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.

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. 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. One of the most popular techniques of edge detection is the Sobel Filter.

* 1. 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. 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. 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. 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.

2.1 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.

2.2 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.

1. Canny Edge Detector

Canny filter basically improves the Sobel Filter by including some pre-processing operations on the input image before feeding into the Sobel core. There are also some post-processing operations that are carried out on the output generated by the Sobel Core. It is important that these algorithms are applicable only on grayscale single channel images, hence all 3 channel RGB images need to be converted to single channel grayscale beforehand.

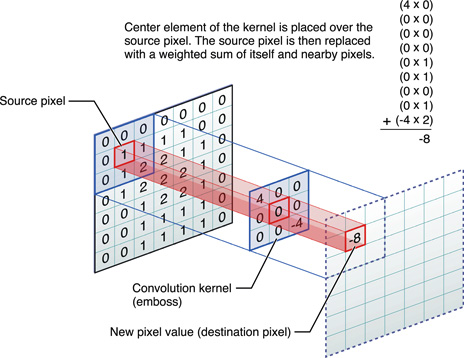
* 1. Gaussian Blur

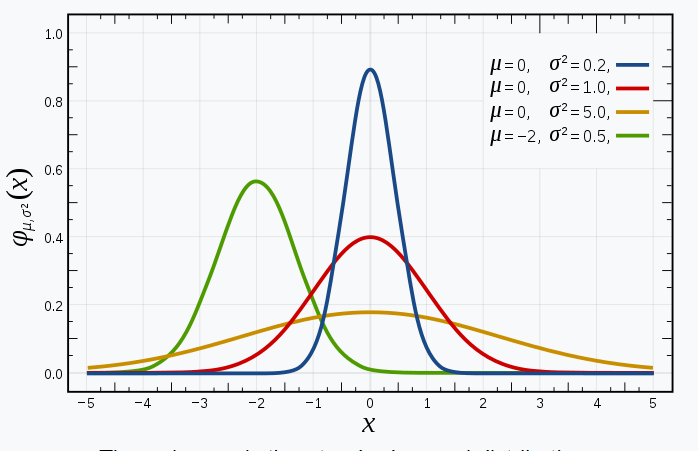
This stage of pre-processing applies a Gaussian Filter convolutional operation on the input image. Since mathematical operations involved in the edge detection are mainly based on the derivatives (gradient calculation), the output edges are highly sensitive to image noise. Applying Gaussian Blur is one 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. Fig. shows the Gaussian distribution with various means u and standard deviations sigma. The square of sigma denotes the variance. 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, value of sigma is assumed to be 1 and kernel size 3x3 is selected based on equation [opencv documentation].

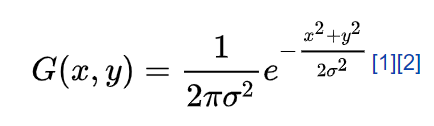


The Gaussian Blur works by weighing neighboring pixels’ intensity according to the gaussian mask and calculating the new value of the pixel as per equation. In this case it results in the multiplication of the image matrix with 3x3 kernel. This helps to reduce the noise but there is a tradeoff as the sharpness of the image is reduced and the processed image appears to be a bit blurry.

The following equations describe the relationship between sigma and Gaussian mask calculation.







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.

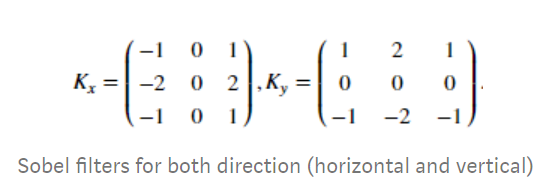
1.1 Gaussian Blur

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. Fig. shows the Gaussian distribution with various means u and standard deviations sigma. The square of sigma denotes the variance. 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 [opencv documentation].

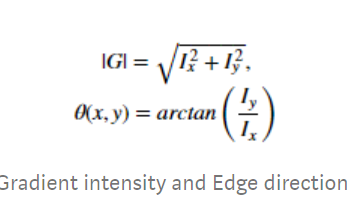
The Gaussian Blur works by weighing neighboring pixels’ intensity according to the gaussian mask and calculating the new value of the pixel as per equation. In this case, it results in the multiplication of the image matrix with 3x3 kernel. This helps to reduce the noise but there is a tradeoff as the sharpness of the image is reduced and the processed image appears to be a bit blurry.

* 1. Sobel Filter

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



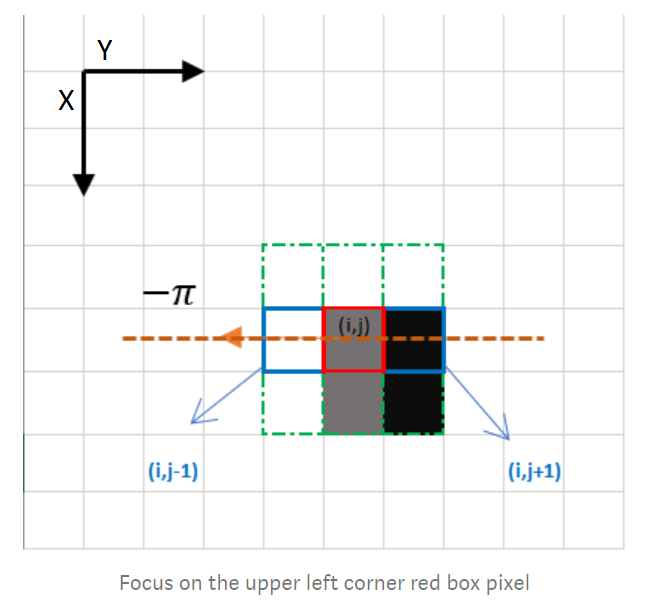
Since the sobel kernel itself is 3x3 in size, gradients are summed up in a 3x3 matrix in each direction and the process is repeated over in multiple passes sliding by a factor of one pixels in order to cover the whole image. Then finally, both directional gradients are combined by taking sqare root of the sum of squares of the individual gradient as shown by the formula



Apart from the magnitude, we also calculate the direction vector theta by the formula. 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. This shall be useful in further processing steps. The output produced by sobel filter has 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.

* 1. 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.



For eg. Take the above image for an example. 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.

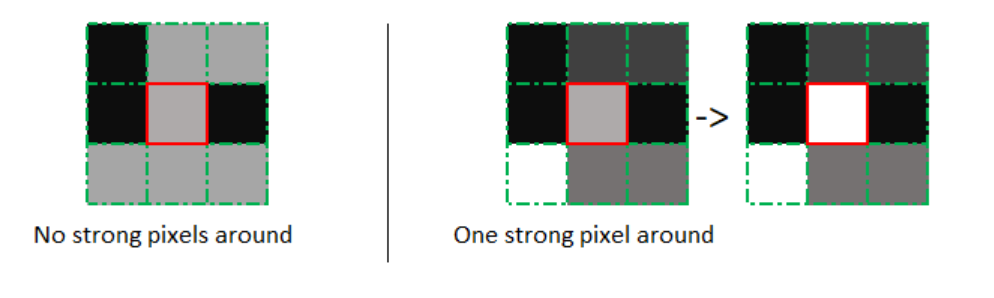
This helps to make the edges thinner, however the problem of varying intensity is still persistent.

* 1. 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 max 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 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.

* 1. Edge tracking

Edge tracking is a technique to trace the edges in the same direction by checking the intensity value of each pixel in an iterative manner 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 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 the above example in 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.

1. Implementation

// Describe the headers of the functions / kernels, what it does, the input/output arguments

The Canny filter is implemented in two phases. One phase implements the CPU and GPU implementation as an eclipse project that compares the timing performance for both. 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.

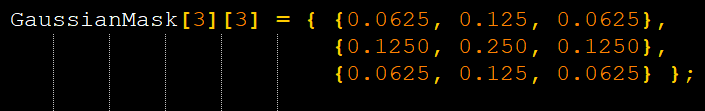
One of the limitations of the first implementation is that the input image file, hysteresis threshold values need to be specified at compile time. If a different image is to be selected or thresholds need to be adjusted, we need to compile the code every single time. This not only consumes a lot of time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

The second implementation extends the first implementation by porting the eclipse project in a QT environment with added functionality of a GUI. This enables ease of usage for the user, as the input image need not be specified at compile time as a command to read image, rather it can be dynamically specified in a File browse GUI. One of the drawbacks of this method is the added library support that are used by qt, but seamless environment setup makes up for the complexity.

This makes the code to be compiled once and run multiple times. To setup the environment, the OpenCL SDK is installed and linked with the qt c++ compiler to add the GPU support.

As explained in the previous section, the entire Canny Filter implementation is broken down in sub-modules.

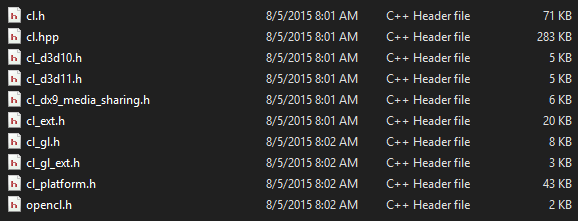
* 1. Gaussian Blur

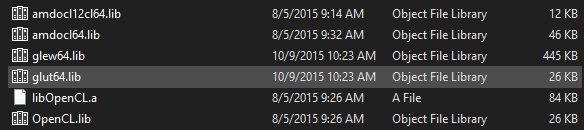


A 3x3 gaussian mask as shown in fig[] is used based on the formula give in equation []. For the CPU implementation, the input arguments are the input and output image as a vector of unsigned char, height and width. We iterate over each pixel and perform the multiplication as per the gaussian mask in a 3x3 region of each pixel. For the GPU implementation, we access the global memory using pointers and copy blocks of image in local memory with necessary padding to perform the multiplication in parallel across each available work item per workgroup. After the Gaussian blur is applied, the processed image is copied to the gauss\_out global buffer, to be used by the next stage

// or add some flowchart if need be for the cpu vs gpu

// add some code line from the cpu and kernel to show the calling API





* 1. Sobel Filter

Sobel filter uses two sobel masks, one for each direction, x and y, as shown in eq(). This filter takes in the gauss-out image as an input argument, and gives out the image of the edges, theta map as an output argument. Similar to the previous stage of the filter, global memory pointers are used to access the input image and post processing, the image is copied to the global output data buffer.

//flowchart of implementation

4.3 non maximum suppression

1. Performance Benchmarks

// add the performance numbers and some graphs

|  |  |  |  |
| --- | --- | --- | --- |
| Image | CPU time (i7 | GPU time AMD 390 | Speedup |
| Valve | 0.139 sec | 0.000663 sec | 209x |
| Airplane | 0.136 sec | 0.000710 sec | 191x |
| Car engine | 0.141 sec | 0.000713 sec | 197x |

The table above shows benchmarks for the performance data for canny filter when run on CPU and GPU for implementation phase I I.e the eclipse project. This implementation is executed on the paspool high performance systems with intel core i7 () and with AMD 390 GPUs.

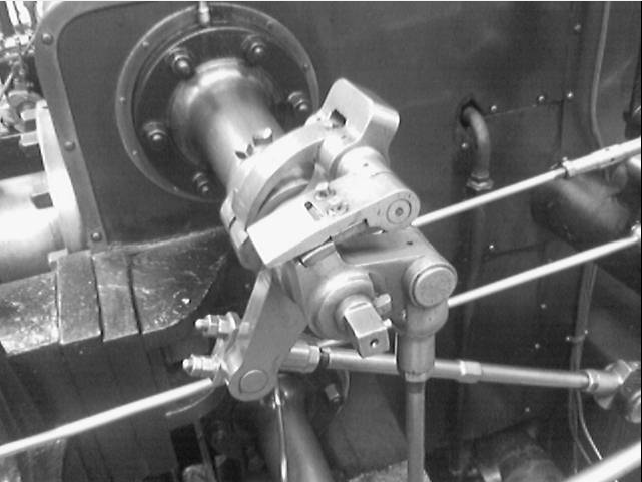
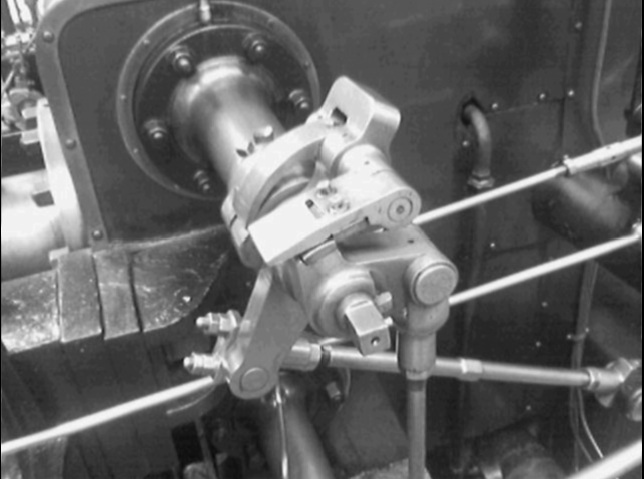
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Image | CPU (i5 6200U) | GPU (Nvidia 940mx) | Intel onboard GPU | Speedup |
| Valve | 633 ms | 0.002 sec | 7 ms | 316x |
| Airplane | 507 ms | 0.002 sec | 6 ms | 253x |
| Car engine | 513 ms | 0.003 sec | 6 ms | 171x |

The table above shows the benchmarks for the performance data for canny filter when implemented in phase two I.e. the qt implementation on a portable laptop with a low power processor and GPU. The processor used is intel core i5 6200u and the GPU is the nvidia 940 mx

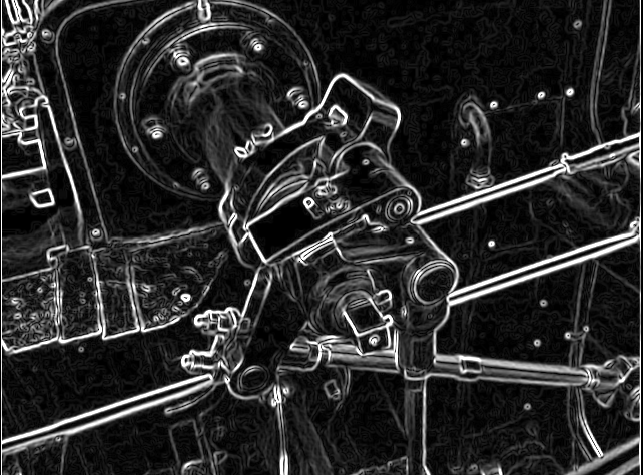
1. Results

// Add images of the results, at each stage of processing along with comparison from CPU implementation

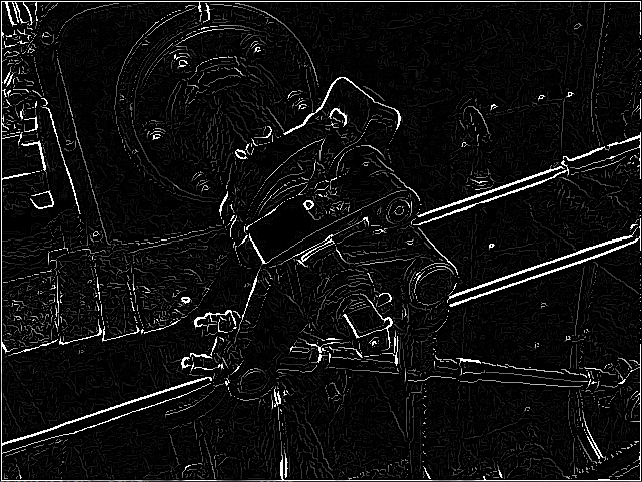
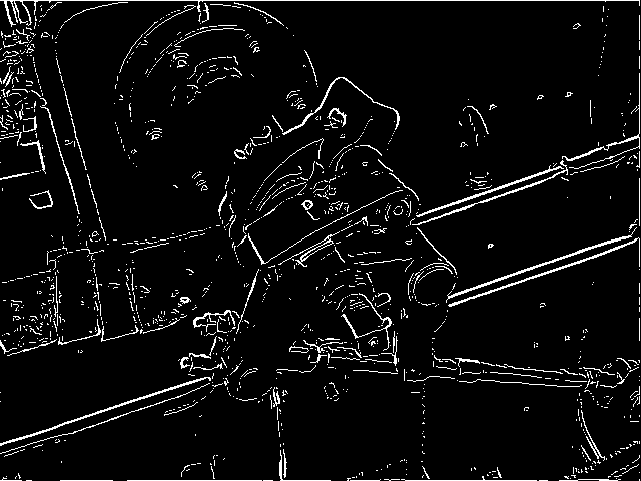
1. Valve

1. Input 2. Gaussian

3. Sobel 4. Theta direction

5. Non-Maximum Suppression 6. Hysteresis and Edge Tracking

1. Conclusion

// add a brief concluding 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.

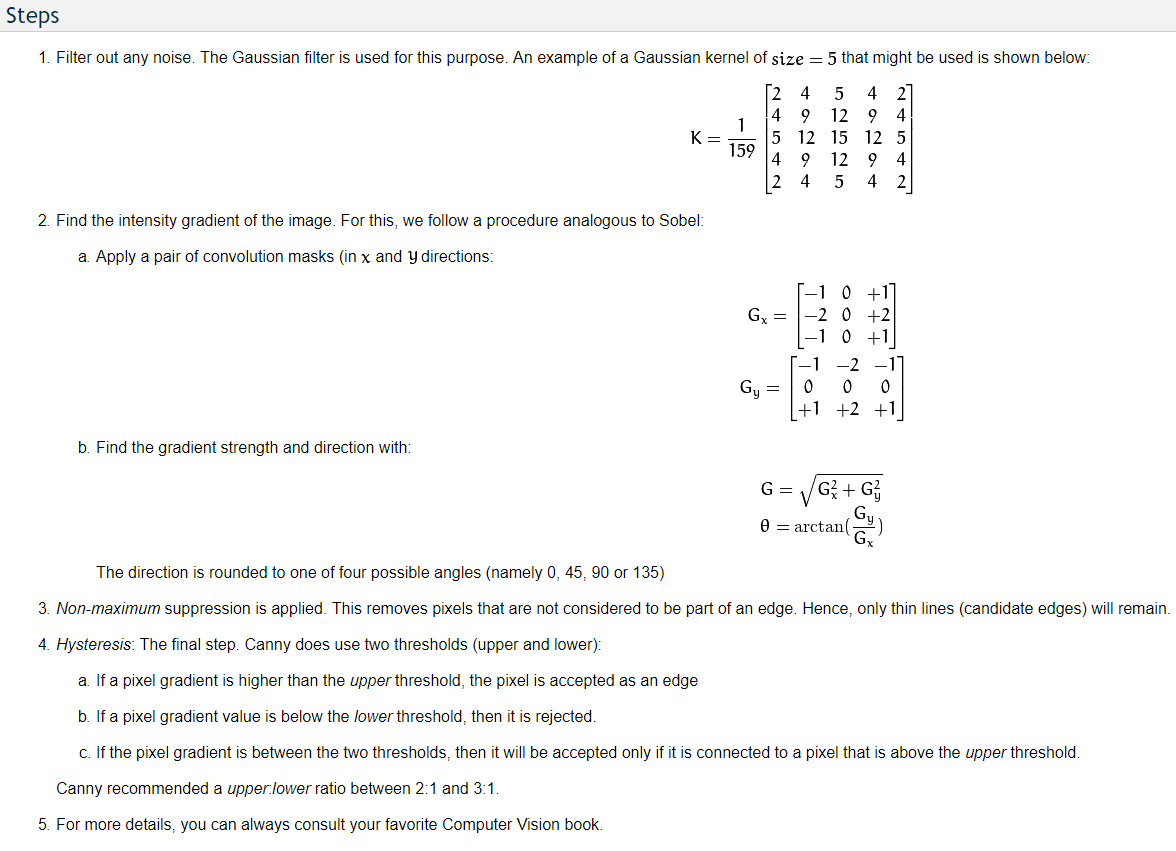
1. References

From OpenCV documentation [2]:

Canny is an optimized auto thresholding edge detector that takes an image and based on local values, determine a threshold to create a single pixel thick edge for you.

1. Low error rate: Meaning a good detection of only existent edges.
2. Good localization: The distance between edge pixels detected and real edge pixels have to be minimized.
3. Minimal response: Only one detector response per edge.

Steps involved:

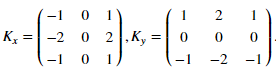


The Canny edge detection algorithm is composed of 5 steps:

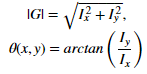
1. Noise reduction
2. Gradient calculation
3. Non-maximum suppression
4. Double threshold
5. Edge Tracking by Hysteresis.

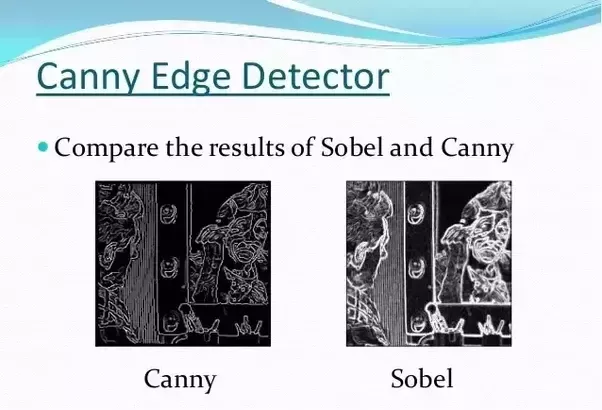
Based on grayscale images. Initial step important to convert the images to grayscale.

1. Noise: Since the mathematics involved behind the scene are mainly based on derivatives (cf. Step 2: Gradient calculation), edge detection results are highly sensitive to image noise. One way to get rid of the noise on the image, is by applying Gaussian blur to smooth it. To do so, image convolution technique is applied with a Gaussian Kernel (3x3, 5x5, 7x7 etc…). The kernel size depends on the expected blurring effect. Basically, the smallest the kernel, the less visible is the blur.
2. Gradient: The Gradient calculation step detects the edge intensity and direction by calculating the gradient of the image using edge detection operators. Edges correspond to a change of pixels’ intensity. To detect it, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal (x) and vertical (y). When the image is smoothed, the derivatives Ix and Iy w.r.t. x and y are calculated. It can be implemented by convolving I with Sobel kernels Kx and Ky, respectively:



Then, the magnitude G and the slope θ of the gradient are calculated as follow:





References

[1] Medium blog for canny filter implementation online: <https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123>

[2] Canny Edge Detector OpenCV documentation online: <https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html>

[3] <http://justin-liang.com/tutorials/canny/>

[4] <https://en.wikipedia.org/wiki/Canny_edge_detector>

[5] Some literary material

R. Boyle and R. Thomas Computer Vision: A First Course, Blackwell Scientific Publications, 1988, p 52.

J. Canny A Computational Approach to Edge Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 8, No. 6, Nov. 1986.

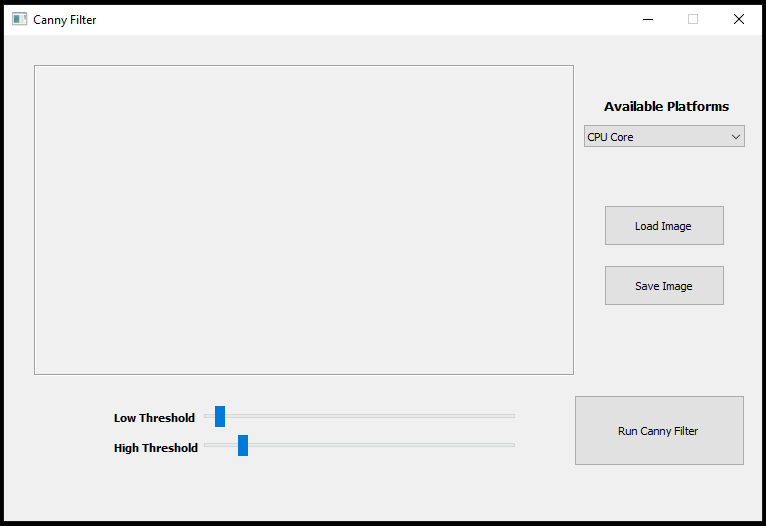
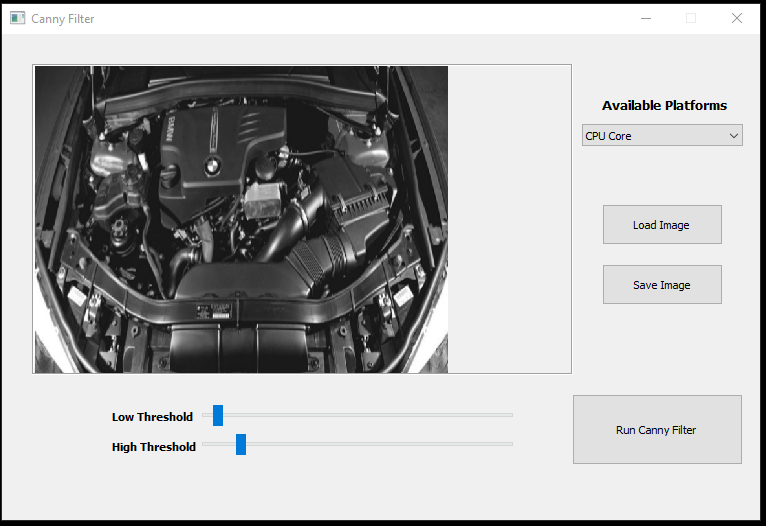
E. Davies Machine Vision: Theory, Algorithms and Practicalities, Academic Press, 1990, Chap. 5.

R. Gonzalez and R. Woods Digital Image Processing, Addison-Wesley Publishing Company, 1992, Chap. 4.

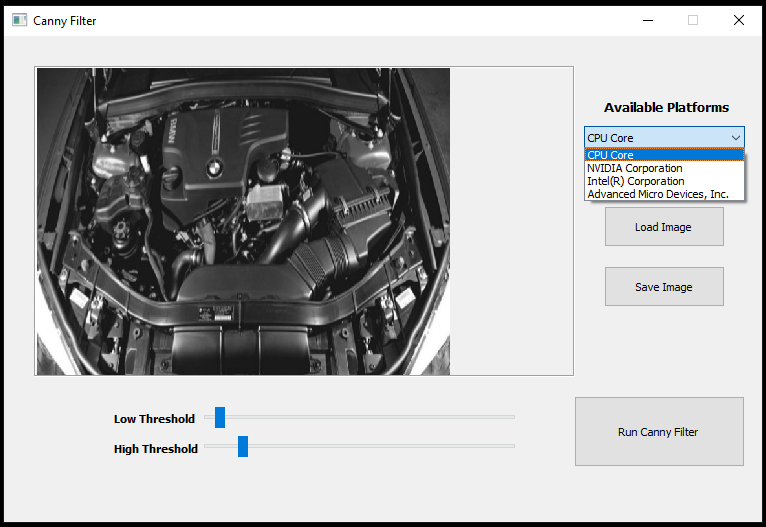
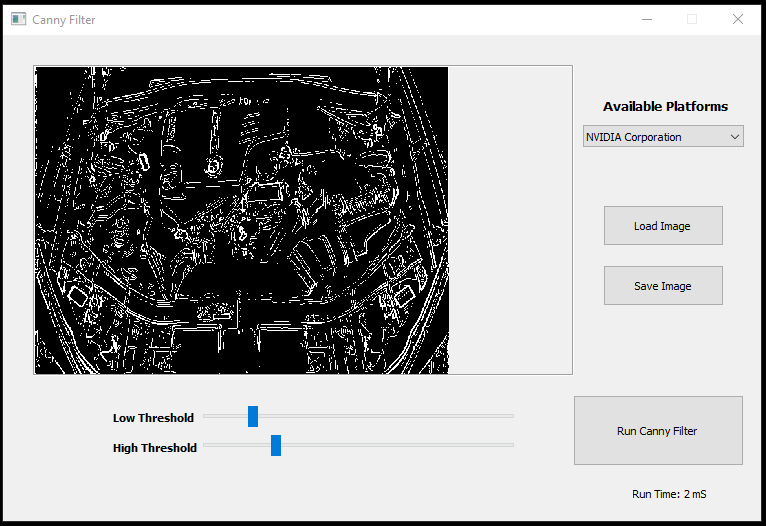
[6] <https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_canny/py_canny.html>

[7] canny vs sobel <https://www.quora.com/What-is-the-difference-between-edge-detection-Sobel-detection-and-Canny-detection>

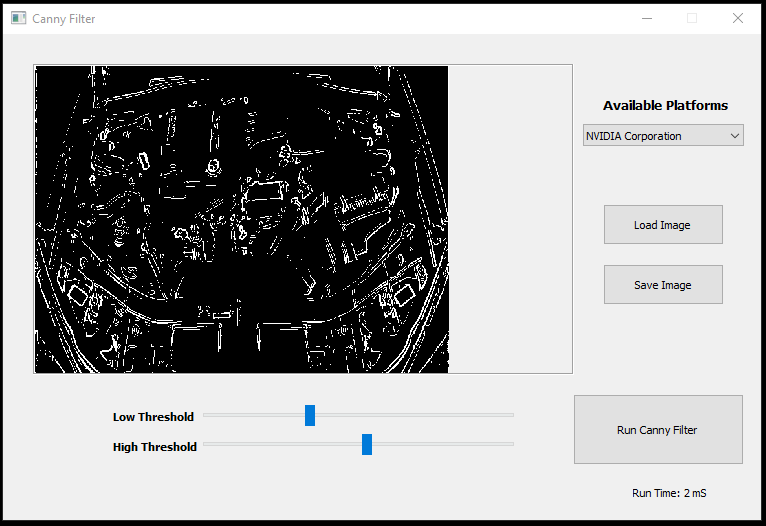
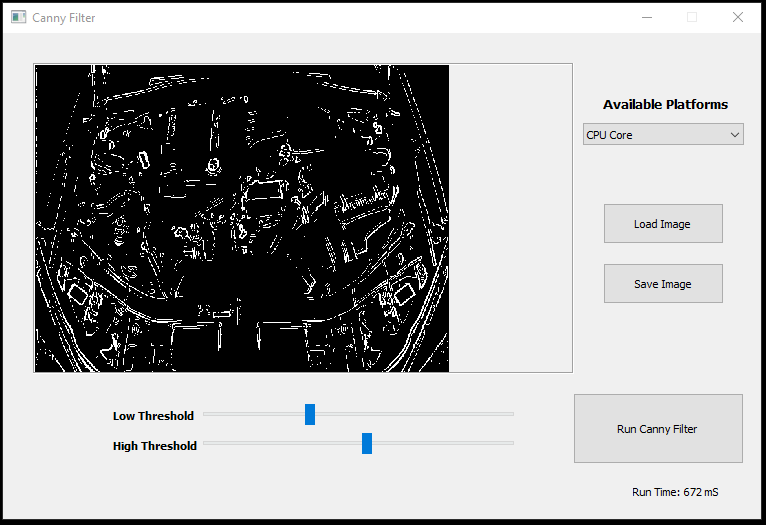
[opencv kernel formula] <https://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html>

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)