An Improved Canny Edge Detector and its Realization on FPGA

Abounassif, Mahmoud 260261368
Kurdi, Mohamed 260320137
LeBlanc, Bertin 260191026

McGill University
Montreal, Canada

Abstract - **Edge detection is a domain in image processing and computer vision that allows the detection and extraction of features from an image. It aims at identifying points in a digital picture at which the image brightness changes sharply or has discontinuities.**

Edge detection is a computer vision algorithm that is very processor intensive. However optimization could be performed using hardware parallelism. In this project, an edge detection algorithm will be implemented in hardware.

The Canny edge detector is one of the most widely-used edge detection algorithms due to its good performance. However, due to the facts that the algorithm requires a threshold to be set manually beforehand and the algorithm being time consuming, it can't be implemented in real time. To solve the problems, a self-adapting pipelined algorithm was designed.

I. INTRODUCTION

Canny Edge Detector is an edge detection algorithm that is optimal for step edges contaminated by white noise. The self–adopting pipelined Canny edge detector implementation in hardware consists of 3 main steps. The design flow figure below shows the steps of the hardware implementation of the algorithm.

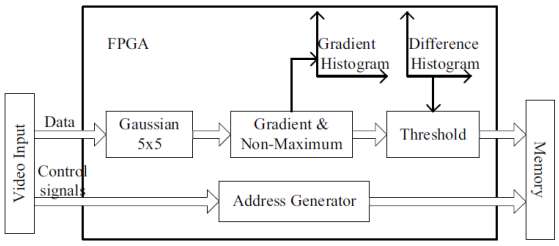


Figure 1: Overall Design

II. SMOOTHING THE IMAGE

The first step in Canny Edge detection is to smooth the image. We do this so that the number of edges detected by the algorithm can be controlled and kept to a reasonable number (only the most salient edges will be detected), by means of a Gaussian filter. If no smoothing is performed, insignificant edges are detected and show up at the output.

First, we required an appropriate data format in which to store our original image. In order to accomplish this, we have written a script in Python using the Python Imaging Library (PIL). This allowed us to read in the image and output its binary grayscale representation (one byte per pixel, representing the intensity). We had the script output this information to a Memory Initialization File (.mif), which allowed us to dump the binary image data into a memory module. Since ModelSim does not accept .mif files, we thus converted to Intel's .hex format using an Altera utility called mif2hex.

Next, in order to implement the various blocks of our design, we looked at Altera's Library of Parameterized Modules (LPM). They have a generic design for a Read-Only Memory

(ROM) as well as for a First-In First-Out data structure (FIFO). We chose the LPM ROM implementation as it has a parameter where one can specify the input .hex file. The FIFO module was used to buffer the smoothed pixels while they await the second stage. This buffer is checked to see that it is not full before writing to it; a state in the state machine is devoted to this.

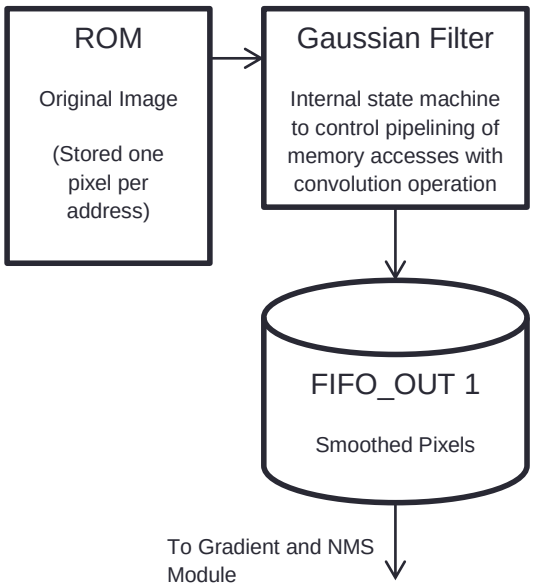


Figure 2: Image Smoothing Stage Block Diagram

Given this memory module where the original image is stored, we need to apply the Gaussian filter to the image. Essentially, the image is streamed out of the memory byte by byte (or pixel by pixel). A 3x3 Gaussian Filter kernel is convolved with a 3x3 frame of the image, giving us a smoothed pixel value. After the calculation for one frame is complete, the output value is written to the FIFO in our design.

An example can be seen in fig. 3 on the right. The greyed out 3x3 matrix represents the filter, while the transparent matrix represents the image matrix. The Hadamard product is taken to get the smoothed pixel value in the top left corner of the image. Note that our image is padded with a border of 0s in order to be able to calculate a product for the edge of the picture. Each of the 9 products are summed together and then divided by a normalizing factor. In particular, our Gaussian filter is specified as follows:

1	2	1
2	4	2
1	2	1

The normalizing factor is required to have the sum of the values at each index in its denominator, which in this case is 16.

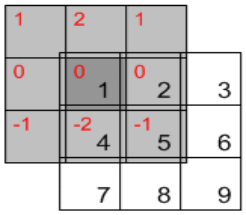


Figure 3: Image Smoothing Example

This operation was made efficient by pipelining the memory accesses with the matrix operations. In particular, the state machine for this part contained the states MULT, DELAY and SHIFT. While the multiplication operation was being performed in MULT, we would buffer a pixel from the next column. We would buffer a second pixel from the next column in DELAY, and read in the last pixel as we were shifting the kernel over to the next frame. In shift, the second column of the frame would be shifted into the first, the third into the second, and the buffered column into the third, from where we would begin the next multiplication operation.

A number of Python scripts were also written in order to reconstruct the image after this stage, as well as the entire system. A VHDL module was implemented in order to write the values written to the FIFO to a file, from which the image could be created pixel by pixel by the aforementioned scripts. Together, these would show the correct operation of the smoothing stage as well as the edge detection algorithm as a whole.

III. Gradient Calculation and Non-Maximal Suppression (NMS)

This stage implements the calculation of gradient and then directional non-maximal suppression operations. Using a { -1,1 } operator with the 3x3 adjacent window of the current pixel, vertical, horizontal, left-diagonal, and right-diagonal, EV, EH, EDR, and EDL, gradients are generated. Then:

$$\left|grads\left(H\left(i,j\right)\right)\right| = \max\left\{\left|E_H\right|,\left|E_V\right|,\left|E_{DR}\right|,\left|E_{DL}\right|\right\}$$

$$\theta = Arg\left(\max\left\{\left|E_H\right|,\left|E_V\right|,\left|E_{DR}\right|,\left|E_{DL}\right|\right\}\right)$$

To compute the gradients, 8 neighbouring pixels are accessed in a signal clock cycle, 2 buffers will be used to store the output pixels of the previous stage.

Next, the pixels that have no local maximum are eliminated. The decision is made by comparing the pixel with its neighbours along the normal to the direction calculated from the equations above. For example, if the gradient orientation is in the range from 22.5 to 67.5 degrees, it means change is occurring in this direction – from the top left corner to the bottom right corner. This means the edge lies from the top right corner to bottom left (the red line). If the values of the top left corner to the bottom right corner are the same, there is an edge at the pixel in the middle.

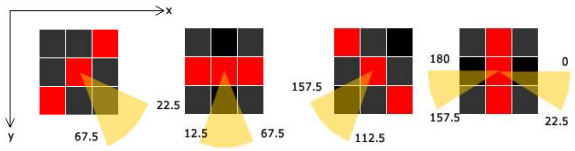


Figure 4: NMS Algorithm

Finally, 2 buffers will also be used at this step since a 3x3 window is needed. The pipelined design of this stage is shown below:

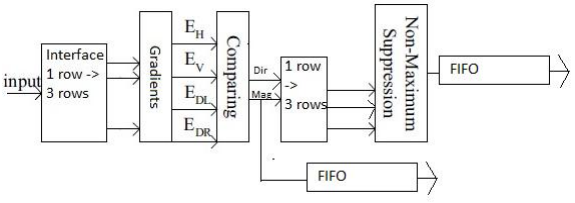


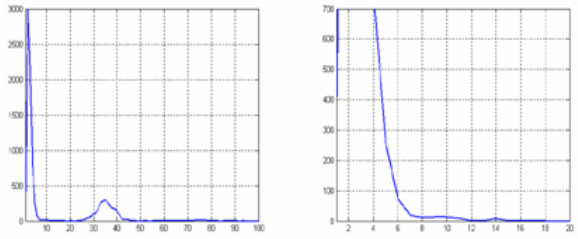
Figure 5: NMS Algorithm

IV. SELF-ADAPTING THRESHOLD EDGE DETECTOR

In the traditional Canny detector, the threshold is set beforehand according to the background-to-image ratio of the image to be processed. This would not be an issue for the purpose of our demo because we are only processing one image. However despite that, we have decided to implement the automated way for setting that threshold, to make our system immune to fluctuations of the background-to-image ratio value that may arise from one picture to another.

Properties of gradient magnitude histogram

Fig. 6a shows one of the gradient magnitude histogram of a ceiling image after the non-maximal suppression done in the previous stage and Fig. 6b is the enlargement of the first part of Fig. 6a. The X-axis is the gradient magnitude, and the Y-axis is the number of pixels with the same magnitude.



(a) (b)

Figure 6: Gradient Histogram

There is a flat part between the background peak and the first edge peak in the histogram. The threshold should be chosen from this part to get minimum error detection.

Self-Adaptive threshold value setting

To locate the flat part more easily, we convert the gradient magnitude histogram into a difference histogram by the following operation:

$$\mathrm{diff(i)} = \begin{cases} \mathrm{abs}\left(\mathrm{NMS(i+1)} - \mathrm{NMS(i)}\right) & \text{if } \mathrm{abs}\left(\mathrm{NMS(i+1)} - \mathrm{NMS(i)}\right) > \mathrm{th} \\ 0 & \text{else} \end{cases}$$

We will choose the gradient magnitude of the first zero-crossing point as the value of threshold. Thl is equal to half of the threshold that we found before.

Implementation

Since the threshold is calculated based on the gradient histogram we need the histogram of the image after the NMS operating.

Taking a 360x280x8bit gray image for instance, the magnitude of each gradient is always between 0 to 100, so we need a registers array containing 100 12bit-width registers to store all of the pixels with different gradient magnitude. A pixel with certain gradient magnitude will arrive at every clock. Taking its magnitude as the address of the registers array, the content of this register is then accumulated by 1. When the end signal arrives, the clock generator circuit generates 100 clock cycles. During these cycles, the content of accumulator 2 which is accumulated by 1 in each clock is set as the address of the registers array, and then the difference circuit calculates the difference between the contents of this register and the next address register. A comparing unit will compare this difference with 0. If the difference is 0, a stop signal will be generated to stop the accumulating of accumulator 2.

The value of accumulator 2 is then the high threshold Thh, and the low threshold Thl will be set the value of the half of Thh. The pipelined design is shown in Fig. 7.

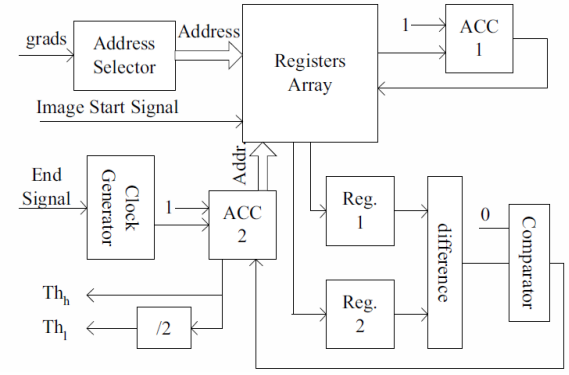


Figure 7: Self-Adaptive Threshold Design

V. TESTING

Because the different modules of our design were independent, we focused on unit testing them, and making sure that each one of them was fully functional as standalone entity. This made our integration testing smooth and facilitated bug detection.

Each module has been rigorously tested through an elaborated test bench. We didn't proceed to integration until all the test cases have successfully passed.

Once, we were confident enough that all the pieces were eligible for proper combination; we finished with an overall integration testing, which turned to be successful. We actually managed to get a decent edge representation of a picture. We will go into more details about that in the results section.

VI. RESULTS

The results were astounding; we managed to get the edges of complex pictures like bellow: Fig. 8a is the original picture of a fish followed by the high weighted edges in Fig. 8b and the low weighted edges in Fig. 8c.

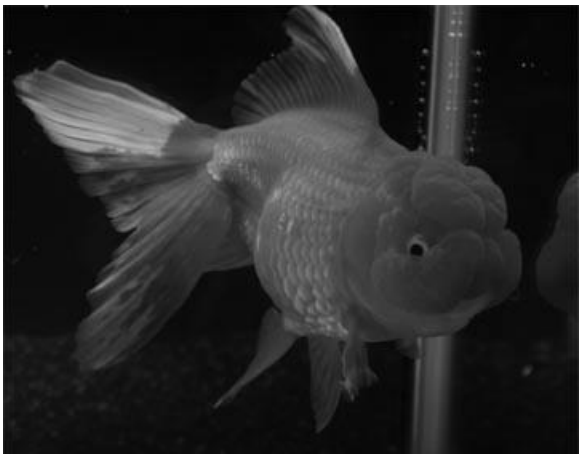


Figure 8 (a): Original

Figure 8 (b): High Weighted Edges

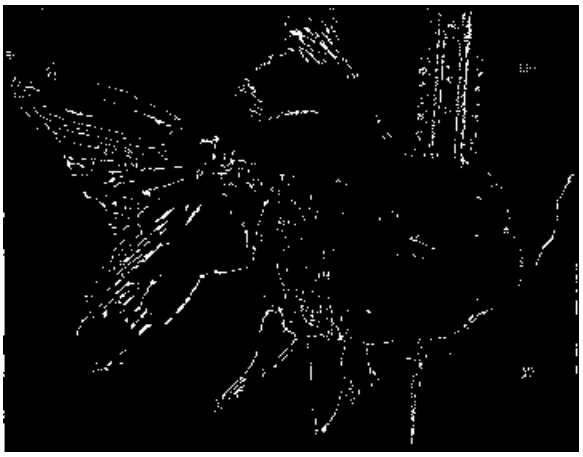


Figure 8 (c): Low Weighted Edges

Synthesis on the Stratix II

Synthesising our project was a big hassle for us; the machines on which we were trying to perform the operation were too slow to support it. Instead, we opted to synthesis the parts individually, and here are the results:

To synthesis the design dealing with an image of size 280x360, 2,400 logic gates were used, and around 1,000,000 registers (image width x image height).

The maximum clock frequency was 80 MHz.

VII. CONCLUSION

In this paper, a self-adapt threshold Canny edge detection algorithm is proposed. In this new Canny edge detector, instead of being set by manual, the threshold can be set automatically by the algorithm itself. Therefore, this new algorithm is more adaptable to the changes of environments and illumination conditions. The effectiveness of the proposed method has been shown by experiments. Although we successfully ran the simulation under 4 ms with 100 MHz clock rate, the synthesis showed that the maximum clock speed that our system could support was 80 MHz which still allows us to finish processing the image in less than 7 ms, making our design still suitable for real time applications. Compared with an implementation in a PC based system, this implementation on FPGA takes much less implementation time and can therefore be used for the mobile robot vision system which is very strict for the real-time performance of its vision system.

VIII. REFERENCES

[1] Wenhao He and Kui Yuan “An Improved Canny Edge Detector and its Realization on FPGA”, Proceedings of the 7th World Congress on Intelligent Control and Automation, June 2008.

[2] Christos Gentsos, Calliope-Louisa Sotiropoulou and Spiridon Nikolaidis “Real- Time Canny Edge Detection Parallel Implementation for FPGAs”, Aristotle University of Thessaloniki, June 2010

[3] Kobzili El Houari and Benbouchama Cherrad “A Software-Hardware Mixed Design for the FPGA Implementation of the Real-Time Edge Detection”, E.M.P.