University of Stuttgart
Germany

Institute for Parallel and Distributed Systems
Professor Dr.-Ing. Sven Simon

GPU Lab Project Report

Performance Benchmarking for Canny Edge Detector using OpenCL

Mohit Kalra, Fahad Ghouri

Date of hand out:	June 7, 2019
Date of hand in:	September 19, 2019
Supervisor:	Kaicong Sun

Abstract

In modern-day computer vision applications, edge detection is one of the fundamental tasks that is carried out to extract features from an image. These features can then be used to perform higher semantics vision processing and object detection. Traditionally, Sobel filters have been quite popular for the task of edge detection as they are the easiest to implement. However, Sobel filters lack crisp and precise edge detection and output images are usually very noisy. Canny Filter makes use of Sobel Filter at its core and applies some Pre- and Post-Processing to improve the quality of the edge detection by many folds. For image processing algorithms, there is a high potential for parallelism since the mathematical operations performed are fundamental in nature and quite repetitive. Implementing these algorithms on a CPU is quite easy but is not highly efficient. Since CPUs usually have fewer cores, much of execution of the filtering algorithms are sequential in nature. To exploit the true potential of parallelism of these algorithms a parallel GPU implementation is highly desirable. This report elaborates the project carried out to implement and accelerate the Canny Edge Detector on a GPU using OpenCL. This is achieved by breaking down the complete filtering algorithm into smaller independent kernels which carry out one aspect of the processing element. To make for a smooth user interface experience, the algorithm is packaged along with a QT-based GUI running in the front end. Finally, performance benchmarking of the Canny Edge Detector's implementation is done both on the CPU and GPU to compare the speedups that are achieved.

Title page image: Comparison of Input Grayscale Image vs Canny Edge Detector Output

Contents

1 Introduction

This section briefly introduces the different techniques which are used in this project for edge detection. Development Toolkit such as Eclipse for core and QT for GUI based functionality has been used.

1.1 Edge Detection

Edge detection is the technique of extracting boundaries between different objects of an image by the process of checking for gradient change in the intensity value pixel by pixel. These edges hold to define the change of object or patterns and hold valuable images in terms of segments and semantics [6]. One of the most popular techniques of edge detection is the Sobel Filter [3].

1.2 Sobel Filter

Sobel Filter uses first-order gradients along each axis of a 2-D image and combines them using the square root of the sum of squares of each gradient sum calculated in each 3x3 filter size. Sobel filters are quite simple to implement and deliver good results for the most part where images have good contrast in between the objects. However, it is highly sensitive to the scale of detection particularly at the edges where the contrast within pixels isn't quite high. This results in some edges with high intensity and boldness while on the other hand, some edges with minuscule intensity, disconnected edges and a lot of noise. This can also occur when there is a contrast change within an object without an edge or vice-versa.

1.3 Canny Filter

The limitation of Sobel Filter can be improved by using certain Pre- and Post- Processing algorithms. These collectively make up the Canny Filter [4]. The Canny filter is an excellent filter for edge detection task and produces uniform edges for the objects in the image. It is also well known for good localization, uniformity of the edges, and excellent noise reduction. There are four main stages within a Canny Filter, these are

1. Gaussian Blur

2. Sobel Filtering
3. Non-Maximum Suppression
4. Hysteresis
5. Edge Tracking

1.4 Computational Complexity

Implementing these algorithms can be quite computationally demanding. Consider an example of an image 640x480 pixels in size. There are 307200 pixels in this image and each pixel is operated upon by mathematical operations of the image processing algorithms such as multiplications, summation and trigonometric functions, etc.

1.4.1 CPU

Since most of the programming languages such as C/C++ are sequential in nature, executing without multi-threading on a single core can result in quite a considerable CPU load. Moreover, these sequential languages are not able to exploit the massive parallel potential that these algorithms process. Take convolution operation for an example. A 3x3 filter convoluted over an input image could be done in parallel since the output of one convolution doesn't affect the result of the other. On the CPU is not possible to achieve this by simple techniques.

1.4.2 GPU

GPUs, on the other hand, are designed to be massively parallel because of their architecture. They are capable of exploiting parallelism by breaking down a complex operation into smaller simpler operations and then utilizing multiple parallel cores to process simultaneously on instances of the input. In this work, we focus on the use of OpenCL framework for General Purpose GPU programming to implement Canny Edge Detector.

2 Canny Edge Detector

Canny filter improves the Sobel Filter by adding some pre-processing operations on the input image before feeding into the Sobel core followed by some post-processing ones. Since these operations are applicable only on grayscale mono-channel images, hence, all 3 channel RGB images need to be converted beforehand.

2.1 Gaussian Blur

This stage of pre-processing applies a Gaussian Filter convolutional operation on the input image. Since mathematical operations involved in the edge detection are mainly based on the derivatives (gradient calculation), the output edges are highly sensitive to image noise. Applying Gaussian Blur is one of the ways to reduce the noise in an image. A Gaussian kernel mask of appropriate size (usually 3x3 or 5x5) is applied and the weights of each element of the kernel is dependent on the gaussian distribution. The following equations describe the relationship between sigma and Gaussian mask calculation [2][1].

$$\sigma = 0.3\left(\frac{n}{2} - 1\right) + 0.8 \tag{2.1}$$

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.2}$$

Fig 2.1. shows the Gaussian distribution with various means μ and standard deviations sigma. As it can be seen, the smaller the value of sigma, the sharper the curve is and less spread out. However, as the value of sigma increases, the distribution becomes more spread out and less sharp. This is a trade-off that needs to be optimized since larger sigma values lead to higher noise reduction but also increase the blur effect which in turn makes the edges less sharp and blurrier visibly. The size of the kernel is dependent on the size of sigma. In this work, the value of sigma is assumed to be 1 and kernel size 3x3 is selected based on equation (insert citation). The Gaussian Blur works by weighing neighboring pixels' intensity according to the gaussian mask and calculating the new value of the pixel as per equation. In this case, it results in the multiplication of the image matrix with 3x3 kernel. This helps to reduce the noise but there is a tradeoff as the sharpness of the image is reduced and the processed image appears to be a bit blurry. Fig. 4.2 (2) shows the effect of applying Gaussian Blur to an input image

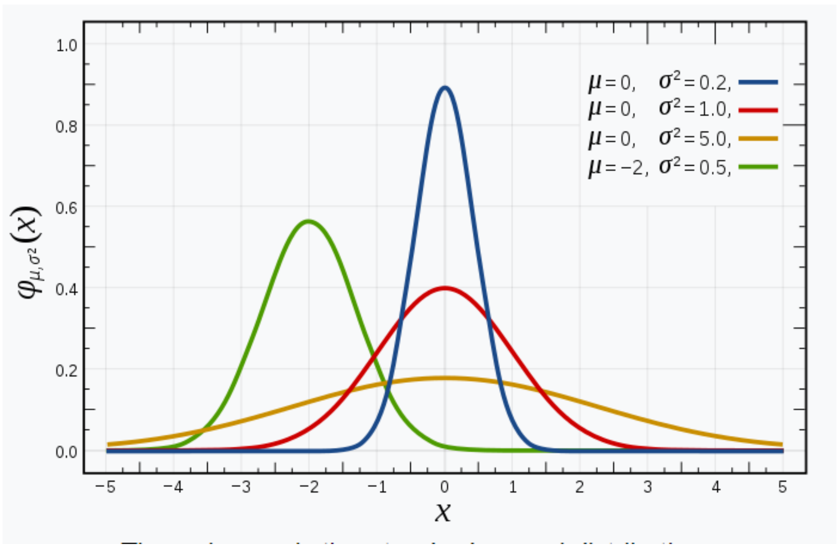


Figure 2.1: Gaussian Curve

2.2 Sobel Filter

Sobel filter determines the edges in an image by using first-order gradients that are calculated for each direction (horizontal and vertical) for a 2-D image (denoted by matrix A). To calculate the gradients, a Sobel filter mask is used (and multiplied with image matrix A) for each direction as given by the following equation.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.4}$$

$$|G| = \sqrt{I_x^2 + I_y^2} \tag{2.5}$$

$$\theta(x,y) = \arctan\left(\frac{I_y}{I_x}\right) \tag{2.6}$$

Since the Sobel kernel itself is 3x3 in size, gradients are summed up in a 3x3 matrix in each direction and the process is repeated over in multiple passes, each sliding by a factor of one pixel, to cover the whole image. Then finally, both directional gradients are combined by taking the square root of the sum of squares of the individual gradient as shown by the formula. Fig. 4.2 (3) shows the effect of applying Sobel Filter to an input image. Apart from the magnitude, we also calculate the direction vector theta by the formula above. This direction vector symbolizes the direction of the edge which pertains to a heading from higher brightness pixel to a lower one, thus the edge runs normal to the theta vector [5]. This shall be useful in further processing steps. The output produced by Sobel filter has a varying intensity of the pixels hence it appears brighter and sharper at edges with higher contrast to the background and appears dull and blurry at edges with poor contrast, thereby resulting in a ghosting effect. Fig. 2.2 [7] shows the possible angle values when applying a 3x3 kernel size, it can be clearly seen that an edge could be in four possible configurations.

1. Horizontal ($\theta = \{0, \pi\}$) or 0
2. Vertical ($\theta = \{\frac{-\pi}{2}, \frac{\pi}{2}\}$) or 90
3. Slanted with positive slope ($\theta = \{\frac{-3\pi}{4}, \frac{\pi}{4}\}$) or 45
4. Slanted with negative slope ($\theta = \{\frac{-\pi}{4}, \frac{3\pi}{4}\}$) or 135

When these values are plotted for each pixel as an image we obtain Fig. 4.2 (4) theta direction.

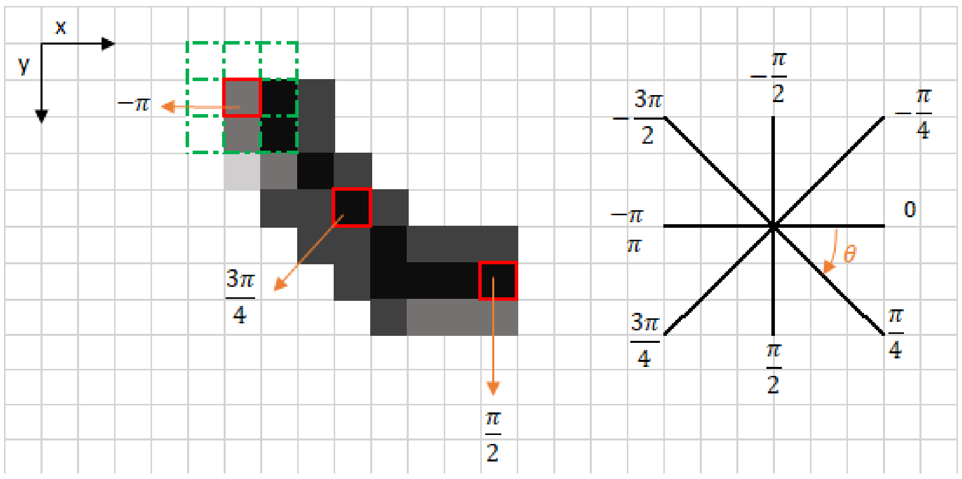


Figure 2.2: Direction Angle Theta

2.3 Non-Maximum Suppression

To fix the uneven thickness of the edges, we apply non-maximum suppression to the edges. In principle the algorithm iterates over all the points on the gradient intensity matrix to find the pixels with the maximum value in an edge's direction by comparing the value of theta. In each iteration of a small kernel size 3x3, for the given theta, neighboring pixels are checked, if they have the same theta, in that direction only the pixel with the maximum intensity is kept and rest all are made to zero.

In fig. 2.3 [7] , the green areas represent the kernel size of the non-max suppression that is iterating over the highlighted section on the image. In this area, it can be clearly seen that some pixel (x,y) such as (i,j), (i,j+1),(i-1,j),(i+1,j+1) are low in intensity as compared to pixel (i,j-1) which is white (255 in 8 bit grayscale format). It can be concluded that the edge runs in the vertical direction passing through (i,j-1). Thus the pixels with low intensity falling in the direction of theta are set to zero ie. (i,j) and (i,j+1). This helps to sharpen the blunt edges and make them only the brightest pixel in width. This helps to make the edges thinner, however, the problem of varying intensity is still persistent.

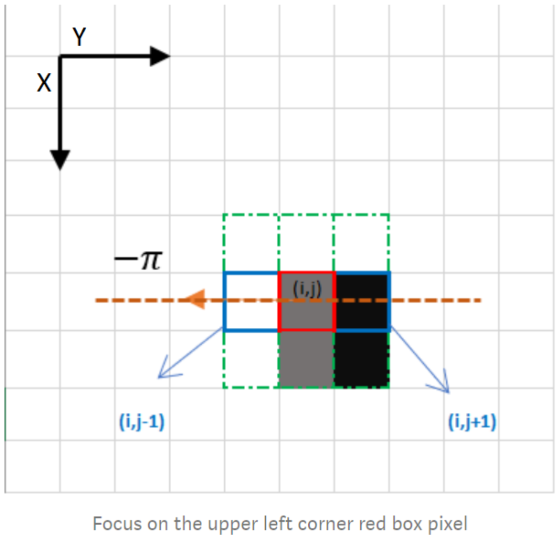


Figure 2.3: Non Maximum Suppression

2.4 Hysteresis

After non-maximum suppression, we have thinned out the edges, but there are still edges with uneven intensity and some stray ones. To solve this problem, hysteresis with thresholding is applied. We use two threshold levels, min threshold, and max threshold to determine 3kinds of edges, strong, weak and noise. Strong edges are those in which the intensity of each pixel is higher than the maximum threshold, this ensures that they are a part of an edge in the image and hence make up the final image. We pull up their intensity to maximum (255) making them white. Noise are those in which the intensity of each pixel is lower than the min threshold. For the noise pixels, we pull down their intensity to the minimum (0) making them black. Weak edges are those where the intensity of the pixel lies between both the threshold values. These pixels need further looking into to decide whether they are a part of any existing edge or are they just noise. Double thresholding makes the processed edge image nearly uniform as all the strong pixels are white and noise is black except for the weak ones which need further processing.

2.5 Edge Tracking

Edge tracking is a technique to trace the edges in the same direction by checking the intensity value of each pixel iteratively along with its neighboring pixel. This ensures that weak pixels which are a part of an existing edge are pulled up to the maximum intensity and the ones left without any neighboring edge are discarded as being noise and their intensity is reduced to zero. This results in the cleanup of the processed image such that no pixel has an intermediate value. All the pixel intensity values are either 0 (black) or 255 (white). As shown in fig. 2.4 [7] , the first case since there is no strong pixel in the vicinity, the red highlighted pixel is set to black (0). In the second case, since there is a strong neighboring pixel, we set the highlighted pixel to strong as well. This process is carried out iteratively over the complete image to get rid of the weak pixels.

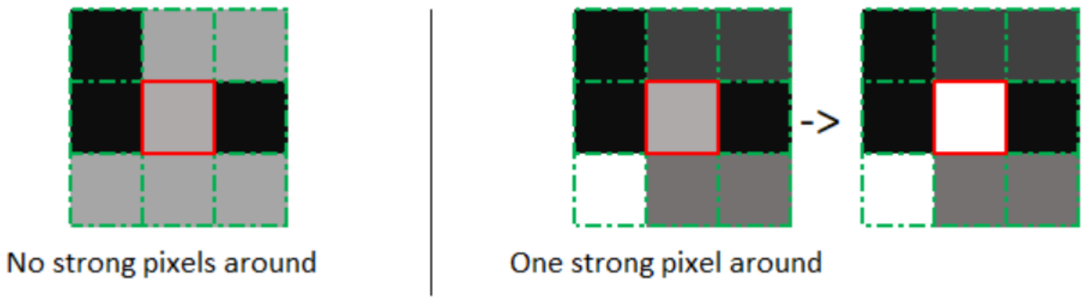


Figure 2.4: Edge Tracking

3 Implementation using OpenCL

The Canny filter is implemented in two phases. Phase One implements the CPU and GPU implementation as an eclipse project on a University pool Desktop that compares the timing performance for both.

3.1 Eclipse Implementation

This implementation takes the input image as a single channel PGM image. The input image is floating number in data type, it is first processed to normalize the image to 8-bit unsigned int values.

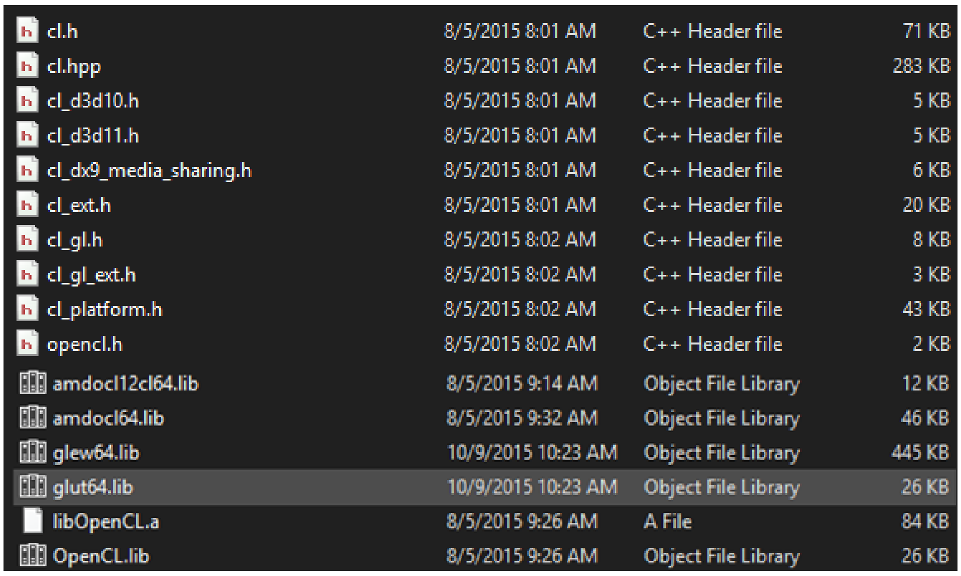
1. One of the limitations of the first implementation is that the input image file, hysteresis threshold values need to be specified at compile time.
2. If a different image is to be selected or thresholds need to be adjusted, we need to compile the code every single time.
3. This not only consumes a lot of time but the chances of introducing bugs and non-deterministic behavior increases.

3.2 QT Implementation

The second implementation extends the first implementation by porting the eclipse project in a QT environment with the added functionality of a GUI. Phase two implements the CPU and GPU implementation as a QT GUI project on a portable Laptop with low power consumption Hardware (Intel Core i5, Nvidia 940MX, 12GB DDR4).

3.2.1 OpenCL - QT Interfacing

Since there is no off-the-shelf OpenCL plug-in or add-on available for QT environment, special provisions were made to integrate OpenCL SDK with QT ensuring that compiler conflicts and dependency issues are taken care of. OpenCL SDK from AMD is used and interfaced with the project by pre-compile directives and includes.To set up the environment, the OpenCL SDK is installed and linked with the qt c++ compiler to add the GPU support. Following list of header files are used by the API as shown in fig. 3.1.



cl.h	8/5/2015 8:01 AM	C++ Header file	71 KB
cl.hpp	8/5/2015 8:01 AM	C++ Header file	283 KB
cl_d3d10.h	8/5/2015 8:01 AM	C++ Header file	5 KB
cl_d3d11.h	8/5/2015 8:01 AM	C++ Header file	5 KB
cl_dx9_media_sharing.h	8/5/2015 8:01 AM	C++ Header file	6 KB
cl_ext.h	8/5/2015 8:01 AM	C++ Header file	20 KB
cl_gl.h	8/5/2015 8:02 AM	C++ Header file	8 KB
cl_gl_ext.h	8/5/2015 8:02 AM	C++ Header file	3 KB
cl_platform.h	8/5/2015 8:02 AM	C++ Header file	43 KB
opencl.h	8/5/2015 8:02 AM	C++ Header file	2 KB
amdocl12cl64.lib	8/5/2015 9:14 AM	Object File Library	12 KB
amdocl64.lib	8/5/2015 9:32 AM	Object File Library	46 KB
glew64.lib	10/9/2015 10:23 AM	Object File Library	445 KB
glut64.lib	10/9/2015 10:23 AM	Object File Library	26 KB
libOpenCL.a	8/5/2015 9:26 AM	A File	84 KB
OpenCL.lib	8/5/2015 9:26 AM	Object File Library	26 KB

Figure 3.1: List of Header Files

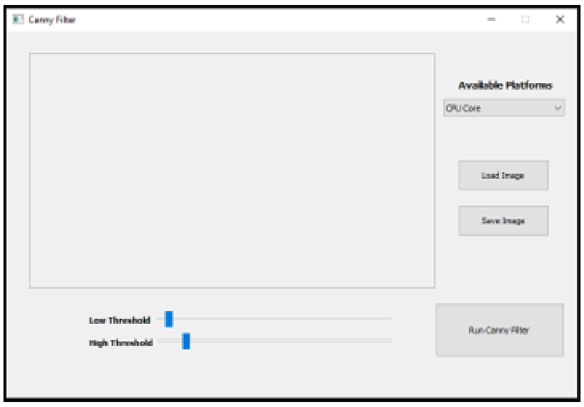
Since phase two implementation does not use OpenCV libraries to read the image, we implemented a custom ReadBMP function which reads .bmp image file from the file-system, iterates over its header to extract the parameters such as height, width, to fetch the image pixel data as an array. All the necessary pre-processing such as conversion to single-channel grayscale, normalization to 8-bit value are taken care of my the algorithm.

1. This enables ease of usage for the user, as the input image need not be specified at compile time as a command to read the image, rather it can be dynamically specified in a File browse GUI.
2. The threshold values for hysteresis and edge tracking can be changed dynamically
3. User can select any .bmp image without hassling about pre-processing.
4. One of the drawbacks of this method is the added library support that is used by qt, but seamless environment setup explained in the previous subsection makes up for the complexity.
5. This makes the code to be compiled once and run multiple times.

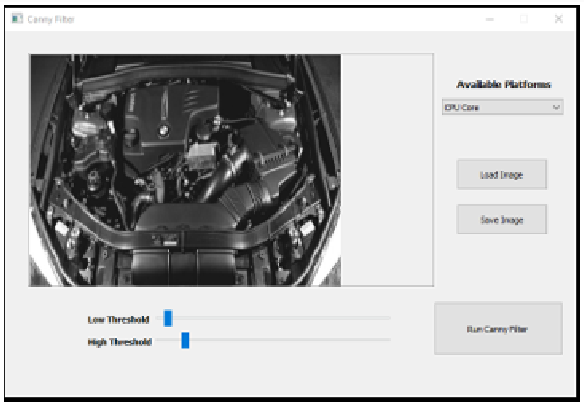
For the Gaussian Blur, 3x3 filter size is used with the weights defined as follows

$$GaussianMask[3][3] = \begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix} \tag{3.1}$$

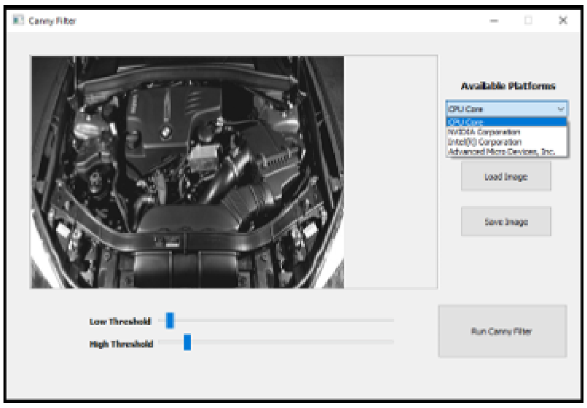
For each stage of the filter, CPU implementation follows an iterative approach, where multiple loops are used to traverse over each pixel and the desired operations are applied. For the GPU implementation, global memory pointers and data buffers are used to copy blocks of input image to local memory with necessary padding to avoid image tearing. Fig. 3.2 shows the different stages of GUI while processing.



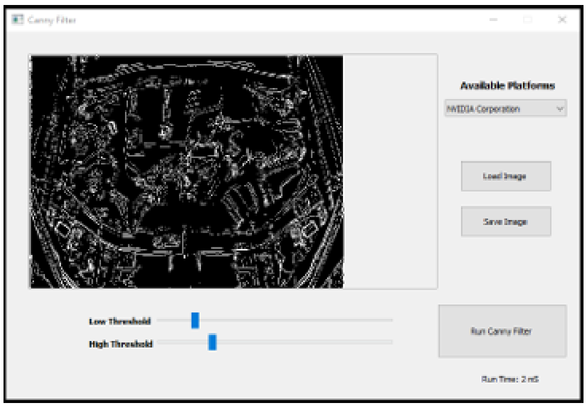
1. Main GUI



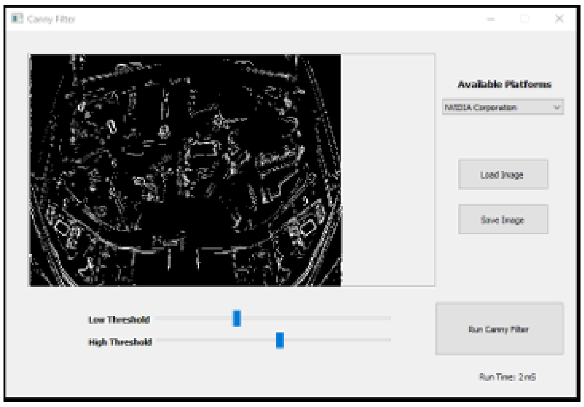
2. Load Image



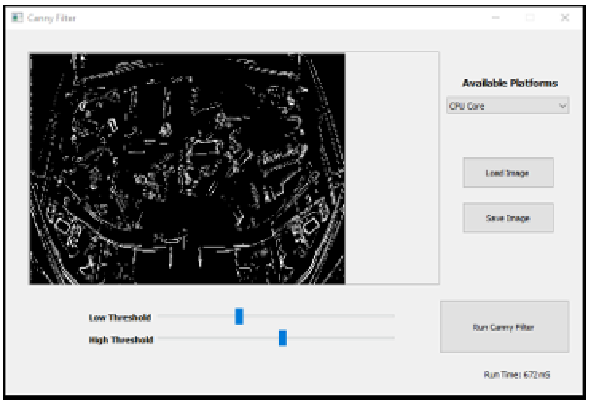
3. Select Platform



4. Tweak Threshold and Run (NVIDIA)



5. Tweak Threshold and Run (NVIDIA)



6. Change Platform and Run (CPU)

Figure 3.2: Different stages of GUI

4 Results

4.1 Performance Benchmarks

Table 4.1: Performance Benchmarks

	Implementation I (Desktop-Eclipse)				Implementation II (Laptop-QT)		
Image	CPU (i7)	GPU (AMD 390)	Speedup		CPU (i5)	GPU (940MX)	Speedup
Valve	0.139 s	0.000663 s	209x		0.633 s	0.002 s	316x
Airplane	0.136 s	0.00071 s	191x		0.507 s	0.002 s	253x
Car Engine	0.141 s	0.000713 s	197x		0.513 s	0.003 s	171x

Table 4.1 compares the performance benchmarks for Canny Filter when implemented on CPU and GPU using OpenCL. It can be clearly seen that massive speedups of around 200x are obtained for both the implementations. The Desktop performance numbers are better than those of the laptop because of improved hardware with higher performance.

Timing of CPU is started as soon as the first stage of filter function is called until the last stage returns the final resultant image. Timing of GPU is started as soon as the input data buffer is loaded with the image until the final processed image is read from the output data buffer.

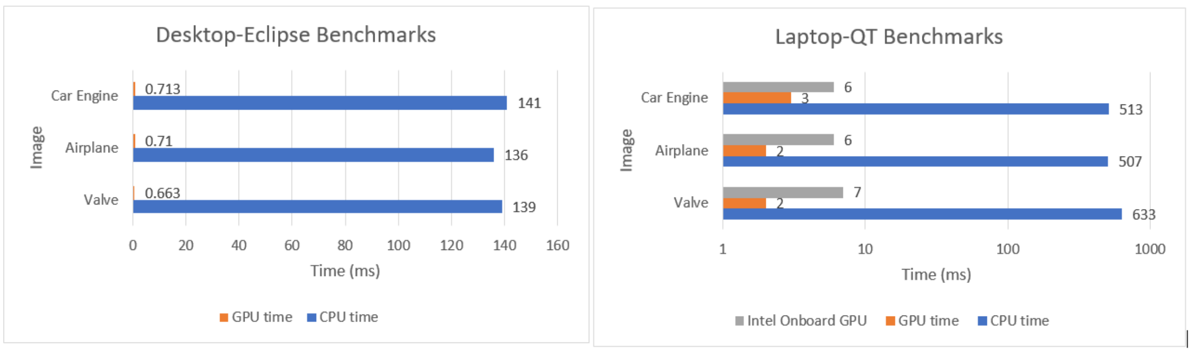


Figure 4.1: Performance comparison

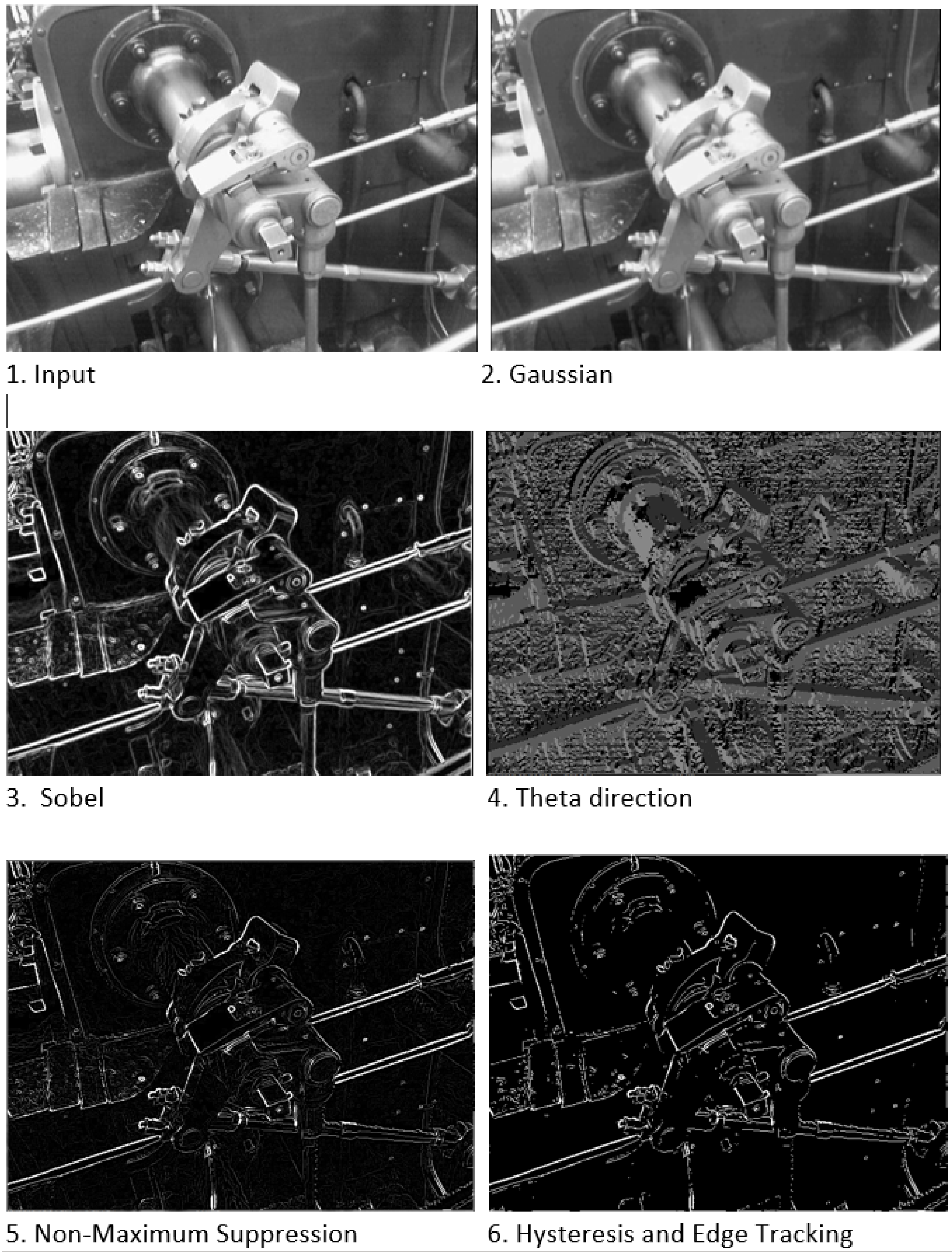


Figure 4.2: Different stages of processing of Canny Filter

5 Conclusion Summary

This project successfully demonstrates that Canny Edge Detector when implemented on GPU results in massive speedup times in execution as compared to when executing on a CPU. On an average speedup of about 200x is achieved on different platforms. However, this number should not be taken as a literate default as the speedup varies from hardware to hardware. The promising conclusion that can be derived is that even when implemented on a low power GPU such as Nvidia 940MX, the speedups are quite impressive. Moreover, the QT based GUI makes up for an easy to use interface for the end-user with interactive button selections and options to tweak the thresholds, select the platform to run and save the output image. This not only increases the usability of the code but also the input image can be changed dynamically since the code doesn't need to re-compile as compared with the Eclipse-based project.

Bibliography

[1] Gaussian Blur https://en.wikipedia.org/wiki/CannyEdgeDetector Accessed: July, 2019. (p. 3)

[2] *OpenCV Filtering Documentation https://docs.opencv.org/2.4/ Accessed: July, 2019.* (p. 3)

[3] Roger D Boyle and Richard C Thomas. *Computer vision: A first course*. Blackwell Scientific Publications, Ltd., 1988. (p. 1)

[4] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986. (p. 1)

[5] E Roy Davies. *Machine vision: theory, algorithms, practicalities*. Elsevier, 2004. (p. 5)

[6] R Gonzales and R Woods. Digital image processing.[sl]. *Addison-Wesley Publishing Company*, 1992. (p. 1)

[7] Sofiane Sahir. Canny Edge Detection Step by Step in Python — Computer Vision. Jan, 2019. (p. 5), (p. 7)