6.375 Final Project Summary
Implementing Edge Detection Algorithms on FPGAs
By Konstantin Martynov and Angel Carvajal

Introduction

Today's cameras are becoming increasingly smaller and sophisticated. Because of their compactness and resolution, they are suitable for installation on robots, quadcopters, cars, etc. If an individual controls the movement of a high definition camera, they might be interested in capturing a beautiful scene. However, if a robot controls the camera, it doesn't care about the colors – it cares about the shapes and patterns that enable it to achieve its goals. Edge/corner detection is implemented to reduce the amount of information needed to process for a robot to arrive at a decision. Instead of viewing a field of colors, the robot is given only salient data about its environment such as sharp corners or shape outlines. Edge detection filtering can be performed in a number of ways.

One way is to transfer camera data wirelessly to a computer, perform some image processing and transfer a command back. This procedure may be very slow – it requires processing on a computer and transferring data from robot to computer. An attractive alternative way is to do all the processing on an FPGA placed on the robot. This method offers a number of advantages:

1. The filtering performed on a FPGA will be much quicker than the processing done on a computer. It can also aid in real-time processing of images coming from a camera.
2. We avoid transferring data back from a computer, which saves both time and energy, since wireless transferring usually takes more energy than processing.
3. FPGA can implement further processing necessary for the control of the robot.

Our project will implement a Sobel and Canny Filters for HD images on FPGA. A Sobel filter finds edges of the images by applying 2D convolution with a 3x3 kernel, which is an expensive operation on a computer. Optionally, a smoothing filter can be implemented to remove high frequency noise that'll produce artifacts in the final product. The filtered image could be used by a robot for further processing and path correction. Using high definition images increases the probability of correct planning by providing more data points.

Using FPGA for image processing could provide orders of magnitude improvements to the speed of processing, which may be important for quick moving robots.

Background

I. Sobel Filter

Sobel filter is the filter that helps to detect edges on images. Using this filter robot could make decisions on movement automatically.
This filter is basically gradient map of an image. The image could be thought of as a hill with varying height that represent brightness at given coordinate. Therefore the variance in brightness related to the slope of the hill. If derivative is applied to the height with respect to the coordinate, then the derivative is higher at steeper parts of the hill. This analogy makes it clear how Sobel Filter works – by applying gradient filter to the image it enhances the edges and suppresses slowly varying parts.
Gradient filter is achieved by using convolution of an image with two Sobel Kernels (Fig. 1). Since images are 2 dimensional objects, to find their full gradient field in Cartesian coordinates, knowing horizontal and vertical gradients is enough.

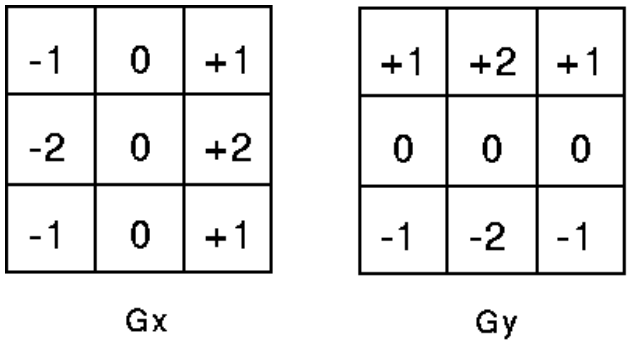


Figure 1. Kernels for horizontal (Gx) and vertical (Gy) gradient fields.

Alternatively output of Sobel filter may be represented as a magnitude and angle (direction) of gradient.

II. Canny Filter

Our Canny Filter is operating in a very strict fashion to ensure we are producing the right results at the output, once simulation is working properly; then we can use short cuts. It operates on 6 stages and has a register that keeps the stage value.
Stage: Firstload, Secondload, Load, LastLoad, Suppress, Output.

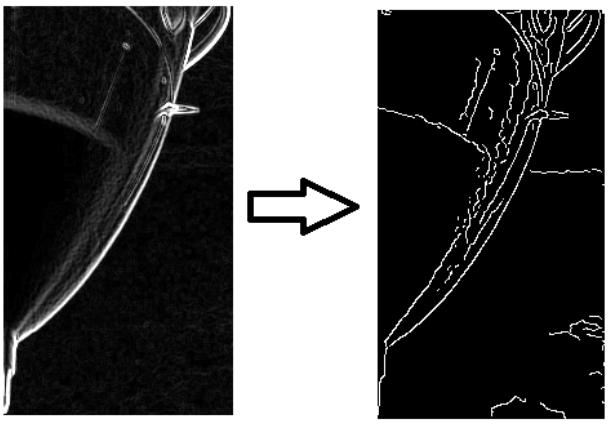
It uses a server interface, allowing an input of a pixel line worth of augmented pixels (pixels that have their magnitude attached with their angle values offered by Sobel). It outputs a pixel line worth of regular pixels for the output of the filter overall. A test bench user can input 4 parameters into the module: height of image, width of image, lower threshold and higher threshold. These parameters then alter the sizes of the instantiated elements in the module to accommodate the size of a given image and what levels the user wants to see. The threshold levels are 8-bit values that are used to filter out final pixel values.

We instantiate a vector for our output and 3 augmented pixel vectors in preparation for the calculations that will occur within the module. They are 2 pixels longer than the width for extra pixel padding at the ends so we can cover the whole image.

Our first stage, FirstLoad, loads up our first 2 extended-APixel vector with the same value. We should be receiving a line of Apixels from Sobel. We send the first value of that line to the first and second value of our line, and the last value of that line to the ultimate and penultimate value of our line. Then we fill in the rest. This is how line filling works in this module. We change the stage to SecondLoad.

In our secondload we just load up the third register with the second line of the image given by sobel. Then we can proceed to our suppression stage. In the suppression stage we use a function defined as suppress to perform non-maximal suppression and thresholding. The function works by receiving 9 values, 3 from each register, and using the middle values angle in a case statement. Angles 0,7,3,4 indicate gradients going from left to right/right to left so we'd like to check the values perpendicular to it to find maximums. What follows are a number of comparisons of magnitude and threshold values. If the middle pixel's magnitude is greater than its perpendicular surrounding and within the threshold, then that pixel will turn into its maximum value: 255. If not, it gets zeroed. Angles 1,2,5,6 go up and down and so a similar computation applies to them.

After the suppress stage, we go into our output stage where we just rip off the angle appended onto this pixels and release an outline vector or image-width. Then we go into the normal load stage which loads a line into reg3 and shifts the values from reg3 to reg2 and reg2 to reg1. This process continues until the last load (there's a counter indicating how many lines we've outputted) and that will conclude all the register loads into the module. Suppression happens again and then output and it's done. Expected results are shown below.



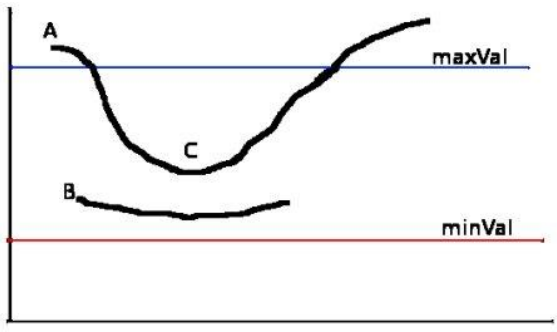


Figure 2. Result of applying Canny filter to image processed with Sobel filter.

Figure 3. Example of applying Canny filter to 'Lena' image.

III. High Level Design On computer

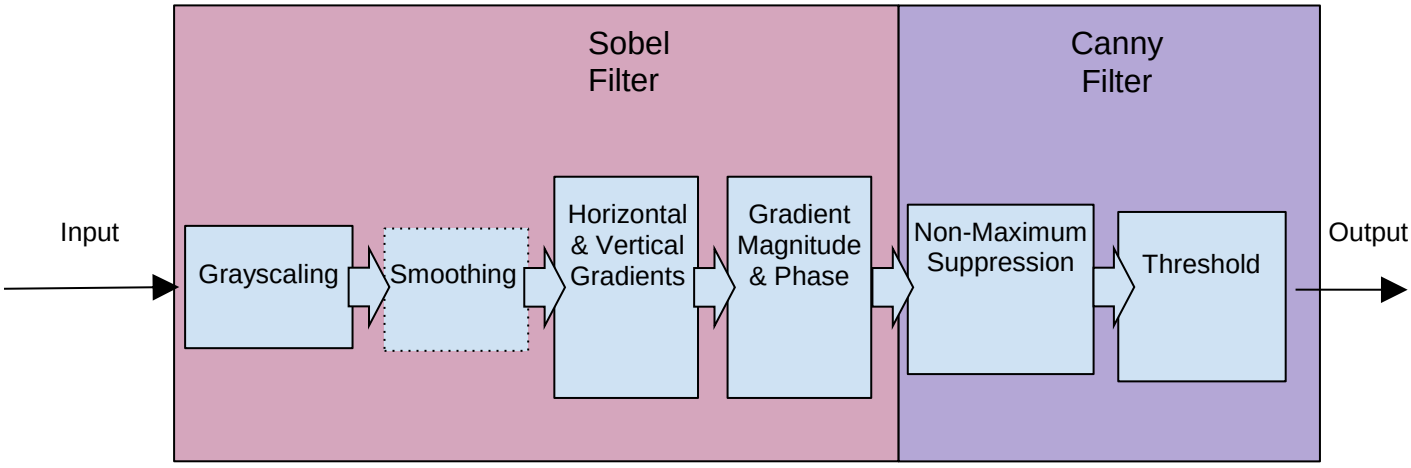


Figure 4. Typical Design of Canny filter in software. In our hardware implementation grayscaling and smoothing are skipped.

Typically on a computers Canny filtering is done in the following order: First a Grayscaling of the image to reduce pixels to 8 bit values, then a Gaussian blurring to remove high frequency noise. The gradient extraction from sobel follows and leads into Canny. Canny will perform non maximal suppression to thin out lines and perform a threshold standard to remove unwanted gradients. Non-maximal suppression works by looking at the angle measurements and looking at the line of pixels orthogonal to that direction and seeing if the pixel is a local maximum along those lines.

IV. Test Plan

The most common way of comparing two images is 2 dimensional cross-correlation, which is calculated using the following formula:

$$r = \frac{\sum_m \sum_n (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_m \sum_n \left(A_{mn} - \bar{A}\right)^2\right)\left(\sum_m \sum_n \left(B_{mn} - \bar{B}\right)^2\right)}}$$

This formula uses mean values of image matrices, therefore if we had 2 similar images, but one had a uniform increase in brightness, we will still have correlation coefficient equal to 1. For human vision, images look similar if they have correlation coefficient higher than 0.9.

However, there several other ways of comparing the work of algorithms. One different way to check that the algorithm works as expected is to implement the same algorithm on computer and check that FPGA produces exactly the same answer. This testing is very convenient after each optimization, because simple cmp function in terminal window can be used to verify the code. The correlational testing can be used for verifying that compression is still not below the desired level. Correlation can be calculated both for computer and FPGA algorithm, since they are expected to produce the same answers.

Approximations needed for FPGA implementation

I. Magnitude of gradient field

After applying both Gx and Gy sobel filters on an image we have both horizontal and vertical derivatives of image. A common way to get the magnitude is following :

$$I = \sqrt{I_x^2 + I_y^2}\,.$$

However, squaring each pixel, summing and taking square root is a very expensive in hardware. Since this result is passed into Canny filter, the precise values of magnitude does not matter, rather their relative values. Therefore, the following approximation could be done :

$$I \approx |I_x| + |I_y|$$

This implementation is much easier, faster and requires much less hardware, since only addition is needed and there is no need to convert values into floating values.

II. Direction of gradient

In software applications to get the directions of the gradient, typically function atan2(Iy,Ix) is used. This function is also is very expensive in hardware.
One typical way to implement it is to use lookup table and get the value depending on the ratio between Iy and Ix. However this way requires division, which is not very cheap in hardware and also lookup table, that requires space.

However, since each pixel is surrounded by 8 neighboring pixels, there is 4 possible gradient directions – horizontal, vertical, and 2 diagonal. Ideally, for horizontal direction the angle should lie within [-22.5, 22.5) or [157.5, 202.5) etc. In other words each direction has an angle opening of 45 degrees in opposite directions.
Checking those precise ranges of angles may be complicated too. However, it may be mentioned that $\tan(22.5) \approx 0.4$. Therefore simple if statements could be used to check the direction :

$$if\ abs(5 * I_y) \leq abs(2 * I_x) => 0$$
$$else\ if\ abs(2I_y) \geq abs(5 * I_x) \quad => 2$$
$$else\ if\ Ix\ and\ Iy\ same\ sign => 1$$
$$else\ 3$$

This implementation is very cheap, since it requires only comparators and multiplication by 2 and 5. The most expensive operation is multiplication by 5, but otherwise it is very cheap.
The error of approximation is very small. Instead of 45 degree openings, there 43.6 and 46.4 degrees. This is not a large price for that much simplification of the hardware, since it is not high probability of angle falling on the area between two different directions.

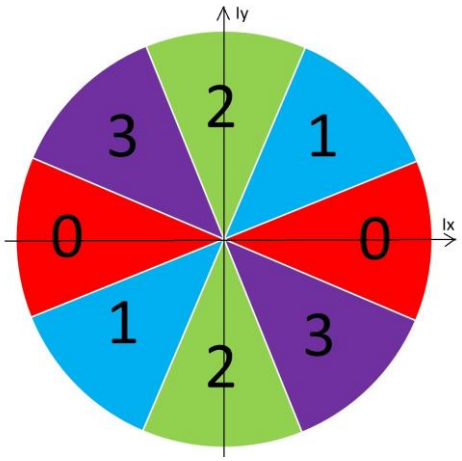


Figure5. Four different possible directions of gradient depending on the relative values of horizontal and vertical gradients.

FPGA Implementation

I. High-Level Design

Software side:

For the software behind scemi, we used the example as a template. Instead of sending numbers from 1 to 256 however, we were sending 4 bytes at a time (32 bits because that's what scemi is comfortable with). We open the file in.bin, which contains our image produced by matlab, and we also open up a textfile, output.txt for the output of the software. We created a buffer value that was of type unsigned int so it was 4 bytes and created a for loop. The loop goes on for every 4 bytes in the bin file. Because of that we also had to calculate the size of the bin file using a stat function. We used the size method and arrived at the size through that. So now our for loop would run and quickly deposit 32 bits/4 bytes/4 pixels into our hardware side. Once we reached the end of the input file, we included an if statement that would close the file. Closing the file was essential or else we would get ugly segmentation faults.

In order for designs in the FPGA to complete, scemi needs to give the completed signal so we also need to keep a count of how many pixel values scemi has received back. This is the only thing that changes in software through different size implementations of the module. In the out_cb method we have response count that goes upto the amount of 4 pixel quantities in the picture (a 128*72 image has 9216 pixels so respCnt would be capped at 2304). Once the cap was reached, then scemi will close the output.txt file and send a "Done" message over the hardware side and finishes up fpga operation.

During the operation of the hardware, response counts are going up and scemi gets information a pixel at a time. We store this value as "a", and perform an if-statement. If a is 255

then we output “11111111” to the outfile, and if a=0, then we print “00000000” to the outfile. This is just printing to the output.txt so matlab can express it as an image once again.

Wrapper Module

We produced a wrapper module that will serve as the interface between our filter and the dram controlled by the dut module. The reason for this is that the dram contains 4 pixels in every page, so we need a module that will collect the pixel information until we get enough for a line of pixels so we can input it into our filter. So it takes in 32bits and outputs a vector of image width size of pixels.

It has a constructor rule which pulls from its input fifo 4 pixels. Next we start a counter that goes up to the width of the image divided by 4 (bit shifted twice). Everytime the counter is not W/4 we continue loading a vector of pixel width with 4 pixels. Once we reach W/4 then we can send the pixel line off into our filter.

The output of the wrapper module is just a pixel line, it gets it from the filter module output and it sends it over to the top dut module.

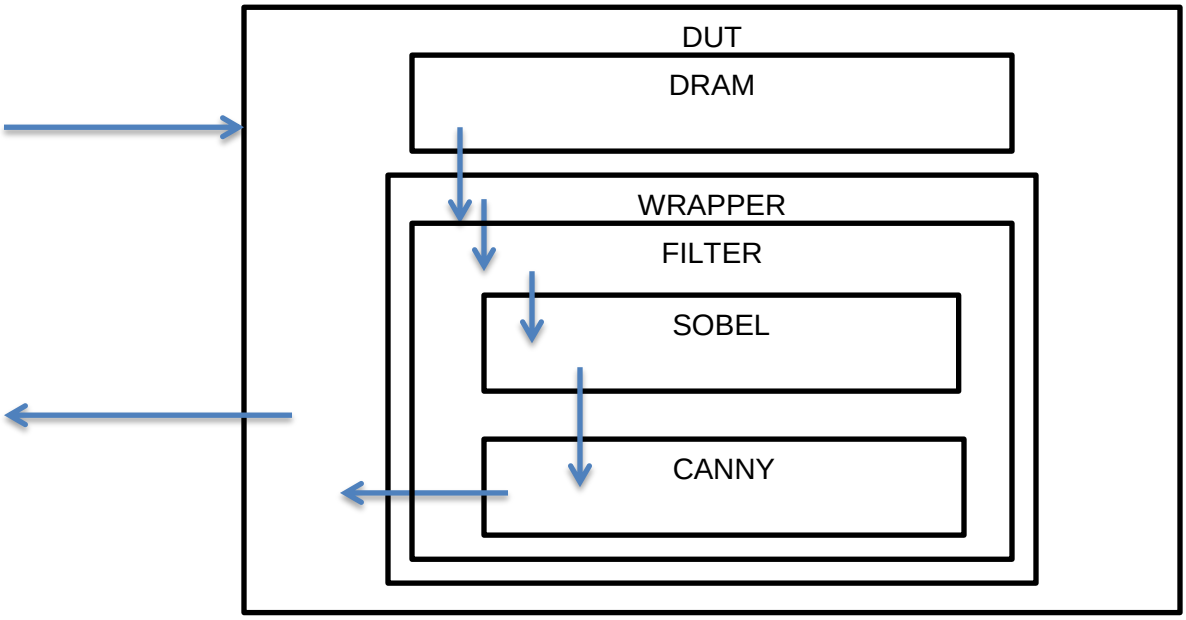


Figure 6. Architecture of the project on FPGA

Scemi to DRAM: 4 pixels
DRAM to WRAPPER: 4 pixels
WRAPPER to FILTER: W pixels
FILTER to SOBEL: W pixels
SOBEL to CANNY: W augmented pixels
CANNY to FILTER to WRAPPER to DUT: W pixels
DUT to SCEMI: 1 pixel

Filter Module

Our filter module was easy to implement because it was practically our testbench module for testing the integration of the 2 filters. It instantiates versions of Sobel and Canny modules and runs them. The Sobel module requires some latency adjustments so we included a counter "sobel count" that will repeat the sobel stage and then start doing the canny stage. Sobelin rule gets a line from the filter input and just request.puts it into sobel. The canny rule transfers over the gradient and angle values from sobel into an Apixel value; just a struct created for convenience. It has a magnitude value and a angle value. We then put this line of augmented pixels through Canny. The output stage just reads the canny response and sends it over to the filter output. We then add 1 to a line number count and once the line number count reaches the height -1 (due to starting at 0) then we go to a close stage where we stay for the rest of operation.

DUT Module

Very similar to the example by Ming. The biggest difference is the outplacerule. This was created to place pixels back to scemi so they can be written into the output file. There were a couple issues with output multiple pixels at a time, so we had to design a 1 pixel output at a time system. This means we would read from an fifo, but only deq once we got the whole cluster of pixels out. This is probably the most inefficient part of the design, but the get outputs without debugging forever, it was a sacrifice well worth it. Because of this, we stall on this rule because other modules can't output the fifo that we haven't deq from. It acts as a bottleneck.

II. Microarchitectural Design : Sobel Filter

The design specified in this section is very intuitive and simple to implement.
The module takes whole line of image and puts it into a register. Additionally it keeps last 2 received lines in registers too, since Sobel kernel is 3x3 matrix and needs all surrounding pixels to calculate the value.
After the receiving data, all register values sent to calculate Ix and Iy for one line. After, magnitude and direction is found for that line, register values are shifted (r1<=r2, r2<=r3) and new line is waiting to be received.
There some exceptions due to the requirement of having the values for previous and next lines in order to calculate magnitude and direction for current line. First exception is that conversion could be started only after at least two lines are received. Second exception is that data don't need to be received in order to convert last line.
Because of the exceptions, there is one cycle latency in processing.

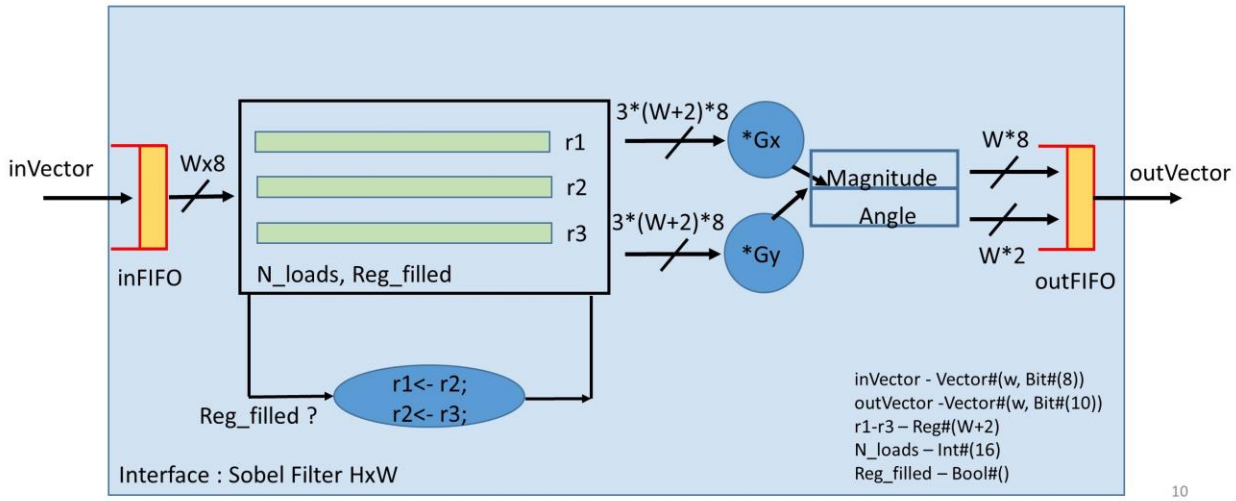


Figure 7. Simple Microarchitectural design of Sobel Filter Module.

The Canny Filter will assume Sobel applied it's effect on the image and that each pixel that was originally in DRAM is replaced with a Sobel'ed version and a bit indicating its orientation for non-maximal suppression. I assumed we were working with 128x75 images

Our Canny filter will work in the same way our Sobel does in terms of looking at every pixel. Begin by:
- We load the top row of pixels from DRAM onto a vector of registers V1 and use that same row for V2 and use the next row for V3. We also pad the beginning and end of each vector with the last register in each so each V is 130 pixels (where V[0] = V[1] and V[127] = V[128])

- Then we run a suppress function which takes a 3by3 block of pixels, looks at the middle one to see if it is a maximum in the direction orthogonal to its theta value and greater than the inputted threshold value. We can implement our theta values differently using a tan function, or simplifying it by just developing quadrants of interest. I assumed either vertical or horizontal.

- When the 3x128 pixel row is complete, we shift V2 to V1, V3 to V2, and we load the next row of pixels into V3. And go into another suppression stage.

- Let's say we do rows n, n+1, n+2 then where n is even. Then we just created the new line for n+1. We output this result to Vout1. We can't immediately copy this back into DRAM because we'll need the n+1 row value for the next suppression computation. Then we pull n+1, n+2, n+3. We can throw the results of n+2 into Vout2. Now when we get to n+2, n+3,n+4 we can update the n+1 values so we can shove Vout1 into DRAM and then update it with n+3 values. And so on…until the end and beginning…which still needs to be thought out.

- The module will take in a threshold value of 0-256 indicating the threshold of gradient values to keep.

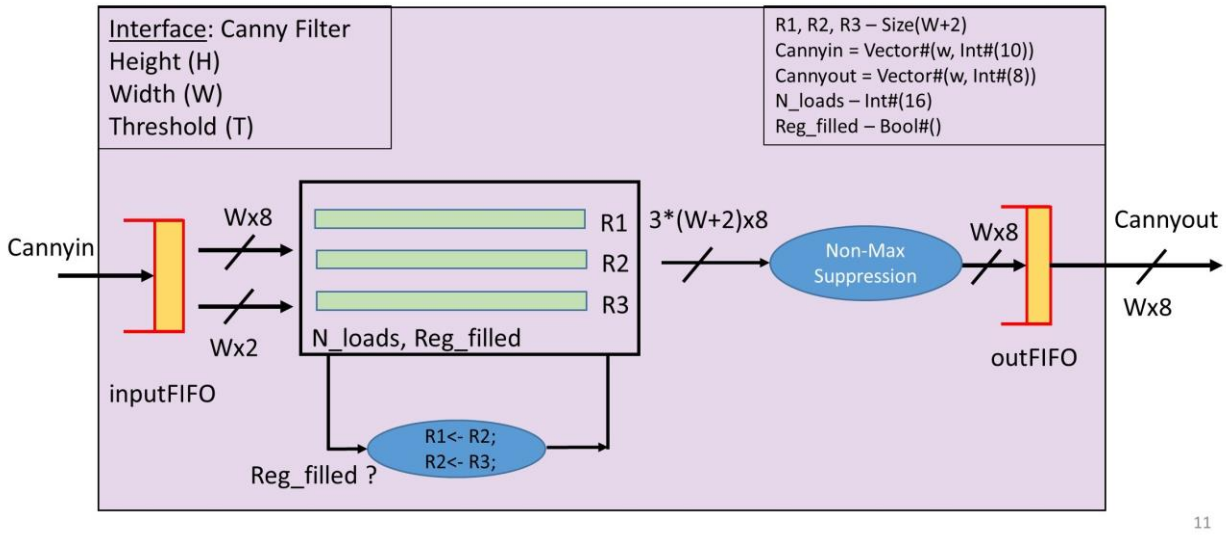


Figure 8. Simple Microarchitectural design for Canny filter module on FPGA

IV. Microarchitectural Design : Modifications

Since DRAM sends data in chunks of 64 Bytes it makes sense to implement Sobel and Canny modules to receive and output 64 Bytes vectors and process data while line is not fully received. This strategy is very nice for slow DRAMs, since it doesn't waste time while line is not completely received.
Another huge advantage of this method is that it reduces the amount of processing hardware. Instead of processing for whole line, now we need only 64 processing units.
Therefore, this design scales nicely with larger images, since only 3 registers are increasing size, while the rest of hardware kept the same.

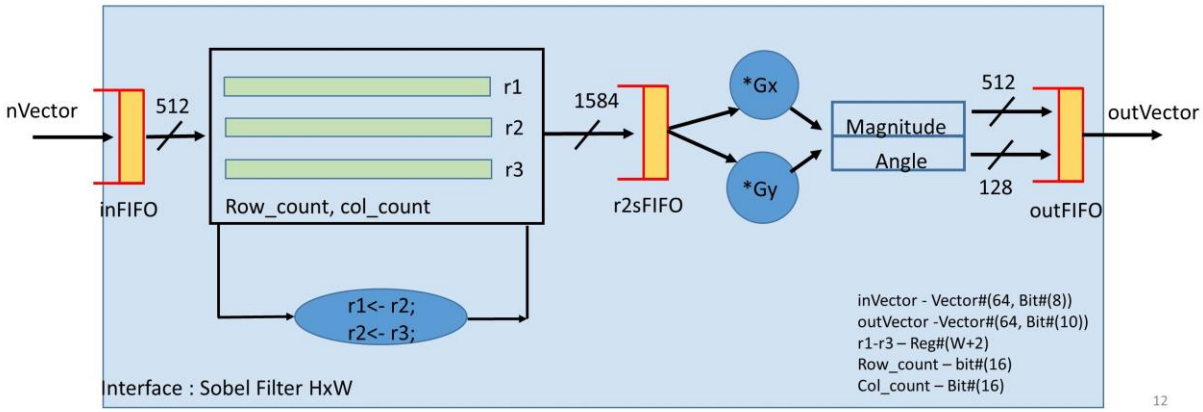
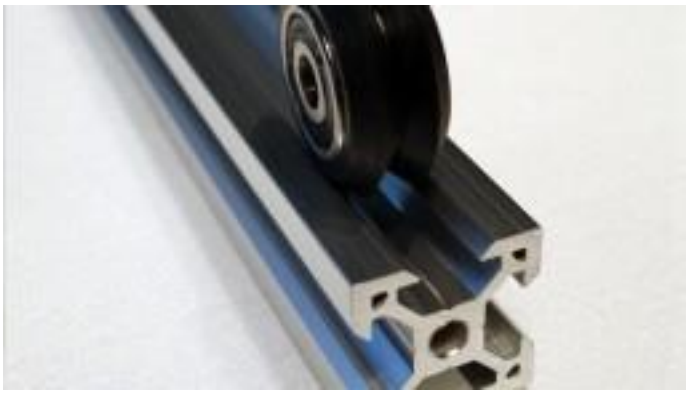


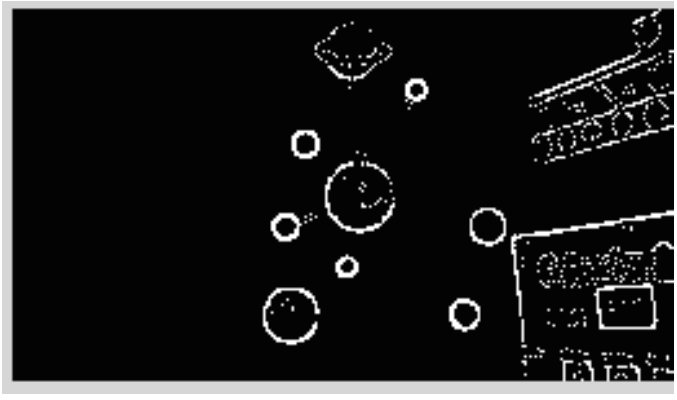
Figure9. Modified Microarchitectural design of Sobel Filter Module.

Though this design is very nice and has a lot of advantages in terms of performance. However it is getting much more complicated than the simple design discussed before. For example, if chunk #j is received at current cycle, then only now the previous chunk #j-1 could be converted, because there was lack of one pixel to make the processing of chunk #j-1 at last cycle. When the last chunk in line is received, the next cycle could be processed without receiving data from DRAM. Therefore, if assumed that data is always available, this implementation needs w/64+1 cycles to convert 1 line of image, where w is the width of an image. This number of cycles is larger than number for easy implementations, but the critical path is smaller for new implementation and frequency of the clock could be increased, since less processing is done in one cycle.

Figure 10. Frozen State of Modified Sobel Module. Helpful for cycle analysis.

Results





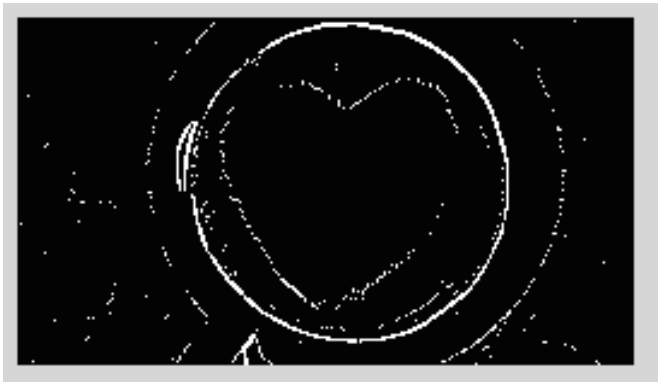


Figure 11. Outputs of Canny filter FPGA implementation.

Synthesis, timing and area

Implementation Results for 128x72:

Timing:
Worst Negative Slack: .158ns
Total Negative Slack: 0ns
Worst Hold Slack: .018ns
Total Hold Slack: 0ns
Total Number of Endpoints: 101419

Utilization:
Slice LUTs: 63766
Slice registers: 43862
Memory: 24
IO: 160
GT Channels: 8
Clocking: 14

Implementation Results for 256x144:

Timing:
Worst Negative Slack: .091ns
Total Negative Slack: 0ns
Worst Hold Slack: .023ns
Total Hold Slack: 0ns
Total Number of Endpoints: 152278

Utilization:

Slice LUTs: 102140
Slice registers: 67800
Memory: 24
IO: 160
GT Channels: 8
Clocking: 14

Design Exploration

This project was designed as if the FPGA received the data directly from camera at very high speed. However, if the data from computer or slow memory need to be converted, there are nice ways to continue exploration of the design:

- Implement image compression and decompression. This will allow take input as compressed image format and output it into image format. This is very convenient, because then it wouldn't require matlab doing extra conversion from binary or text file.
- Implement video compression. This is much harder than image compression, but it will achieve very high compression ratio.
- Output only coordinates of white pixels instead of whole image. This will reduce the amount of data sent back, because lines take not much space after canny filtering, therefore sending the coordinates of only white pixels will increase speed of transmission.

Some other possibilities to continue the design project are concerned expanding the capabilities of implemented modules:

- Implement Canny filter using both Lower and higher thresholds. This will allow to clear output image even further. But this design is more complicated, since it requires whole image, instead of only 3 last lines.
- Implement corner detection. This may be helpful for robots, since it finds peculiar edges instead of lines, which are easier to follow or to avoid.