ECE 5775: Final Project Report
FPGA-Based Acceleration of Canny Edge Detection

Aadeetya Shreedhar and Alexander Wang

1. Introduction

This project implements an algorithm to perform real-time Canny Edge Detection on a high definition video stream. Our algorithm is executed on a Xilinx ZYNQ evaluation board using C++ and Vivado HLS. Edge detection extracts discontinuities, or edges from an image by identifying sharp changes in image intensity. The edge detection algorithm filters an input color image and outputs a black-and-white image with the edges in white. Canny edge detection is an "optimal" edge detection algorithm, which means that it has 1) good detection - it detects as many real edges as possible, 2) good localization - edges are found as close as possible to the real edge in the image, 3) minimal response - edges are only detected once and the algorithm should not respond to image noise. Canny edge detection seeks to improve the results over other edge detection algorithms, such as the Sobel filter.

In this project, we demonstrate a marked improvement in execution time of the real-time Canny edge detection through a hardware implementation of the algorithm over an OpenCV-based [7] software implementation. We modified an existing image processing project from Xilinx [2] to perform Canny edge detection through a dataflow which utilizes OpenCV-like HLS constructs and C++ functions. Using the Vivado HLS flow, we were able to convert our C++ code into synthesizable RTL which we mapped onto the Artix 7 based FPGA on the ZYNQ board. Using a standard HDMI cable, we can input real-time video into the ZYNQ and show real-time Canny Edge Detection of our output on a screen using either the software implementation or the hardware implementation. Our hardware edge detector achieves a clock frequency of 83.74 MHz to yield an effective frame rate of 40.38 frames per second, where each frame is 1920x1080 pixels.

2. Algorithm Overview

The Canny Edge Detection algorithm is structured in a way that lends itself to pixel-by-pixel streaming data flow of the image. The algorithm consists of 5 main stages, which are mapped to a 5-stage pipeline in which data flows linearly from one stage to the next. Almost all computations on a pixel can be completed on that pixel individually, or within a small neighborhood of pixels around the pixel in question. Line buffering and memory windows are used in our implementation to store the pixels around the current pixel being processed such that it is not necessary to ever obtain a pixel more than once in a certain stage of the algorithm. This way we can stream the image in 1 pixel at a time, and never have to store the entire frame of the image. This dataflow organization allows us to achieve a throughput of one pixel per clock cycle, reduces the need for large memory structures, and facilitates real-time video processing.

This section describes the 5 stages of the algorithm:

1. **Grayscale Conversion** - Convert the image to grayscale in order to represent the intensity of each pixel with a single number
2. **Noise Filtering** - Apply noise filtering through convolving the raw image with a 5x5 Gaussian Blurring Filter kernel, with a sigma value of 1.4, such as the one below in Figure 1. This reduces noise in the image through blurring such that the algorithm doesn't react to individual noisy pixels.

$$\mathbf{B} = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} * \mathbf{A}.$$

Figure 1: Gaussian Blur Kernel [8]

3. **Find the Intensity Gradient** - The image is convolved with two 3x3 Gradient kernels in the x and y direction, which is identical to performing Sobel edge detection in both the x and y direction (using a kernel as shown below in Figure 2).

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

Figure 2: Sobel Operator Kernels [11]

   We use an existing Sobel function provided in the Vivado HLS Video API to perform this, which slides the kernel across the image and convolves it with the pixels in the image. The result of this is the gradient (first derivative) of the image in both the x and y directions, which provides rudimentary edge detection. We then combine the absolute values of the directional gradients to calculate the gradient's magnitude, and find the gradient's direction through the arctan2 of the two directional gradients (this indicates which direction the edge is against). We do not compute the square root required for gradient magnitude computation to simplify hardware and optimize execution time. Since we cannot perform floating point arithmetic in hardware without excessive resource utilization, we generalize this arctan2 function using some conditional statements and compute the direction to be one of 0°, 45°, 90°, or 135°. Each pixel of the image will have its own gradient magnitude and direction value after this step. These first order approximations for gradient magnitude and direction computation trades off quality of result for execution time.
4. **Non-Maximum Suppression** - Also called edge thinning, this technique attempts to reduce edges to be one pixel wide. We use 3x3 kernel to check whether the gradient magnitude of the two pixels around the current pixel in the direction of the gradient is larger than the gradient magnitude of the current pixel itself; if so, we remove the gradient of the current pixel, otherwise, we leave the gradient. Removing the gradient of a pixel removes it from being a part of the edge, as it is not a local maximum of the gradient and thus cannot be part of the maximal

edge.

5. **Hysteresis Thresholding** - We use hysteresis thresholding to perform edge tracing, which highlights and intensifies important edges in the image. Given two threshold values, if a pixel is below the lower threshold, it is rejected as an edge. If it above the higher threshold, it is accepted as an edge. If it is in between the two thresholds, it is only accepted as an edge if it is connected to another pixel that is an edge. This is also done through a 3x3 kernel, as we need to check all 8 edges next to the current pixel to see if they are an edge for cases when the current pixel is between the two thresholds. We chose the values of 30 and 90 for our threshold values (given an 8-bit black and white pixel that ranges from 0 to 255), as these heuristically provided the best results.

3. Implementation

Our implementation is based off of a Xilinx Application Note [2] which describes a C++ implementation of an algorithm which performs a series of video processing functions on a color image in a data flow similar to what we desire for the Canny Edge Detector. It allows for real-time streaming of the pixels, such that we never need to store the entire video frame/image in memory. This Application Note performs the algorithm both in software with OpenCV library functions to create a "golden" image, as well as in hardware using OpenCV-like HLS video API functions which are synthesizable into RTL using Vivado HLS to create the result image. Each of the major stages in the Canny Edge Detection algorithm described in the previous section acts as a functional block, such that the pixels of the image can be pipelined through each functional block.

On the OpenCV-based software side, a full implementation of the Canny Edge Detection algorithm in addition to the Grayscale and Gaussian Blur functions is already available as a function in their API, such that we can chain these three functions together and push the image through to obtain the "golden" software implementation output image.

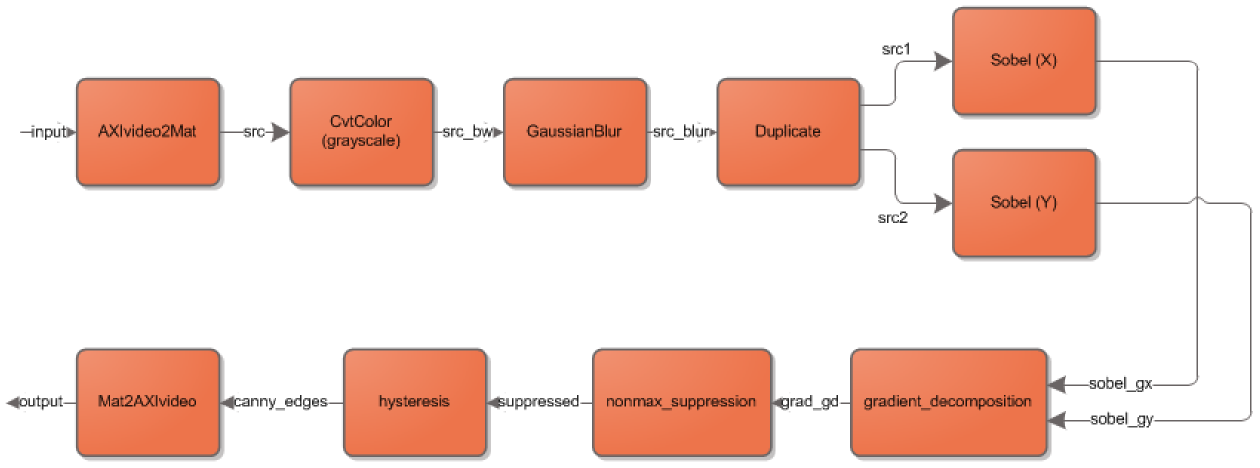

Figure 3: Block Diagram of the Canny Edge Detection Dataflow (data and functional blocks are labeled with their names in the code)

On the hardware side, we needed to write C++ code that is synthesizable into RTL through HLS tools in order to closely match the software implementation. As stated before, Xilinx provides a HLS Video API which provides a small subset of functions similar to those found in the OpenCV library, including the Grayscale function, the Gaussian Blur function, and the Sobel function. Our dataflow passes the “RGB_IMAGE” datatype between each functional block. RGB_IMAGE is a matrix that represents the entire image/video frame. Figure 3 shows the dataflow in our implementation of the Canny algorithm. The data is not actually passed as a matrix, but rather a stream of 3 channel, 8-bit pixels, going from the top left to the bottom right of the frame in an x-direction priority order. We first push the raw image pixel stream from the raster into the Grayscale block, then the Gaussian Blur block, and then duplicate the pixel stream to be inputted into the two Sobel blocks (one for x direction, one for y direction). Taking the two outputs of the Sobel blocks that represent the directional gradients Gx and Gy, we input them into the “gradient_decomposition” block and calculate the magnitude and direction of the gradient pixel by pixel (through a double nested-for loop), concatenating the two pieces of data into 16 bit “pixels” to form a 16-bit version of an RGB_IMAGE (this is so we don’t need to buffer two separate pixel streams in the next step).

Next we take this 16-bit RGB image and pass it into the “nonmax_suppression” block, which will perform Non-Maximal suppression as described in the previous section using a 3x3 kernel. The 3x3 kernel is implemented using line buffers and memory windows, such that input pixels which are needed for computation of future output pixels are saved until they are no longer needed so that input pixels never need to be read more than once (resulting in an Initiation Interval of 1 and ensures a linear streaming flow of the data). Line buffers are implemented in hardware using dual-ported BRAMs (with 2 read/write ports per row of the line buffer), and memory windows are implemented using shift registers.

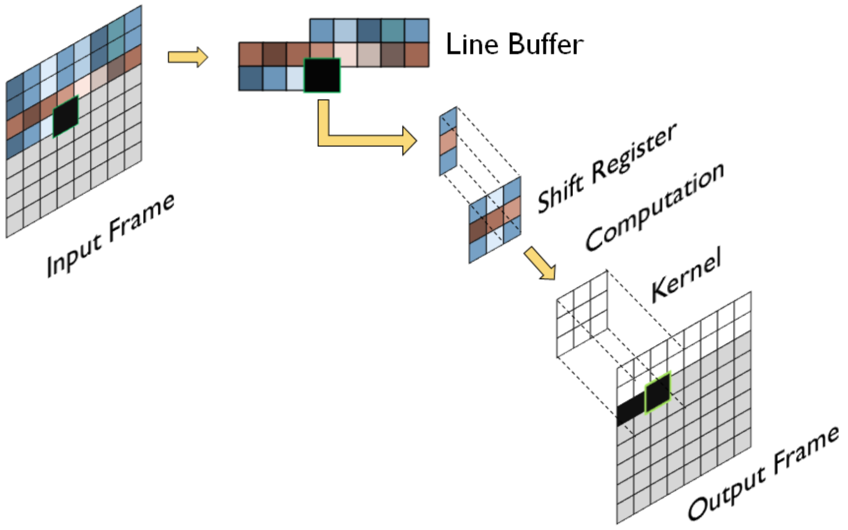

Figure 4: A diagram of how the line buffer and shift register memory window operate

The 2-row line buffer stores data to be pushed into the 3x3 sliding window which represents the kernel that stores the data necessary to compute the current output pixel. On each iteration of the loop, we first save the two values in the line buffer at the current column into temporary variables, and then shift the two values up 1 row, discarding the top value. We then read 1 new pixel, and store it in the bottom right of the line buffer (where there is now a gap). Finally, we shift the memory window right by 1 pixel and take the two temporary variables plus the new pixel and push them into the right column. By using line buffers to store pixels that will be necessary for the window, we can obtain all of the necessary 9 elements of data in the 3x3 kernel for the computation in exactly 1 iteration of the loop, resulting in an Initiation Interval of 1 and allowing us to perform a computation in real-time as the pixels are streamed into the block. A visual diagram of this process is shown in Figure 4. Once the new kernel's data is completed, we can calculate the output value according to the non-maximal suppression function for the pixel at the center of the kernel.

After non-maximal suppression is completed, we pass the data onto the "hysteresis" block where we use a similar line buffer and sliding memory window implementation to perform hysteresis thresholding as described in the previous section. After this point, the final output image is created for the hardware implementation. Much of the code in the second half of this dataflow is inspired by the work in [9].

In order for the C++ to be properly synthesized into hardware, we use HLS directives to tell the HLS tool to pipeline the inner loop of each of our functional blocks (the loop which iterates over the columns of the image). Using this "HLS PIPELINE" pragma ensures that the loops are dataflow pipelined such that one pixel can be processed every cycle, and it moves through a pipeline between the functional blocks. We do not loop unroll because the loop bounds get fixed at synthesis time, but we wish to allow for dynamically changing video frame sizes. We also find that loop unrolling would result in unacceptably high HW resource usage since there is a loop iteration for every pixel of the image.

The ARM core on the ZYNQ board is capable of running a Linux OS distribution, and can run the OpenCV software version of the algorithm whereas the hardware version of the algorithm will be implemented on the FPGA on the ZYNQ board. Simple commands can be executed over a serial connection to the board in order to choose whether to perform the algorithm through the software or the hardware implementation. The board displays the output of the algorithm on a screen. The input video is simply inputted through a HDMI cable, and the output video is also outputted to a computer screen through a separate HDMI cable. Note that our implementation only supports video frames of up to 1920 pixels wide, since that is the size of our line buffers (the HLS Video API doesn't support parameterizing the size of the line buffers). The user can access the board's Linux OS through a terminal using the TeraTerm program on a PC which is connected to the board through USB UART.

Results

Shown below in Figure 5 (a), (b) and (c) is an example image as it is pushed through our algorithm's data flow. We show the image after each intermediate stage of the process until we reach the final output image.

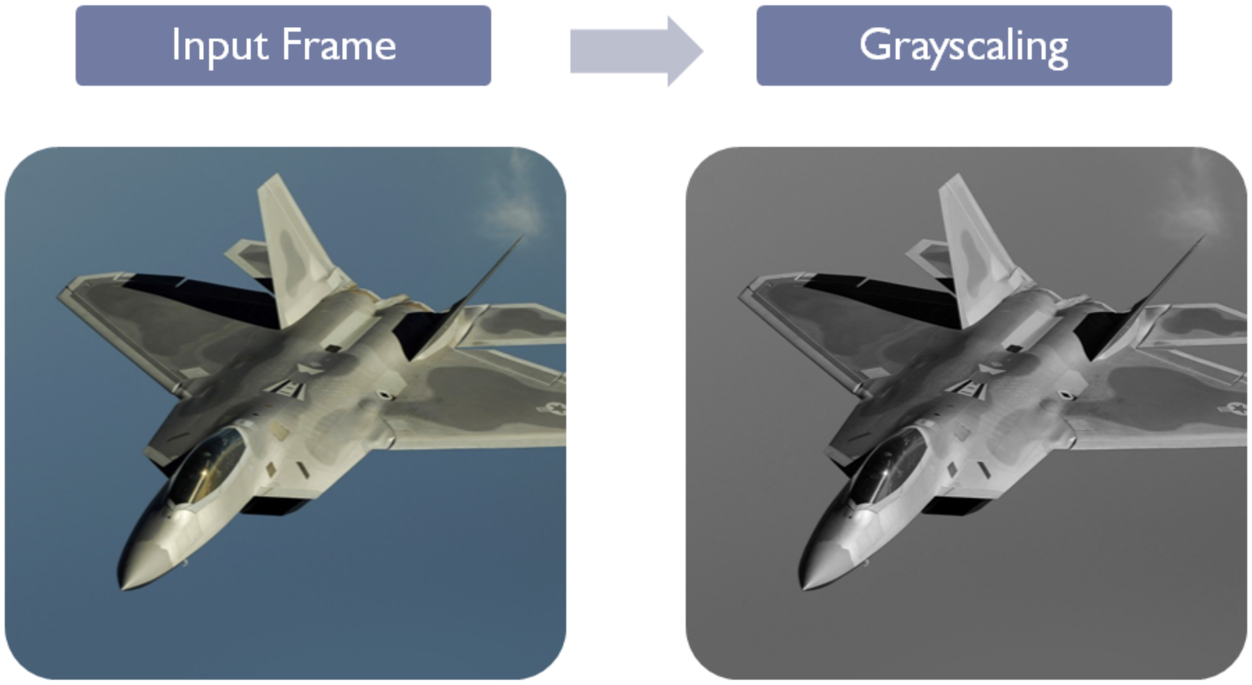

Figure 5 (a) Canny Filter: Input image and grayscaled image

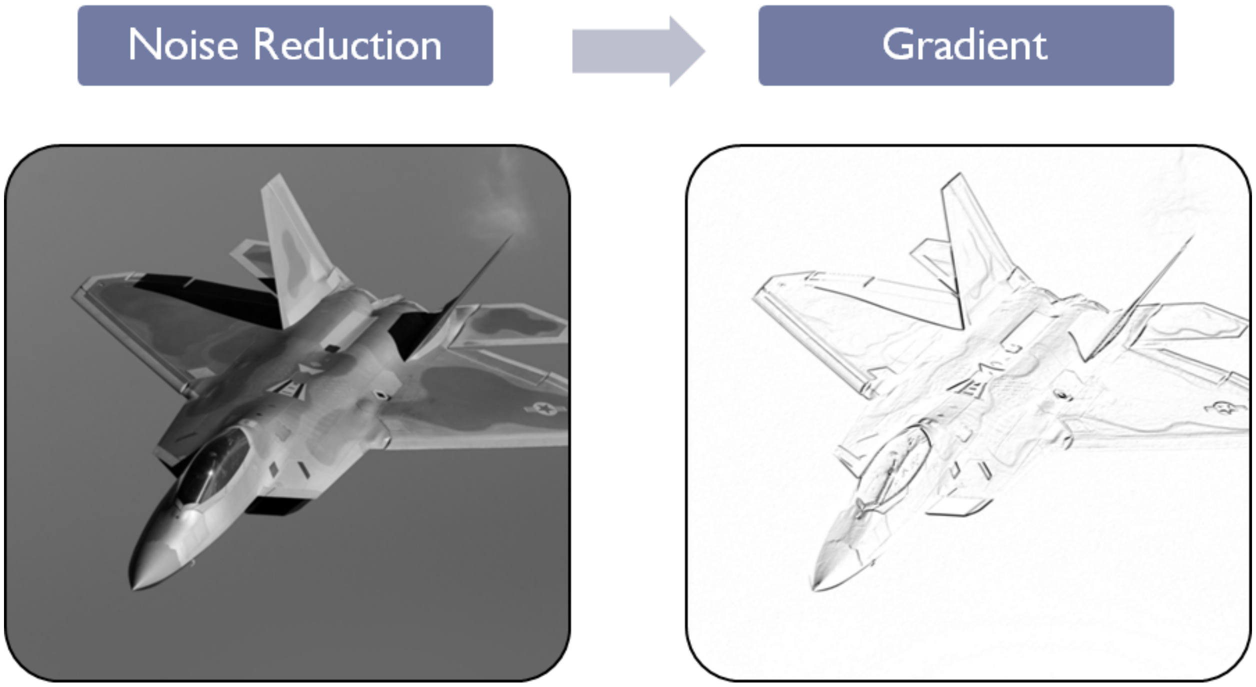

Figure 5 (b): Canny Filter: Gaussian blurring for noise reduction and Sobel filtering

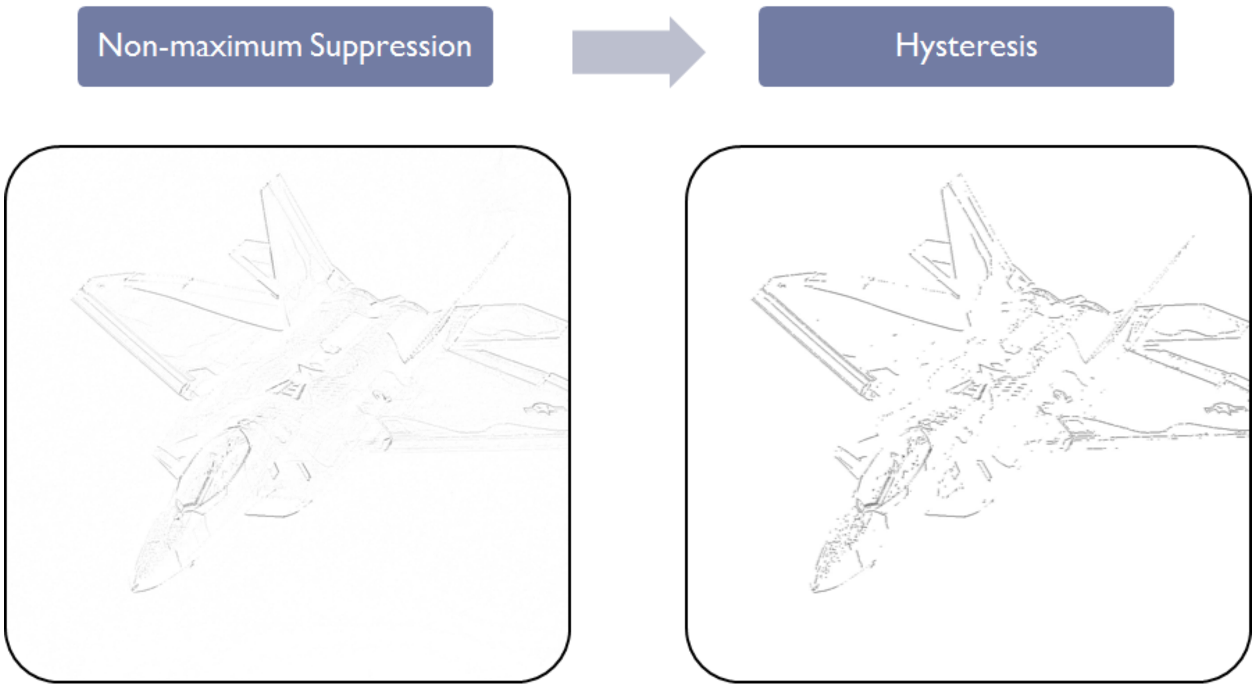

Figure 5 (c): Canny Filter: Non-maximum suppression and hysteresis (image inverted for visibility)

We found that our HLS hardware implementation of the algorithm produces slightly worse edge detection in quality than the software OpenCV implementation. This is largely due to the fact that we simplified the gradient magnitude and direction calculation and we use 8-bit integers to perform our calculations instead of more accurate data types. We used these simplifications to due to time limitations, but given more time we could create a hardware implementation that fully matched the software implementation in quality. Additionally, using these simplifications likely reduces the amount of resource utilization to synthesize our design on the FPGA. Meanwhile, we can see that the both implementations of the Canny edge detector perform better than the Sobel edge detector, resulting in much sharper edges and removing non-edge pixels. This verifies why Canny edge detection is a superior method of edge detection and has more value in real-time applications who need higher accuracy edge detection.

On the other hand, we can immediately see after performing the algorithm for both implementations using the ZYNQ board that using the hardware implementation brings a marked improvement in real-time image processing. The hardware implementation results in nearly lagless processing of a live video stream whereas the software implementation lags significantly (estimated about 3-4 frames per second). After obtaining the final clock rate of the hardware design through the post place-and-route timing reports of the synthesis, we found that our design runs at a maximum frequency of 83.738 MHz. Since our design is designed to process 1 pixel per clock cycle as it is pipelined with an Initiation Interval of 1, we can assume that 1 pixel is processed every clock cycle. Given a 1080p input video (1920x1080 resolution), we can process that stream at an estimated 40.38 frames per second, which in most cases is real-time to the human eye.

Table 1 below shows our total resource utilization after synthesizing the hardware for our design. The Canny edge detection accelerator occupies a relatively large amount of resources on the FPGA, especially in lookup-tables. However there are enough unused resources to accommodate other accelerators or to improve the accuracy of the Canny Edge detector. In particular, 72% of DSPs are still available, and could be used to compute square roots and arctan functions instead of the first order approximations used in our design. This would be an interesting area for future work.

Resource	Utilization Count	Utilization %
BRAM_18K	25	8%
DSP48E	62	28%
FF	27391	25%
LUT	23948	45%

Table 1: Table of our design's total resource utilization

Conclusion

This project gave us good experience in creating synthesizable C++ code that can be transformed into RTL with the Vivado HLS tool. We learned valuable lessons on how to create a streaming video processing architecture, such as performing dataflow pipelining and using line buffers and memory windows to store data for video processing. We also gained insight into image processing and edge detection algorithms and the C++ based OpenCV library. Finally, we learned how much easier it is to design hardware accelerators and coprocessors using C++ and HLS tools rather than low level RTL. Overall, this project allowed us to experience first hand how to implement an algorithm in hardware through the use of high level synthesis.

References

[1] Xilinx Application Note: Zynq All Programmable SoC Sobel Filter Implementation Using the Vivado HLS Tool
http://www.xilinx.com/support/documentation/application_notes/xapp890-zynq-sobel-vivado-hls.pdf

[2] Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries
http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf

[3] Implementing Memory Structures for Video Processing in the Vivado HLS Tool
http://www.xilinx.com/support/documentation/application_notes/xapp793-memory-structures-video-vivado-hls.pdf

[4] Introduction to FPGA Design with Vivado High-Level Synthesis
http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf

[5] Xilinx Product Selection Guide:
http://www.xilinx.com/publications/matrix/Product_Selection_Guide.pdf

[6] Xilinx ZYNQ SoC Board User Guide:
http://www.xilinx.com/support/documentation/boards_and_kits/zc702_zvik/ug850-zc702-eval-bd.pdf

[7] OpenCV Canny Edge Detector:
http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html#

[8] Wikipedia Canny Edge Detector:
http://en.wikipedia.org/wiki/Canny_edge_detector

[9] C. Desmouliers, E. Oruklu, S. Aslan, J. Saniie, and F. V. Martinez, "Image and Video Processing Platform for FPGAs Using High-Level Synthesis", *IET Computers and Digital Techniques, vol. 6, no. 6, pp. 414-425, 2012.*
http://www.ece.iit.edu/~eoruklu/IIT/Publications_files/IET%20HLS%20Video%20processing.pdf

[10] Vivado Design Suite User Guide: High-Level Synthesis:
http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_2/ug902-vivado-high-level-synthesis.pdf

[11] Wikipedia Sobel operator:
http://en.wikipedia.org/wiki/Sobel_operator