**Edge Detection using Sobel and Canny filter on CUDA**

**Ramidi Tarun Reddy and Mallepalli Rakesh Reddy**

**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 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.

**Canny Edge Detection Algorithm**:

The algorithm runs in 5 separate steps:

1. **Smoothing**: Blurring of the image to remove noise.
2. **Finding gradients**: The edges should be marked where the gradients of the image has large magnitudes.
3. **Non-maximum suppression**: Only local maxima should be marked as edges.
4. **Double thresholding**: Potential edges are determined by thresholding.
5. **Edge tracking by hysteresis**: Final edges are determined by suppressing all edges that are not connected to a strong edge.

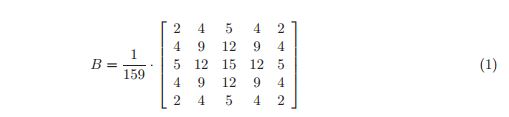
The image in Figure 1 is used throughout this document to demonstrate how Canny edge detection works.



Figure 1: The image used as example of Canny edge detection

**Smoothing**

It is inevitable that all images taken from a camera will contain some amount of noise. To prevent that noise is mistaken for edges, noise must be reduced. Therefore the image is first smoothed by applying a Gaussian filter. The kernel of a Gaussian filter with a standard deviation of σ = 1.4 is shown in Equation (1). The effect of smoothing the test image with this filter is shown in Figure 2.



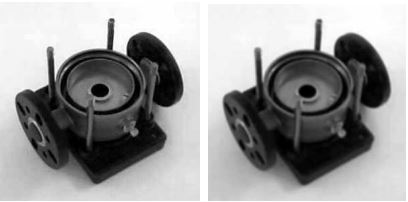
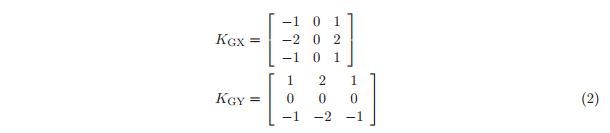


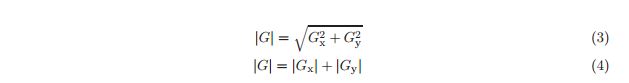
Figure 2: The original grayscale image is smoothed with a Gaussian filter to suppress noise.

**Finding gradients:**

The Canny algorithm basically finds edges where the grayscale intensity of the image changes the most. These areas are found by determining gradients of the image. Gradients at each pixel in the smoothed image are determined by applying what is known as the Sobel-operator. First step is to approximate the gradient in the x- and y-direction respectively by applying the kernels shown in Equation (2).



The gradient magnitudes (also known as the edge strengths) can then be determined as a Euclidean distance measure by applying the law of Pythagoras as shown in Equation (3). It is sometimes simplified by applying Manhattan distance measure as shown in Equation (4) to reduce the computational complexity. The Euclidean distance measure has been applied to the test image. The computed edge strengths are compared to the smoothed image in Figure 3.



Where: Gx and Gy are the gradients in the x- and y-directions respectively. It is obvious from Figure 3, that an image of the gradient magnitudes often indicate the edges quite clearly. However, the edges are typically broad and thus do not indicate exactly where the edges are. To make it possible to determine this (see Section 2.3), the direction of the edges must be determined and stored as shown in Equation (5).



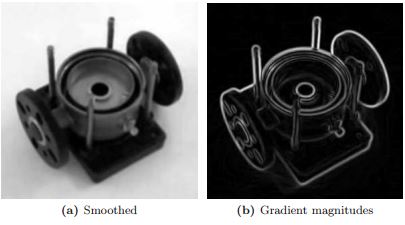


Figure 3: The gradient magnitudes in the smoothed image shown in 3b as well as their directions are determined by applying e.g. the Sobel-operator.

**Non-maximum suppression**: The purpose of this step is to convert the “blurred” edges in the image of the gradient magnitudes to “sharp” edges. Basically this is done by preserving all local maxima in the gradient image, and deleting everything else. The algorithm is for each pixel in the gradient image:

1. Round the gradient direction θ to nearest 45◦, corresponding to the use of an 8-connected neighborhood.

2. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient direction. I.e. if the gradient direction is north (theta = 90◦), compare with the pixels to the north and south.

3. If the edge strength of the current pixel is largest; preserve the value of the edge strength. If not, suppress (i.e. remove) the value.

A simple example of non-maximum suppression is shown in Figure 4. Almost all pixels have gradient directions pointing north. They are therefore compared with the pixels above and below. The pixels that turn out to be maximal in this comparison are marked with white borders. All other pixels will be suppressed. Figure 5 shows the effect on the test image.

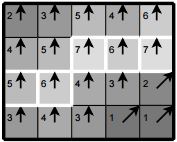


Figure 4: Illustration of non-maximum suppression. The edge strengths are indicated both as colors and numbers, while the gradient directions are shown as arrows. The resulting edge pixels are marked with white borders.

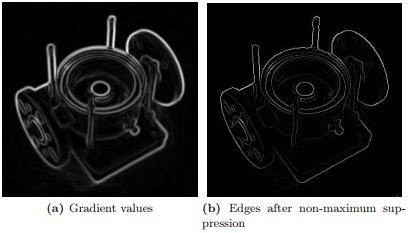


Figure 5: Non-maximum suppression. Edge-pixels are only preserved where the gradient has local maxima.

**Double thresholding:** The edge-pixels remaining after the non-maximum suppression step are marked with their strength pixel-by-pixel. Many of these will probably be true edges in the image, but some may be caused by noise or color variations for instance due to rough surfaces. The simplest way to discern between these would be to use a threshold, so that only edges stronger that a certain value would be preserved. The Canny edge detection algorithm uses double thresholding. Edge pixels stronger than the high threshold are marked as strong; edge pixels weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak. The effect on the test image with thresholds of 20 and 80 is shown in Figure 6.

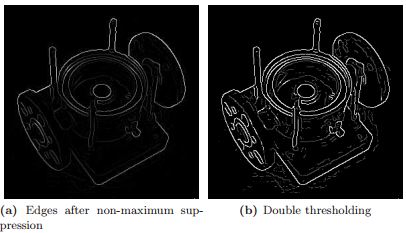


Figure 6: Thresholding of edges. In the second image strong edges are white, while weak edges are grey. Edges with a strength below both thresholds are suppressed.

**Edge tracking by hysteresis** Strong edges are interpreted as “certain edges”, and can immediately be included in the final edge image. Weak edges are included if and only if they are connected to strong edges. The logic is of course that noise and other small variations are unlikely to result in a strong edge (with proper adjustment of the threshold levels). Thus strong edges will (almost) only be due to true edges in the original image. The weak edges can either be due to true edges or noise/color variations. The latter type will probably be distributed independently of edges on the entire image, and thus only a small amount will be located adjacent to strong edges. Weak edges due to true edges are much more likely to be connected directly to strong edges. The effect of edge tracking on the test image is shown in Figure 7.

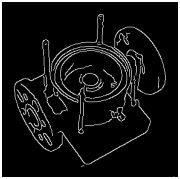
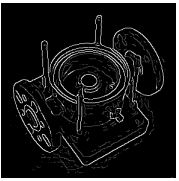
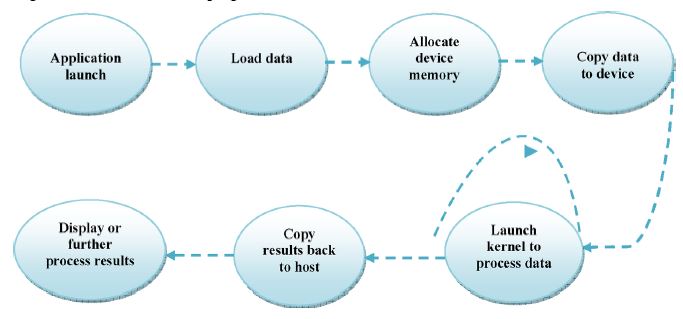


Figure 7: Edge tracking and final output

**GPU Implementation of Canny Edge Detection:**

In this section, we describe implementation of our application on GPU (device) using the CUDA programming language. Most CUDA applications follow a set program flow shown in Figure 8. The host (CPU) first loads data from a source such as a text file and stores it into a data structure in host memory. The host then allocates device memory for the data and copies the data to the allocated space. Kernels are then launched to process the data and produce results. These results are then copied back to the host for display or further processing.

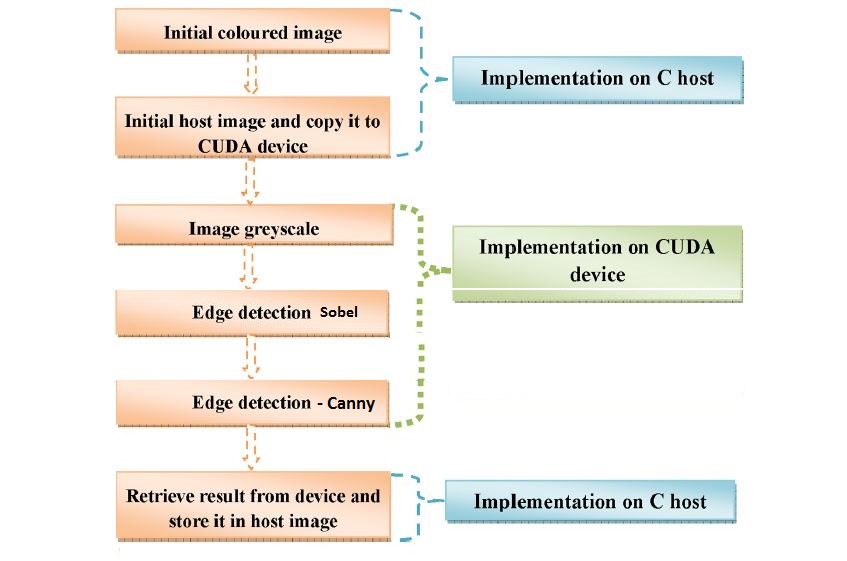


**Figure 8** Common CUDA program flow

‘Convert to greyscale image’ and ‘Edge detection Sobel& Canny’, are the most critical blocks in total execution time, which is why they will be transformed into CUDA.

OpenCV is used to read the input file from disk and to write the output back to disk.

The organization of our application becomes as follows.



**Complexity Analysis:**

Given an M × N image, an examination of a standard Canny edge detector implementation using a Gaussian filter with a 1D window size W pixels, a Sobel operator with a 1D aperture of Y pixels:

|  |  |  |
| --- | --- | --- |
| Functions | CPU Work Complexity | Parallel Step Complexity |
| Gray Scale Conversion | *O(*MN*)* | *O(*1*)* |
| Gaussian Filter | *O(*WMN*)* | *O(*W*)* |
| Gradients via 1-D Sobel | *O(*2YMN*)* | *O(*2Y*)* |
| Non-maximum Suppression | *O(*2MN*)* | *O(*1*)* |
| Hysteresis comparisons | *O(*< MN*)* | *O(*1*)* |
| Gradient Magnitude | *O(*MN*)* (Square root function) | *O(*1*)* |
| Gradient Direction | *O(*MN*)* (Arc-tangent function) | *O(*1*)* |

**Results:**

We compared our GPU Canny implementation with an optimized CPU implementation in the OpenCV library.

|  |  |  |  |
| --- | --- | --- | --- |
| Input File size | CUDA time | OpenCV | SpeedUp |
| 256\*256 | 01.14ms | 01.25ms | 1.09 |
| 512\*512 | 02.39ms | 04.60ms | 1.92 |
| 1024\*1024 | 05.75ms | 15.35ms | 2.67 |
| 1536\*1536 | 10.56ms | 29.62ms | 2.80 |
| 2048\*2048 | 17.97ms | 47.68ms | 2.65 |

*\*Time in milliseconds.*

Hardware used is 2 processor nodes (Intel Xeon x5650 CPU) and 1 GPU (NVIDIA Tesla M2070 GPU) from Oakley cluster of OHIO Supercomputer.

We have demonstrated a version of the complete Canny edge detector under CUDA, including all stages of the algorithm. Absolute runtime of both CPU and GPU algorithms increases proportionally with image dimension NxN. A significant speedup against straight forward CPU functions was seen. The implementation speed is dominated by the hysteresis step .If this post processing step could be made more efficient, and further speedups should be possible using more sophisticated component data parallel algorithms. Our experience shows that using CUDA one can implement complex image processing algorithms on GPU.

**Future Scope:**

For better performance we can improve the use of device memory by using shared memory instead of global memory. To overcome the problem of sharing information across blocks we can allocate memory locations used to store redundant copies of data held by neighboring processes as extra columns. This simplifies parallel algorithm by allowing same loop to update all cells.

We can further improve by keeping all convolution masks copied into texture memory of GPU since texture memory is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.

**Division of Work:**

|  |  |  |
| --- | --- | --- |
| Module | Description | Person Involved |
| Pre Process | Reading image into array using OpenCV | Mallepalli Rakesh Reddy |
| Gray Scale conversion | To convert RGBA image into gray scale | Ramidi Tarun Reddy |
| Gaussian Filter | Blurring of the image to remove noise. | Mallepalli Rakesh Reddy |
| Sobel Filter | Edges are marked where the gradients of the image has large magnitudes. | Ramidi Tarun Reddy |
| Non Maximum Suppression | Only local maxima should be marked as edges. | Mallepalli Rakesh Reddy |
| Double thresholding & Hysteresis | Final edges are determined by suppressing all edges that are not connected to a strong edge | Ramidi Tarun Reddy |
| Post Process | Write output to image file using OpenCV | Mallepalli Rakesh Reddy |
| Time analysis | Capture execution times for both CPU and GPU implementations. | Ramidi Tarun Reddy |

**References:**

1. John Canny. A computational approach to edge detection. Pattern Analysis and Machine Intelligence, IEEE Transactions on, PAMI-8(6):679–698, Nov. 1986.

2. [Efficient implementation of Sobel edge detection algorithm on CPU, GPU and FPGA](https://www.researchgate.net/publication/261959917_Efficient_implementation_of_Sobel_edge_detection_algorithm_on_CPU_GPU_and_FPGA)

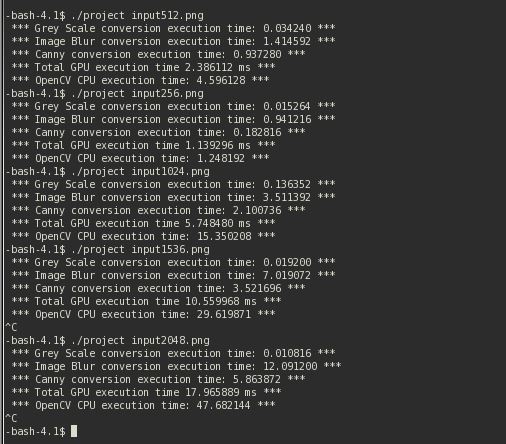
3. [Canny Edge Detection on NVIDIA CUDA, Yuancheng “Mike” Luo and Ramani Duraiswami, University of Maryland, College Park](http://www.umiacs.umd.edu/~ramani/pubs/luo_gpu_canny_fin_2008.pdf)

4.<http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html>

5. <http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf>

**Source Code:** [**https://github.uc.edu/ramidity/6068\_project**](https://github.uc.edu/ramidity/6068_project)



**Presentation Slides:**

