the pixel; however, each of the three components is weighted to account for human perception. Humans are more sensitive to green color, followed by red, and blue;so, the green is weighed heavily followed by red, and blue. That is,

$$L = 0.21R + 0.72G + 0.07B \tag{3}$$

   Therefore, I have chosen to use the Luminosity method of converting the input images to grayscale.

   I have included a sample image below and showcased the grayscale conversion obtained using the Luminosity method.

Figure 1: Original Image

Figure 2: Luminosity Grayscale Image

2.2 Apply Gaussian filter

The gaussian filter is a filter that uses the Gaussian function to perform a Weierstrass transform of the input image, thereby removing noise pixels that might distort the edge detection. The two dimensional gaussian function that is used in this paper is:

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

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution.

This distribution can be approximated using the following kernel 3x3 kernel:[4]

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

A sample output of the application of a gaussian kernel to the grayscale image obtained from the previous operation is given below.

Figure 3: Gaussian Image

2.3 Perform Convolution with Sobel operators

The sobel filters are two 3x3 image gradient kernels that can be convolved over a given image to calculate the approximations of the derivatives for horizontal as well as vertical changes. Given an image A, G_x and G_y where each point contains the vertical and horizontal approximations of the derivatives respectively can be calculated as follows:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \tag{5}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A \tag{6}$$

where * is a convolution operation.

A sample output of the application of the horizontal and vertical convolution operations using the respective sobel kernels is shown in Figures 4 and 5.

Figure 4: Horizontal Convolution Image

Figure 5: Vertical Convolution Image

2.4 Obtain the gradient magnitude and direction

At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$G = \sqrt{G_x^2 + G_y^2} \tag{7}$$

we can also calculate the gradient's direction:

$$\Theta = atan(\frac{G_y}{G_x}) \tag{8}$$

A sample output of the gradient magnitude approximation using the given formula is shown in Figure 6.

Figure 6: Gradient Magnitude Image

2.5 Apply non-maximum suppression

Non-maximum suppression is an edge thinning technique. [5] The algorithm for each pixel in the gradient image is:

- Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.

- If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (e.g., a pixel that is pointing in the y-direction will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.

In general, at every pixel, it suppresses the edge strength of the center pixel (by setting its value to 0) if its magnitude is not greater than the magnitude of the two neighbors in the gradient direction.

A sample output of the non-maximum suppression of the gradient magnitudes from the previous operation is shown in Figure 7.

Figure 7: NonMaxSuppression Image

2.6 Apply double thresholding

Double Thresholding is used to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. [5] One of these three things will happen to a given pixel during this operation:

1. If a pixel's magnitude is higher than the high threshold value, it is marked as a strong edge pixel.

2. If a pixel's magnitude is smaller than the high threshold value and larger than the low threshold value, it is marked as a weak edge pixel.

3. If a pixel's magnitude is smaller than the low threshold value, it is suppressed.

A sample output of the double thresholding operation applied to the output of the non-maximum suppression is shown in Figure 8.

Figure 8: Double Thresholding Image

2.7 Perform edge tracking by hysteresis

Hysteresis is a follow up to the result of the doubleThreshold operation. The 8 neighboring pixels of every pixel are considered and if any one of them are strong, then the pixel becomes strong; otherwise, the pixel is suppressed. [5]

A sample output of the hysteresis operation applied to the output of the double thresholding operation is shown below.

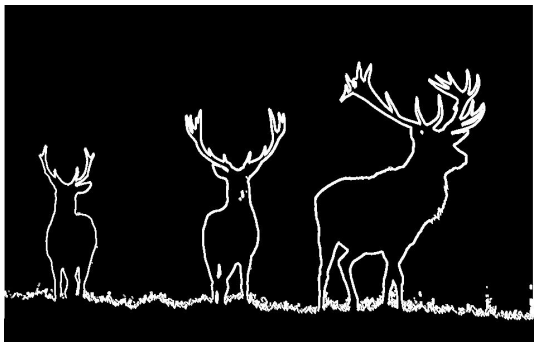


Figure 9: Hysteresis Image

3 Results and Optimizations

I have presented the different optimizations that were included/tested in the edge detection implementation in this section of the paper. But before that, I have presented the timing results for the serial

version of the algorithm on images of different sizes to serve as a benchmark for improvements.

Image Size	Time taken(in milliseconds)
Small (testimg.jpg)	6.975
Medium (fall.jpg)	198.55
Large (gorge.jpg)	1309.354

Table 1: Serial Edge Detection Result

3.1 Parallelization organization

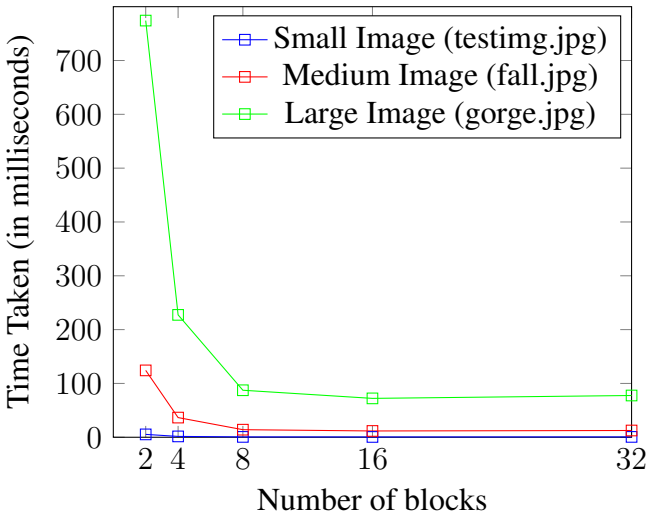
I used 2 dimensional blocks and grids, and the blocks were equal in their x and y dimension (squares) for the testing results presented in this write up. One thing to note here is that the timing information does not include memory transfer times to and from the kernel.

The results obtained using different block sizes 2, 4, 8, 16 and 32 (which is the maximum possible square block size because 32*32=1024, which is the CUDA limit for the number of threads), is tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified. Note that, these are the results of the parallel execution before any shared memory and kernel constant memory optimizations have been performed.

#Blocks	Small (testimg)	Medium (fall)	Large (gorge)
2	5.326848	124.239746	774.173706
4	1.577952	36.549599	227.363678
8	0.656352	14.067744	87.344131
16	0.572416	11.686912	72.226845
32	0.645088	12.551296	77.57814

Table 2: Parallel Edge Detection Result

Performance Results on Parallel



As we can see from the figure above, there is generally an increase in performance with increasing block size. However, the rate of improvement continues to decrease as we increase the block size beyond 8.

We know that a block assigned to an SM is further divided into 32 thread units called warps. Each block will have (threads in block/32) warps and we want to have as many warps as we can in order to avoid stalling. That is, there will likely be at least some warps that are ready to run at any given time when we have many such candidate warps. Therefore this increase in performance with increasing block size is expected, however given the total number of threads that the SM can handle is fixed, increasing the number of blocks doesn't result in the same amount of improvement beyond a given block size.

3.2 Non-Parallelization Optimizations

Some of the non parallel optimizations included in the edge detection implementation are presented below. These include using constant memory to store the convolution kernels when performing convolution, using shared memory to pull in sections of the image via tiling to reduce the number of global memory accesses during convolution, and also a combination of both shared memory use and kernel caching in constant memory.

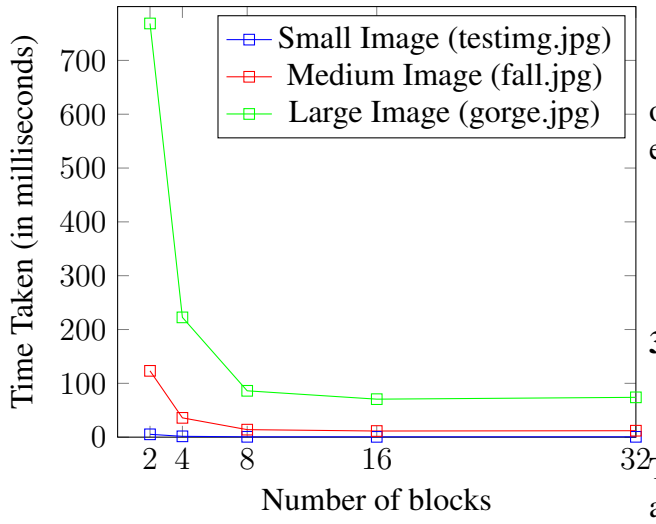
3.2.1 Constant Memory

The results obtained using different block sizes 2, 4, 8, 16 and 32 is tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

# Blocks	Small (testimg)	Medium (fall)	Large (gorge)
2	5.266592	123.195358	768.626648
4	1.543136	35.858433	222.532608
8	0.650176	13.95616	86.10714
16	0.559104	11.397088	70.593536
32	0.619488	11.972672	73.975807

Table 3: Constant Memory Parallel Edge Detection Result

Performance Results on Constant Memory

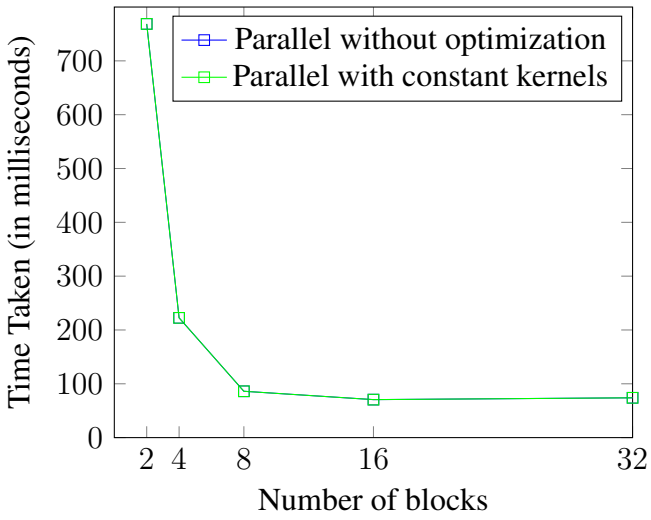


Explanation: As we can see from the figure above, there is generally an increase in performance with increasing blocksize. However, the rate of improvement continues to decrease as we increase the block size. We can see that there increasing the block size from 2 to 4 has a big jump in improvement for the large image which suggests that block size 4 provides a higher occupancy thereby performing better at hiding latency.

For small image the cost of copying the kernels to constant memory is expensive because the frequency of the lookup operation is low for small images. There is therefore not as much of an improvement.

In addition to help gauge the results of this optimization I have reproduced the result of the performance of the algorithm on a large image pre optimization and following this optimization below.

Performance Comparison



We can see from this plot that this particular optimization did not quite benefit the performance enough to see a noticeable difference.

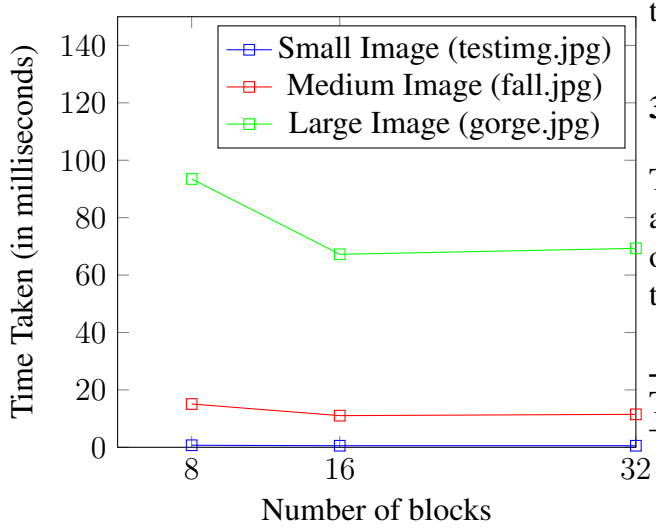
3.2.2 Shared Memory

The results obtained using different block sizes 8, 16 and 32 is tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

Tile Size	Small (testimg)	Medium (fall)	Large (gorge)
6	0.711744	15.078336	93.470848
14	0.54576	11.02848	67.234787
30	0.542816	11.48224	69.304352

Table 4: Shared Memory Parallel Edge Detection Result

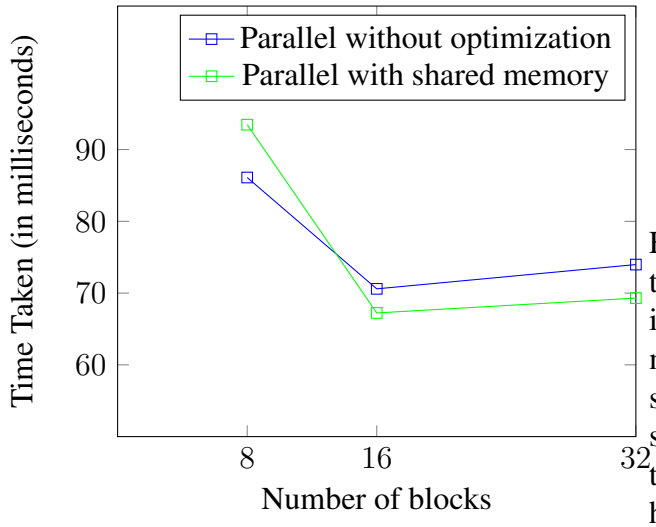
Performance Results on Shared Memory



Explanation: As we can see from the figure above, there is generally an increase in performance with increasing blocksize. However, the rate of improvement continues to decrease as we increase the block size. In addition we observe that the benefit of shared memory is more apparent for larger images than smaller images where the number of global accesses are not as high to begin with.

In addition to help gauge the results of this optimization I have reproduced the result of the performance of the algorithm on a large image pre optimization and following this optimization below.

Performance Comparison



We can see that for higher number of blocks the shared memory begins to outperform the parallel implementation without any optimizations because as the tile size increases more data is pulled into shared memory reducing the frequency of pulling data into the shared memory.

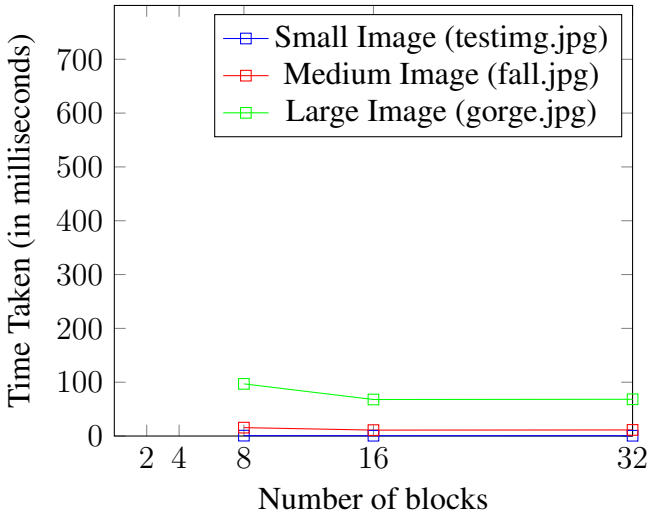
3.2.3 Shared and Constant Kernel

The results obtained using different block sizes 8, 16 and 32 is tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

Tile Size	Small (testimg)	Medium (fall)	Large (gorge)
4	0.722016	15.600704	96.908386
14	0.543744	11.012064	67.849342
30	0.534592	11.309088	68.266014

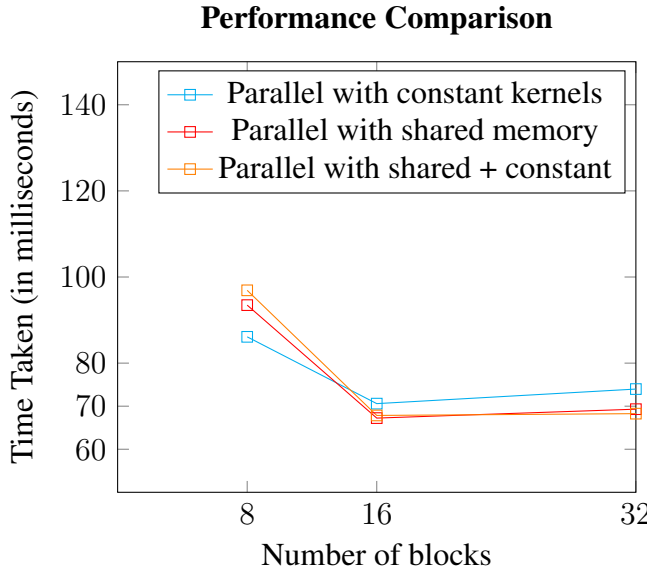
Table 5: Shared + Constant Memory Parallel Edge Detection Result

Performance Results on Shared + Constant



Explanation: As we can see from the figure above, there is generally an increase in performance with increasing blocksize. However, the rate of improvement continues to decrease as we increase the block size. In addition we observe that the benefit of shared memory is present however the addition of the constant kernel memory does not seem to have helped increase the performance.

In addition to help gauge the results of this optimization I have reproduced the result of the performance of the algorithm on a large image with the different optimization strategies considered below.



We can see that the inclusion of shared memory is the crucial optimization that was able to make significant improvements in the performance of the algorithm. This can be explained by the fact that convolutions are operations that need to access a minimum of 8 neighbors for every given pixel and pulling in all of these at a time as one transaction and then having 5 clock cycles per lookup/write (instead of looking up from global memory which takes 500 clock cycles for read and write per grid) is going to speed up the performance of the algorithm.

4 Conclusion

There are a number of edge detection strategies out there; however, there are some operations that are usually present in most detection algorithms such as gaussian filtering and applying convolution using a kernel which is also the strategy that is utilized in performing edge detection.

The aim of this project was to write and test an edge detection algorithm using CUDA and observe the performance benefits of different optimization strategies. We were able to observe that shared memory optimizations provide a good source of improvement for the performance of the edge detection algorithm.

In addition to this, I noticed that parameter selection during the double thresholding phase has the power to make or break the algorithm; however, there is no parameter selection strategy that rules them all. I have found that the high threshold that worked well for most of the images was 0.18 * highest pixel value in the image. However, this threshold parameter tuning is still an open question.

Lastly, the non max compression, double thresholding and hysteresis steps do not significantly change the output of the image and if the edge detection algorithm doesn't need to have very high degree of precision they can be omitted when say trying to do an edges precomputation before using it in Computer Vision algorithms.

5 Future Work

There are a number of different ways to improve the parameter selection process. For instance, Otsu's Method ccould have been used on the non-maximum suppressed grradient magnitude to generate the high threshold.[5] The low threshold would then be 1/2 of the high threshold.

One other simple optimization that was skipped for legibility in this implementation is performing the horizontal and vertical convolutions by the sobel operator at the same time by passing in two outputs. One other area that can be explored is the use of different tiling strategies instead of simply sticking to powers of 2 for the block size.

Calculating the gradient magnitude involves squaring both the horizontal and vertical convolutions of the images, finding their sum and obtaining the square root. However, we can approximate this with an absolute value sum of the horizontal and vertical convolutions. There is a serial implementation of this presented in the source code for this paper.

Finally, one can use other parallel programming frameworks different from CUDA to see if there are any benefits over a CUDA implementation.

References

[1] `https://en.wikipedia.org/wiki/Edge_detection`

[2] Djemel Ziou and Salvatore Tabbone. *Edge Detection Techniques - An Overview* INTERNATIONAL JOURNAL OF PATTERN RECOGNITION AND IMAGE ANALYSIS, Volume 8:537-559, 1998.

[3] John Cook.
https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

[4] https://en.wikipedia.org/wiki/Kernel_(image_processing)

[5] https://en.wikipedia.org/wiki/Canny_edge_detector