**EE 454: Edge Detection using an FPGA**

Aidan Blant blant@usc.edu

Yinyi Chen yinyic@usc.edu

Emma Smith emmarsmi@usc.edu

**Abstract**

Our goal was to design a dedicated edge detecting hardware design that takes in an image and produces a binary edge location image that utilizes parallelization to speed up the edge detection process. We designed our architecture by applying a ‘what defines how’ approach based on our decided functionality.  We created a state diagram in Matlab Stateflow to simulate and verify our working Sobel operations and thresholding. We programmed the hardware-bound software in Modelsim to ensure correctness and to test variations on the final design for how to go about parallelization. Our final synthesis and programming of the Nexys FPGA was performed in Vivado and pulled together all the modules we created to run the edge detection end to end.

**Introduction and motivation**

Edge detection is an image processing technique for finding the boundaries of objects within images. It works by detecting points in the image where there is a sharp contrast in brightness and color, and strings the points together to form edge segments. Edge detection a fundamental step for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision. Our project focused on increasing the performance of using a Sobel based algorithm by using FPGA hardware parallelization.

One of the main challenges was optimizing the use of the Nexys board and its ability to run parallel processes. By decomposing the Sobel algorithm, we identified how to best divide an image and improve the speed at which our process runs as well as reducing the total number of operations.

Another challenge faced in edge detection is the ability for the algorithm to work on any image, regardless of the number of edges or the amount of contrast in the image. Thresholding decides if a particular point should be considered part of an edge and it greatly impacts the final output. We tried to have a dynamic way of assigning this level of threshold to tailor it to specific images with varying amounts of edges and textures. We also implemented a method called to go over the image and correct any “extra” or “missing” pixels and to improve the final output that is displayed.

With a few other teams also tackling this project, we wanted to to stand out by implementing non-trivial interesting features. Some of the considerations were the capacity to handle variable image sizes, automatic adjustment of threshold values, and fine grain parallelization.

**Previous work**

Edge detection is a well know problem with many applications, so there is much prior work on the topic with various aspects specifically targeted. From our research, we have found that FPGAs have been used for their large memory which provides a platform for processing of the Sobel algorithm with substantially higher performance than programmable digital signal processors[[1]](#footnote-1). The fact that our Nexys board has a fairly substantial amount of memory means that we can read in the entire image one byte per pixel and store in memory at once to work on with multiple processes.

In our research we found a paper that used soft-threshold wavelets to remove noise of the picture when applying the Sobel operator resulting in a better picture output[[2]](#footnote-2). As well as a paper that utilized pattern-recognition and probabilities in testing the enhancement of thresholding for various images[[3]](#footnote-3). Finally, there was a paper on edge thinning, which describes an algorithm that can remove spurious or unwanted edge points and add in edge points where they should be reported but have not been in an efficient manner to clean up the final image[[4]](#footnote-4). The focus of our project design is paralyzing the Sobel operator, and we decided that with the time constraints we would not be able to implement the wavelet or probability pattern-recognition methods due to a high complexity. Thus, we worked toward the simpler approach of edge thinning which only needed a few “rules” to be put in place as we passed over every pixel.

**Approach and concrete results**

Our application characterization graph shows these five steps, which we will go into further detail later:

1. Loading and parsing of grayscale image from a file into a one-dimensional height\*width\*8-bit array

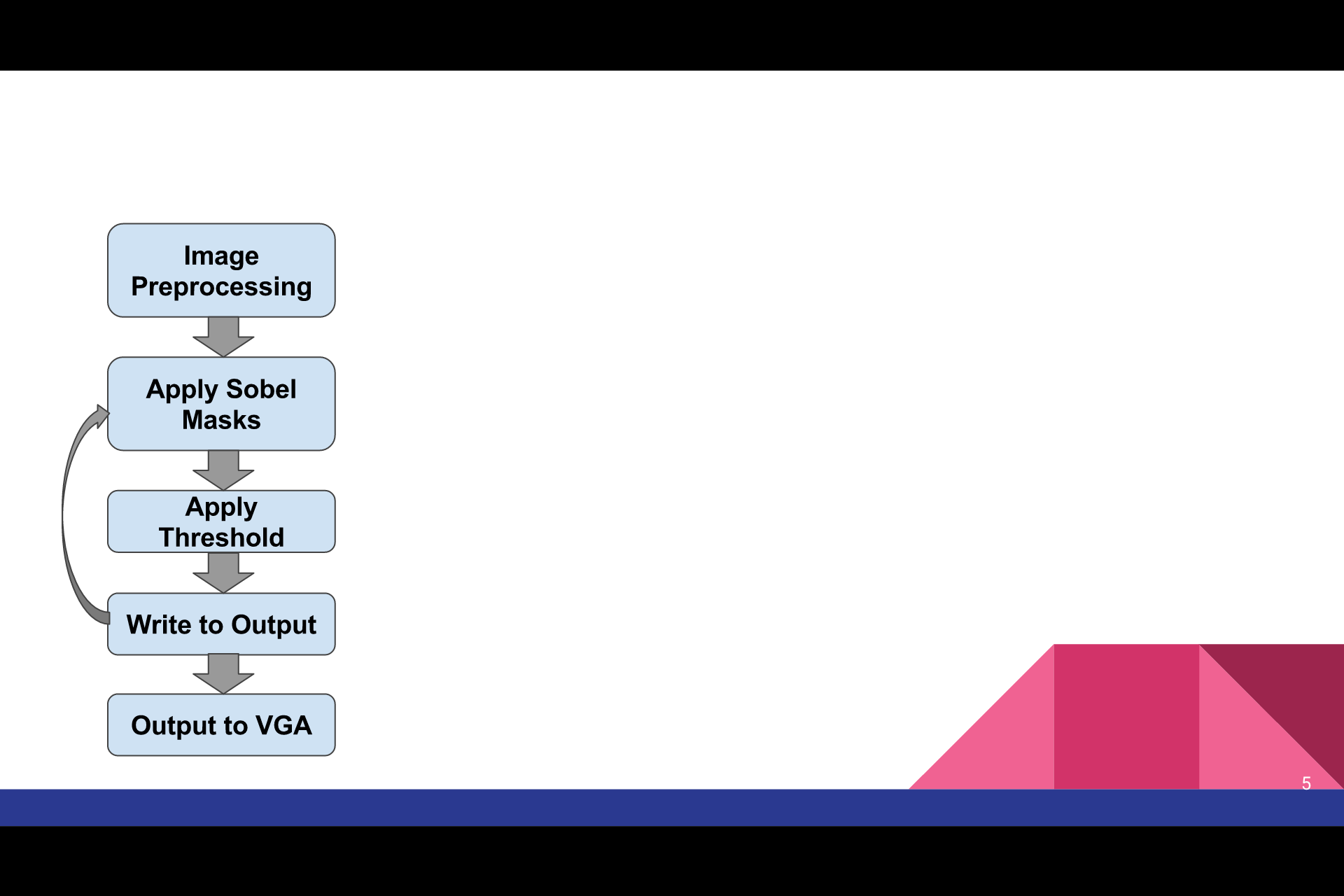
[Parallel step: Division of image into ]

2. Performing Sobel convolution of the 8-bit values in predetermined 3x3 matrices(Gx and Gy) with the 3x3 matrix centered for each pixel in the image outputting an 8-bit value(parallel)

3. Applying the thresholding to each pixel taking in the 8-bit value and outputting a 1-bit value (parallel)

4. Writing the 1-bit output back to a two-dimensional height\*width array (parallel)

5. Displaying output edge image on monitor via VGA, using red, green, blue, h\_sync, and v\_sync values.



Application Characterization Graph

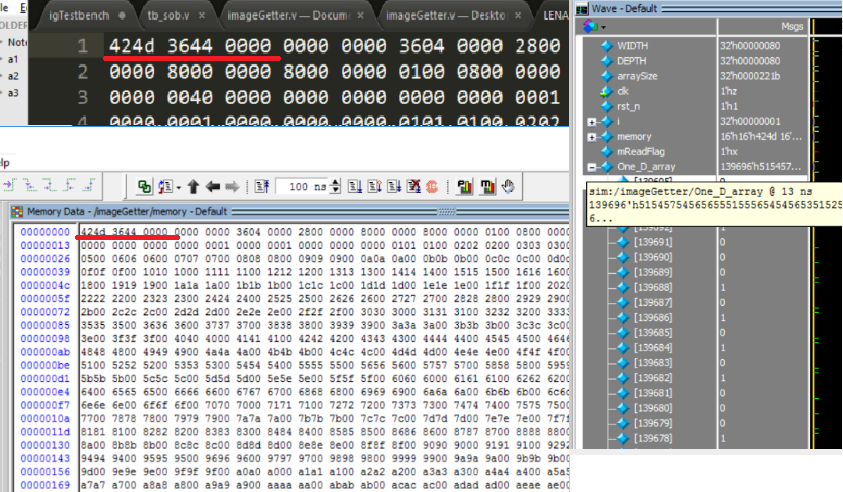
*1) Image parsing and Loading*

Figure 1

Figure 1 demonstrates the functioning image memory reading and parsing into a one-dimensional array. The “imageGetter.v” module performs a readmemh on target file into a 16-bit wide array of length determined by the image size which is stored as a parameter. The parameter ‘arraySize’ is specifically ( 54 +WIDTH \* ( DEPTH + 8 ) ) / 2 where image width and depth are in pixels. The 54 is the header for bitmap images, and the + 8 is padding that bitmap adds to every line as a correctness assurance. These values can be adjusted for other file types, although detection of file type and auto-sizing of header and padding were not in the scope of this project.

imageGetter.v converts this 16-bit wide array into a one-dimensional array with a simple for loop from 0->arraySize that reads each 16 bit segment into the next available 16 bits of the one-dimensional array.

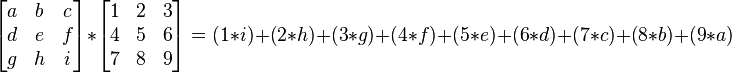
While the final version of imageGetter.v was 40 lines long, it was not a trivial challenge. The bitmap file

Figure 2

must be formatted correctly, and a simple problem as not having an end line in the sample image took well over an hour to

discover and debug. It was in this module we discovered the “+:” operator present in line:

**One\_D\_array[ 16\*j +: 16 ] = memory[j][15:0];**

Whereas earlier implementations were plagued with ‘Range must be bounded by constant expression” errors.

*2) Perform Sobel convolution*

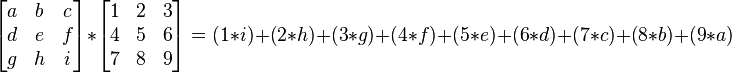
The operator uses two 3×3 kernels which are [convolved](https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution) with the original image to calculate approximations of the[derivatives](https://en.wikipedia.org/wiki/Image_Derivatives) - one for horizontal changes, and one for vertical. If we define A as the source image, and G*x* and G*y* are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations are seen in figure 2.


\mathbf{G}_x = \begin{bmatrix} 
 -1 & 0 & +1  \\
-2 & 0 & +2 \\
-1 & 0 & +1 
\end{bmatrix} * \mathbf{A}
\quad
\mbox{and}
\quad   
\mathbf{G}_y = \begin{bmatrix} 
-1 & -2 & -1 \\
 0 & 0 & 0 \\
+1 & +2 & +1
\end{bmatrix} * \mathbf{A}




\mathbf{G}_x = \begin{bmatrix} 
 -1 & 0 & +1  \\
-2 & 0 & +2 \\
-1 & 0 & +1 
\end{bmatrix} * \mathbf{A}
\quad
\mbox{and}
\quad   
\mathbf{G}_y = \begin{bmatrix} 
-1 & -2 & -1 \\
 0 & 0 & 0 \\
+1 & +2 & +1
\end{bmatrix} * \mathbf{A}





The x-coordinate is defined here as increasing in the "right"-direction, and the y-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude. We can also calculate the gradient's direction.

The Sobel operator represents a rather inaccurate approximation of the image gradient, but is still of sufficient quality to be of practical use in many applications. More precisely, it uses intensity values only in a 3×3 region around each image point to approximate the corresponding image gradient, and it uses only integer values for the coefficients which weight the image intensities to produce the gradient approximation.

*3) Thresholding*

The first thresholding we tried was just a set number that was checked against to determine if a value was actually an edge This requires that the user define the threshold value for each image. We wanted to create a method such that our program would automatically change the value for each image and each subset of the image based on the amount of edges, the contrast, and the detail of the image. When we researched more in depth how others have done this and implement is, we realized that it was far more difficult for just an addition to our main focus.

Instead we utilized an edge thinning technique that was used for adjusting the results that come from a threshold value that is not perfectly tuned to the image and pixels. This involves sweeping through each pixel and applying rules. If a pixel has zero neighboring pixels that are edges, then the edge point is removed. If it has onw neighbor, the neighbor with max edge value is used to continue the edge and full in the gaps. If the pixel has twp neighbors, there are three possible cases: If the point is ``sticking out'' of an otherwise straight line, then compare its edge response to that of the corresponding point within the line. If the potential point within the straight edge has an edge response greater than 0.7 of the current point's response, move the current point into line with the edge. And if the point is adjoining a diagonal edge then remove it. Otherwise, the point is a valid edge point. Next if the pixel has more than two neighbors, and is not a link between multiple edges then thin the edge.

This is the basic description of how our thresholding works. It is also parallelized similarly to our Sobel operator, where each row’s pixels are all their own processes.

*4) Writing to memory*

This is a fairly basic segment of our code. We need to change the method of indexing from one dimensional to a two dimensional array so that it can be correctly displayed. In our testing we also noted that this memory writing had the same issue of latency with how the multiple processes are clashing with each other.

*5) Displaying via VGA*

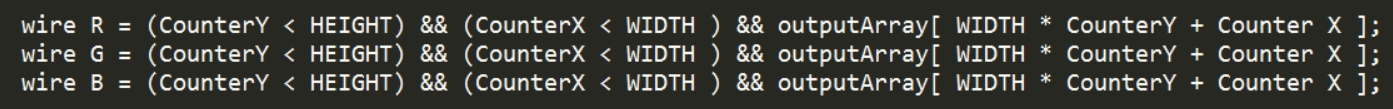
VGA display module “snake.v” was adapted from a ee201 final project adjusted to display the edge image. It successfully displays black/white images via vga cable from a one-dimensional binary array wherein ‘1’ represents an edge to be displayed in white, and ‘0’ is displayed in black. This was the simplest module to implement as it was adapted from earlier functioning projects that we had written in other classes in our careers at USC.

Figure 3

One major architecture design decision was between coarse-grained parallelism and fine-grained parallelism. One method of coarse-grained parallelism for the edge detection operation is the division of the input image into a number of chunks equal to the number of processors, each processor then performs the convolution operation row by row and column by column, and the end image is stitched together. A fine-grained approach would have been to create an array of sets of 3x3 matrices, ie. an array of size 9\*WIDTH\*DEPTH\*(bitWidth), wherein operations on each 3x3 matrix can be performed independently, allowing each processor to claim an unfinished matrix and perform the convolution operation. The fine-grained approach takes nine times the memory of the coarse-grained approach, but could be used to perform “quick approximations” edge detection that achieves higher and higher resolution as time passes], or to calculate edges in a specific order, such as radiating from the center of the image, or top to bottom.

We tested various methods of parallelization, starting from each pixel being processed in sequence, each row being processed in sequence with the pixels inside it in parallel, and each pixel having its own process.

We found that the best performance came when we only parallelized one row at a time. This may be because the Sobel operator needs the data of the eight surrounding pixels and the memory accesses of each process overlap causing latency. From a few of our trials we saw that the level of latency that arose when each pixel had its own process was higher than the benefit of doing the Sobel convolutions at the same time. The math that is used is simple enough that it does not take more than a few clocks to perform when there is no contention for memory.

With more tuning to efficiency of parallelization we could have come up with a more complex method of splitting up the image to for Sobel operations instead of just by row. This could have be done such that there were much fewer pixels that were being calculated at the same time with neighboring pixels that would cause clashing memory reads. However this was much more than we could accomplish in the scope of our project.

**Conclusions**

This project reinforced the importance of the iterative design process and paying special attention to the transitions between states.  Coming from a Computer Science background, the limitations of passing only one-dimensional arrays between modules took some maneuvering, as well as the limitations of for loops within always and initial blocks. It was very interesting to see how the tradeoffs of parallelization are so evident in performance. We would have imagined that more parallel processes would be better, but saw that it was very important to balance that with what they access and how they interact.

As certainly the largest-scale Verilog project for any of us, this provided great instruction in our knowledge of EE architecture design, simulation, and implementation.  Coding beyond the scope of minor lab assignments forced each team member to search for solutions that required the use of new operators (such as the “+:” operator to represent an array width).  This project also required the revision, or entire redesign, of sections of code as we learned the distinct capacities and limitations of Matlab Stateflow, Verilog, and FPGA design.

1. A novel FPGA-based architecture for Sobel edge detection operator

   T. A. Abbasi; M. U. Abbasi

   International Journal of Electronics [↑](#footnote-ref-1)
2. An imroved Sobel edge detection

   Wenshuo Gao; Xiaoguang Zhang; Lei Yang; Huizhong Liu

   Computer Science and Information Technology [↑](#footnote-ref-2)
3. Quantitative design and evaluation of enhancement/thresholding edge detectors

   Abdou, I.E.; Pratt, W

   Proceedings of the IEEE [↑](#footnote-ref-3)
4. Edge Thinning Used in the SUSAN Edge Detector

   S.M. Smith

   Oxford Centre for FMRIB [↑](#footnote-ref-4)