EECS 392 Final Report

Autonomous Vehicle Edge Detection

TEAM 3: YUE WANG, JAE WOO OK, LEANNA HUE
INSTRUCTOR: DR. ZARETSKY

Table of Contents

List of Figures

Executive Summary

Problem

Our group has tried to tackle the problem of trying to simulate an autonomous vehicle on an FPGA. Specifically, the aspect of the vehicle that we tried to address was the vision of the autonomous vehicle.

Purpose and Scope

The purpose of the project was to take a video of the road from the perspective of the car as the car is driving, and be able to compute and visualize the edges of the road which the car is driving on. Important hardware considerations included the area, memory, throughput, and computation speed of the FPGA. Important car interface considerations includes the safety features such as how the car slows down when it sees an object on the road.

Methodology

Extensive thought and revision was put into the construction of the design. The algorithm that we decided on was the Canny Edge Algorithm and the Hough Transform. The way that we tested our methodology was by testing each individual component with a testbench, and then by testing the overall design. Our group was able to compare the result of each individual component by comparing the output of each of our components with outputs from an implementation of the Canny Edge Algorithm and Hough Transform in C code. The output of each stage was compared to the output of each stage of the C code implementation pixel by pixel.

Design and Benefits

Specialized hardware implementations can be much faster than software implementations of the same logic in a serialized fashion. In addition, using an FPGA to implement these algorithms allows for more flexibility on what the FPGA are able to do later. The limitations of our design would mostly come from the amount of memory and speed of the board.

Design Overview

Introduction:

In this project, our team will work to create an Autonomous Vehicle Edge Detector using an FPGA board. Our design will initially plan to first encompass breaking the video into frames and send them onto the board. Then, we will implement the Canny Edge Algorithm to find the edges of each frame and use the Hough Transform to determine the most significant edges. This will allow the detection of the two edges of the road, and the program will project these lane lines back onto the original frame. Finally, we will display the image on the screen. There have been prior implementations of this approach, but nearly all of them have been software approaches. Our group attempted to implement this in hardware.

Broader Considerations:

Autonomous driving is becoming more and more common these days, and to handle the huge amount of calculation, the usage of FPGA boards is a good choice. In our project, we will simulate the process of how a self-driving car is able to detect the edges of a lane. There are some existing designs implementing edge detection on an FPGA, and we will attempt to finish the whole process from edge detection to line extraction on an FPGA board. If we succeed, we can prove the possibility of using FPGA technology in the autonomous driving industry and provide solutions for some possible software constraints that exist today.

Design Constraints and Requirements:

Resources: We need huge number of computing units and registers to let us go through the whole process. Thus, major concerning constraints would be the amount of area and memory needed on the ALTERA DE-10 NANO board for our design to be synthesizable.
Speed: We needed our design to be fast enough because we would have to process many frames per second, if the design is too slow, the car won't be able to make real-time decisions, and thus our design would not be able to be used on autonomous vehicles.
Accuracy: The accuracy should be high enough to ensure the cars can make right decision by clearly identifying the sides of the road.

Design Description:

Overview: The basic overall process of our design was to split the video of a car driving (from the car's perspective) into frames, and transfer the data of the images from the computer's Ethernet port to the board, and let the images go through the Canny Edge and Hough Transform frame by frame. Finally, the output image should be projected on a monitor through HDMI port.

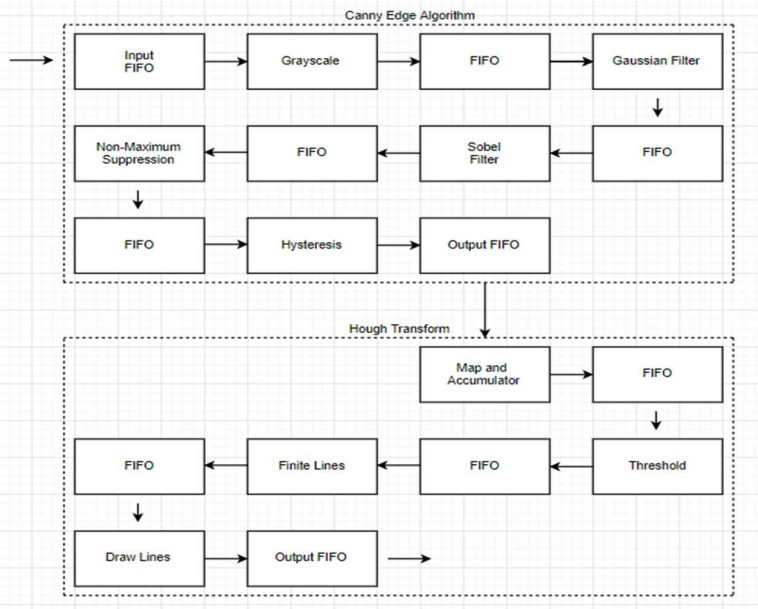

Figure 1: Overall Block Diagram of the Design

First In First Out (FIFO): The FIFO is a component that is used in between every component because it is essentially a memory buffer that is used to temporarily store data. The first byte written into the FIFO should appear as the output. To write the data into the FIFO, the write_en signal needs to be high. When the full flag is high, the FIFO's memory is full and will not accept any more writes until the data is read using the read_en input. Data written while the full flag is high will be ignored. Once a byte has been written into the FIFO, the empty flag will go low. To read the next byte from the FIFO, the read_en signal will be high for one clock cycle and the next byte of data will be available to be read on the next cycle. When the last byte of data is pushed into the data_out bus, the empty flag will go high.

1. Canny Edge Algorithm

Top Level: The Canny Edge Algorithm processes a colored image and leaves only the thin edges within the image. The top level Canny Edge component has a dataflow as described in the block diagram, with five components performing calculations and a FIFO component surrounding each of these components. Each component has the same I/O. It takes in an input of a 'clock' signal, 'reset' signal, the 'in_empty'/'out_full' signals from the intermediate FIFO's, and the 'in_dout' signal which is the 8-bit pixel output from the previous non-FIFO component. The component then outputs the 'in_rd_en' and 'out_wr_en' FIFO signals, and the resulting 8-bit output resulting from the component's calculation to the next intermediate

FIFO. These six components are described in the following sections. The Gaussian filter, Sobel filter, non-maximum suppression, hysteresis components all look at a matrix of magnitudes of the pixels; this is a 5x5 matrix for the Gaussian and 3x3 matrix for the other components. Each component performs a different calculation on this matrix window but are designed the same in terms of memory and state machine. The components each use a shift register that stores (image width *(matrix height - 1) + matrix width) pixels. It stores two full rows of the image and the three or five pixels being used in the component calculation. When we finish the calculation for one pixel and move to the next, the top left pixel of matrix is removed from the shift register and the pixel to the right of the bottom right corner of the matrix is shifted in. In this way, we can save memory and save only a small portion of the image at a time, instead of the entire image. Each of these components also uses a state machine with four states. The first state is the reset state, setting the pixel pointer back to (0,0) and clearing the shift register. The second state is for the beginning of reading and writing an image. When the input FIFO is not empty, it starts to built the shift register with the pixels coming in and outputs a pixel with magnitude 0 since the first row of pixels all have a magnitude of 0. When the pixel pointer reaches the first pixel for which we need to calculate a new value that is not 0 (not the first row or first column), we move to the third state. The third state is the state used to perform the necessary calculation and only operates when the input FIFO is not empty and the output FIFO is not full. It takes the pixel input and adding it to the shift register and removing the old pixel that is no longer needed. It also outputs the value of the pixel obtained from the component calculation. It moves the pixel pointer to the next pixel in a row, or if it is the end of the row, the pointer is reset to the beginning of the next row. When the pixel pointer reaches the pixel in the second to last row and the third last pixel in the column, we move to the fourth state. The fourth state is for at the end of the image and operates when the output FIFO is not full. It does not adjust the shift register at all since there are no longer pixels coming in and just outputs the calculated value of the pixels for the remaining pixels.

Grayscale: Our design takes in each pixel by pixel from the colored image using our ‘First In First Out’ or ‘FIFO’ component every clock cycle. Because it runs through a FIFO component, it takes in an input of the clock, reset, an ‘in_empty’ signal that is high when the image has fully passed through, an ‘in_dout’ signal that passes through each colored pixel in its 24-bit RGB standard logic vector, an ‘out_wr_en’ signal that is high when the ‘out_full’ signal is high, and an ‘out_din’ signal that outputs an 8-bit grayscale pixel into the next stage of our design. Within our ‘grayscale’ architecture, we have two processes, the first process being a clock process named ‘clock_process’ which is basically a process that pushes the main ‘grayscale_process’ through its states every clock cycle. The main ‘grayscale_process’ is made of three states. The first state breaks down the 24-bit input pixel value and breaks it up into three 8-bit sections, and adds each of these together after converting them to integers and divides the sum by three to get the final grayscale value. Then this integer grayscale value is converted into a standard logic vector and sent to the Gaussian Filter component of our design.

Gaussian Filter: This part takes in an input of a 'clock' signal, 'reset' signal, the 'in_empty'/'out_full' signals from the intermediate FIFO's, and the 'in_dout' signal which is the 8-bit pixel output from the grayscale. Gaussian Filter is used to smooth the image by modifying noise pixels. It takes in a 5 by 5 grid of the central pixel and assign different weights to each pixel and get the average value, which is the new central pixel value. In this way, the image is smoothed so it will be more accurate for feature steps. The idea is basically the same with sobel algorithm. It takes in enough pixels in a shift register and starts working. Each time it shifts in a new value and shifts out the oldest value. For gaussian filter, the size of shift register is 4*width +5. So each clock cycle it choose 25 values from the shift register and multiply it with some certain weight and calculate the average value to fill in the central pixel. And when it comes to edges, it just compute part of the matrix and ignore the parts outside of the image. In this way, the noise in the image is eliminated.

Sobel Filter: Same as the Gaussian Filter, the Sobel Filter takes in an input of a 'clock' signal, 'reset' signal, the 'in_empty'/'out_full' signals from the intermediate FIFO's, and the 'in_dout' signal which is the 8-bit pixel output from the Gaussian Filter. The component then outputs the 'in_rd_en' and 'out_wr_en' FIFO signals, and the resulting 8-bit output of the sobel filter to the next intermediate FIFO. The sobel filter is used to calculate the gradient. It calculated both the vertical and horizontal gradient and combine them together so that the area with high intensity gradient are the edges and will be highlighted into white and area with low intensity gradient will be turned into black. In this way, we can identify the edges of the image. In every clock cycle, it takes a 3 by 3 matrix from a shift register whose size is 2*width+3 and multiply it both with fixed vertical and horizontal matrix and adds them up. This results represents the gradient at the central pixel.

Non-Maximum Suppression: Non-maximum suppression works to thin edges after the Sobel filter produces an edges-only image. This component takes a 3x3 matrix of the magnitudes of the pixels and looks at the pixel in the center and compares it to the eight surrounding pixels. For a single pixel, the line going through the pixel can be in the north-south, east-west, northeast-southwest, or northwest-southeast directions. To determine which direction the line through the pixel is going, we compare the sum of the north-south pixels, the sum of the east-west pixels, the sum of the northeast-southwest pixels, or the sum of the northwest-southeast pixels. We know the line is in the direction in which the sum is greatest. For example, if the line is in the north-south direction, this sum will be greater than the sum of the east-west pixels, the sum of the northeast-southwest pixels, or the sum of the northwest-southeast pixels. After determining the direction of the pixel, we find if that pixel has a magnitude greater than those to the directly left and right of it when looking in the direction of the pixel. For example, if the line is in the northeast-southwest direction, we determine is the pixel's magnitude is greater than the magnitude of the pixels both northwest and southeast of it. If the pixel's magnitude is greater than these two surrounding pixels, we keep the value of the pixel; if not, we blacken the pixel and set its value to 0.

Hysteresis: Hysteresis also takes a 3x3 matrix of the magnitude of the pixels. Hysteresis preserves a pixel if the pixel exceeds a high threshold, or if the pixel exceeds a low threshold value and there exists at least one adjacent pixel. In terms of the component itself, we used the same naming system as the non-maximum suppression, and execute many comparisons. We first compare whether if all the surrounding pixels are greater than the high threshold, and if the current pixel is greater than the low threshold. Then we get that value and see if the current pixel is also greater than the low threshold. Finally, we know that if the resulting boolean or if the current image is greater than the high threshold is true, then we will keep the pixel. If not, we do not preserve the pixel.

2. Hough Transform Algorithm

Top Level: The top level of the Hough Transform Algorithm is a wrapper entity that takes in the output of the Canny Edge Algorithm as an input and outputs an image with the edges of the road highlighted in a red line. Each pixel of the input image, as an 8-bit binary number is sent through the accumulator, threshold, and finite lines to give two points for every significant line on the image. Finally, the lines are then highlighted in red through the Draw Lines entity. Each component in the Hough Transform uses a FIFO as an intermediary entity to control the streaming of data.

Accumulator: The accumulator entity indicates all possible lines on the image and represents them in polar coordinate system. For each pixel in the image, the pixel could be on 180 different lines (we set that a line can have 180 different angles to the central point), so, for each pixel, it calculates the distance to the central point 180 times and adds 1 for the corresponding unit in accumulator. After running through the whole image, the output of the accumulator would go into the Threshold entity using an intermediate FIFO.

Threshold: The threshold entity loops over the entire accumulator to get all points whose value is higher than a certain threshold value. This component takes a 7x7 matrix of the values in the accumulator and looks at the pixel in the center and compares it to the eight surrounding values in the Hough space. If the value is a local maxima, we output its rho and theta coordinates in the Hough space. If a value is not higher than a threshold and is not a local maxima, we move on to the next value in the accumulator.

Finite Lines: This component takes a rho and theta value from the threshold block and converts them into endpoints of lines in the rectangular coordinate system. Taking the rho value, the sine of the theta value, and the cosine of the theta value, this block calculates four numbers that represent the two coordinates that are the endpoints of a line: (x1, y1) and (x2, y2).

Draw Lines: The input image is sent to this entity to be outputted into either the SRAM. After the original image has been saved to the SRAM and the endpoints of the lines have been calculated using the Finite Lines entity, we calculate the points on the line between these two points. We calculate the slope and output a red pixel for every point on the line.

This pixel can then be written to an address in RAM to overwrite the original image's pixel value.

Design Optimizations:

In order to speed up the Cordic process which was used in the Hough Transform to calculate angle, the Cordic entity was split into 16 pipelined stages. In addition, our initial design was to take an image with a resolution of 1280 x 720, but because that design needed all the registers and buffers to be a size large enough to take in an entire row, the initial design was too large to be synthesized on the board. The problem was solved by reducing the buffer and register sizes, but instead taking in a smaller resolution image.

Testing / Simulation:

Canny Edge Testing: To test the Canny Edge components, we use a universal test bench, since all components have almost all of the same I/O. The only difference is grayscale component has one input signal (to represent the magnitude of a pixel) that takes 24 bits, whereas the same signal in the other components takes 8 bits. Each component is tested using a higher level entity that wraps the component in two FIFOs, one FIFO feeding the input of the component and one FIFO receiving the output of the component. The test bench reads an input bmp image that the component would be expected to receive and feeds each pixel into the entity with the component and its two FIFOs. The test bench then takes the output of this entity and saves the pixels into an output bmp image, and it also takes a bmp image that is the expected output of the entity and compares each actual output pixel to the expected output pixel. The test bench logs the errors and then outputs the final total number of errors and cycle count when the simulation is finished.

We used the output images of the C source code of the Canny Edge algorithm (from which we based our algorithm) as a comparison for our output images. In our test benches, we ensured that the images after every stage were almost identical. All of the components produced an output image identical to the compare image except for our Gaussian component, which produced pixel values that are off by 1 unit of magnitude. On the top level, the total error count is 11725 pixels out of 518,400 total pixels, which is about a 2% error.
In general, we knew that all the comparison tests through the test bench should take around 514,000 ns, which makes sense because each clock period was 10 ns and the test image was 720x720 pixels. After getting the results, all the tests were within expectations.

Figure 2 : Canny Edge Input

```
# @ 5220235 ns: 518399: 00
# @ 5220245 ns: Simulation completed.
# Total simulation cycle count: 522021
# Total error count: 11725
```

Figure 3: Canny Edge Simulation Results

Figure 4: Canny Edge Output

Hough Transform Testing: To test the Hough Transform, each component was tested individually. This was done by testing the output values of the Accumulator, Threshold, and Finite Lines from the output values from the C code. The difference in the values were output in the test bench terminal. Finally, the image output from the Draw Lines entity was compared to the final image output of the C code. There was a small difference of 1 in the values from the accumulator, but in terms of the overall image, there seemed to be no significant difference.

Overlaying the original image and lines: In the C source code, we output the original image with red lines overlaid to indicate the calculated lines. We instantiated an SRAM block in our test bench to save pixel values at each location before projecting the image back out. The Hough Transform component sends the original image into RAM, and it overwrites the pixel locations that are part of a line. We output one pixel value at a time, and if we do not store the image into memory, we cannot change pixel values to red to indicate lines since some pixels will have already been outputted onto the final image. After the entire image is processed, we create an image from the RAM contents and compare it to the original image.

Top Level Testing: To test the top level, the output of the Hough Transform was compared to the output of the C code implementation of the Hough Transform by running an image with the tracks through the entire process. The image result of the simulation is shown below.

Figure 5 : Result of Simulation of Hough Transform and Top Level

Implementation / Synthesis:

The platform that was used was the Altera DE10-Nano FPGA Board. The board itself has a 1GB DDR3 SDRAM and an 8GB SD Card. The input of our design consisted of a single frame of a video, and the output were three image outputs, one was the image from the Canny Edge and the other two were images with the lines drawn from the Hough Transform. Further implementation/synthesis of each of the algorithms are further elaborated below.

Canny Edge: For the images size of 720 x 480, we can successfully synthesis it but the problem is we need to transfer image data on the FPGA board. So we decide to use UDP(User Datagram Protocol) stack to connect the board through Ethernet port. But we found this involve too much net work level experience about server and protocols. So we don't have enough time for that. Our solution is to run ModelSim simulation instead to check if our algorithm works. This is basically what we did before but we made the image size smaller so that our Hough transform won't take too long to run.

Hough Transform: As shown from the results above, the Hough Transform algorithm succeeded in drawing the lines of the edges of the tracks. However, the Hough Transform gave many problems with the actual synthesis on the board. When we tried the same image size of 720 x 480, the area and memory of the design was too large for the DE10-Nano board. This was probably due to the size of the intermediary buffers and registers that increased in size with the change in size of the input, especially from the accumulator as the accumulator has to store a matrix size of 720*sqrt(2) x 180. Thus, in order for the design to be able to be synthesized on the board, the size of the input image had to be smaller. In terms of the simulation, the 216 x 216 image of the tracks took over five hours to run, so due to timing constraints, we did not run the algorithm on a high definition image.

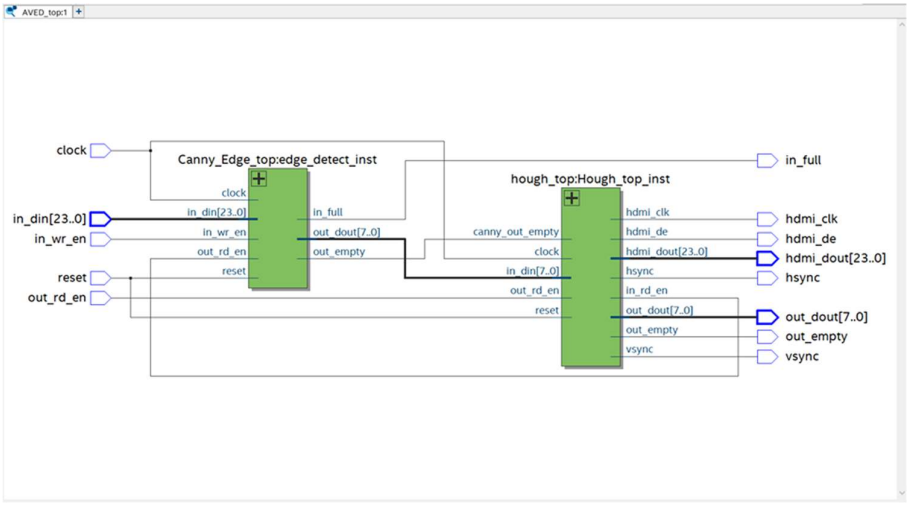

Figure 6: Quartus RTL Synthesis Results

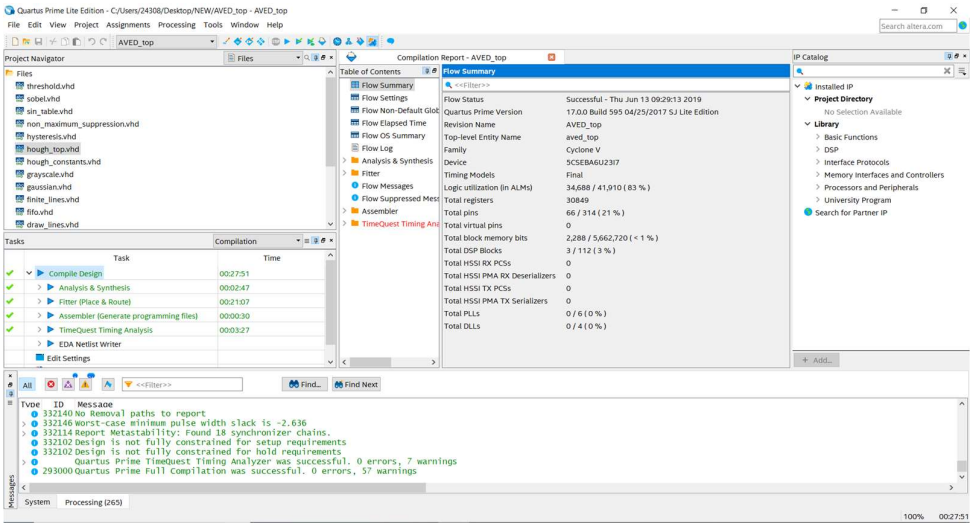

Figure 7: Flow Summary Results

Figure 8: Test Simulation Results (216 x 216 image)

Conclusions:

Our design is able to take in an image and apply both the Canny Edge and Hough Transform on it to produce an image from the Canny Edge, and two other images with the lines that highlight the edges of the road. However, this did not work to the original design specifications because we were not able to use an Ethernet cable to send the packets of data from the computer to the FPGA. Also, although we were able to synthesize the design, we were not able to do it with the original 720 x 480 image size because it took up too much memory and area on a DE10-Nano board. We solved this problem by using a smaller image size to reduce the size of the accumulator. Finally, this implementation of the edge computation of the sides of a road needs to be faster, because the run time of the design was extremely long. For an automated car that needs to calculate where it needs to go in real time, this implementation would be too slow. For future improvements, we would need to do some optimizations to this design. One example would be to remove all “integer” types and replace them with binary values that are a specific length, because integer types are always 32 bits wide.

References

DE10-Nano Kit Specifications. TerasIC, www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=1046&PartNo=2.

Keymolen, Bruno. "Hough Transformation - C Implementation." *Bruno Keymolen Blog*, 3 May 2013, www.keymolen.com/2013/05/hough-transformation-c-implementation.html?fbclid=IwAR3-2EHNLYbxQrXNI4D5t9ImgNS7PqgJ9ek2kHAKsVRrcKR_xLg9NnTRLCg.

Zaretsky, David. "Advanced VHDL - Lecture 7: Computer Vision with VHDL." 4 February 2019, Northwestern University. Microsoft PowerPoint Presentation

Zaretsky, David. "Advanced VHDL - Lecture 9: Quantization with VHDL." 17 February 2019, Northwestern University. Microsoft PowerPoint Presentation