Sign Detection System for Real Time Applications

Brock Harris
Union College - Schenectady, NY

William Christensen
Union College - Schenectady, NY

Sign Detection System for Real Time Applications

William Christensen and Brock Harris

Advisor: Prof. Cherrice Traver

March 2020

Summary

Project Topic

For our senior capstone project we chose to study how hardware-software co-design can be utilized to improve image processing task such as object detection. We chose to implement subsections of the histogram of oriented gradient algorithm on a field programmable gate array to produce compressed features that can be transferred to hard processor system using a data interconnect.

Goals

Going into the project we knew that completing the algorithm totally was not achievable within the limited time frame that we had. This is why we wanted to focus our efforts on the aspect of the project we knew the least about to further develop our understanding of the development process for these types of systems. We had 3 major goals for our project. Our first overall goal, was to understand and facilitate the communication between the hardware and software components. next we wanted to be able to do some image processing on the Field Programmable Gate Array (FPGA) so that we can prove the functionality of the hardware aspect of the algorithm. And finally we want to have some working code in software to show that data can be transferred from the hardware to the Hard Processing System (HPS) which requires using a specific data bus called an Advance eXtensible Interface (AXI) bus, using this the HPS will be able to access data from the hardware as if it were stored in memory.

Design Requirements

We had 3 major requirements for our design. The image processing task done on the FPGA needed to operate in real time. This means that each frame of the image needed to be processed before the next frame came in. Next it is important that we are able to produce data that is useful. We need to ensure that the image processing step on the FPGA produces the correct data so that the important information can be taken out of the image. And finally we need to ensure that the data being transferred from the FPGA to the HPS is the same on both sides of the transfer.

Design Overview

For this project we are using a De10-standard Soc board which allows us to have the FPGA and HPS are on the same chip. This means the FPGA has a direct path to the HPS without requiring extra wiring or any external work beyond programming the FPGA to connect directly to the HPS. We start by reading data into

the FPGA from a camera and processing the data on the FPGA. From the FPGA pixel data goes into a vga monitor to display the image that we received. The data also goes to the HPS to do further computation with the ultimate goal of detection.

Results

Despite challenges along the way were were able to meet most of the initial design requirements. The image processing tasks performed on the FPGA such as Sobel filtering and gradient computation can both be performed within the time requirements and the data can be transferred from the FPGA to the HPS with few errors in data. Unfortunately, while we have a Sobel filter that we believe to work we have not had success having it work within our system using data streamed from our camera.

Contents

List of Figures

Introduction

Most modern cars have features that allow them to drive in certain circumstances without human input. To accomplish this semi autonomous driving capability, many companies have put multiple sensors in their cars to detect obstacles in all directions. The most important sensor in most cars today is a camera. Unlike other sensors cameras are able to capture a level of detail beyond the shapes or distance of objects. using cameras, we can see what is written on a signs, walls, or billboards. These messages, while generally meant for drivers can still provide a great deal of useful information to the car if interpreted correctly.

Cameras are sensors that capture a snapshot of an environment into a two dimensional array of values. Each of these values is called a pixel, and each pixel is made up of three color channels, red, green, and blue. Not every pixel has important information for a driver, so the challenge for developers is how to extract only the most important information in the least amount of time possible. Processing images can be very time consuming because it often requires a processor to move through every pixel and decide if it fits into a pattern that is useful or not. With full HD images (1920x180) there are 2,073,600 pixels to process and, more often than not, each of these pixels needs to be processed more than once to gather the needed information.

Because of the large amount of data that needs to be processed, finding algorithms that can deal with image in a time sensitive manner can be challenging. We want to create a prototype of a system that will pull the important information out of an image so it can be used to easily detect objects in real time and report its findings back to a driver of the car or to a central system. We have chosen to detect stop signs for this prototype. For this we plan to use Computer Vision algorithms that have been shown to work with similar problems. However we also know that these algorithms, while optimized, are still too slow. To remedy this we have planned to use special hardware to run parts of the algorithm.

In this report we will start by going into more detail about how what is require to process images and how that can be done using and hardware software combinations. We will then talk about the specific design requirements for our project and what we hope to accomplish with the project. An then we will discuss how we chose to implement our project and some alternative that we chose against.

Background

Driver assisted technology has and will effect many different parts of our lives. It has a significant effect on our safety, the cost of our cars, and policies and laws for our roadways. In this section we will briefly discuss the background these technologies have in these three major categories and then we will briefly discuss the technologies that are used to make these devices.

Safety

Driver assisted cars have been large field of research in the past few years. These technologies have the potential to significantly increase the safety and reduce the effort of a driver while driving. In 2019 37,133 people died in car crashes in the United States. Of those people 23,757 were do to human error[9], and the largest cause for crashes we drivers failing to recognize hazards[1]. This shows that many deaths could be prevented if some human error could be eliminated from driving. Technologies that inform the driver of oncoming obstacles have the potential to be a great asset to everyone on the road.

Vehicle cost

These technologies, while incredibly useful, come at a cost to the driver. These sensors are often expensive and located on the exterior of the car. The makes any crashes that do happen much more expensive. Front facing collisions on cars with the auto brake feature cause the average payment per claim to go up by $104 compared to vehicles without this feature[5]. The cost of other parts of the also goes up do to these technologies. The cost of replacing a 2017 Subaru forester windshield is $910. Just 5 years ago, a similar replacement would cost $150[19]. This added cost is due to the camera put behind the windshield. The camera needs special glass so that its vision is not obstructed. It also needs special technician to come and calibrate it before it is safe to use. These cost are becoming a necessary evil as more and more cars opt to take advantage of these new technologies.

Political

This topic has led to a significant amount of political discussion about how safe technology really is. The basis for a lot of these devices is machine learning and AI. Both of these algorithms leave room for error and errors are actually expected. This, rightfully, scares a lot of people in the context of driver-less cars[20]. After many publicized crashes by Teslas, people are worried about how effective these cars might be. If the person at the wheel is putting their trust in a car, and not paying attention do they pose more of a threat to the others on the road. This is why the NHTSA is putting a standardized classification on these cars, and regulating what level of cars can be produced and what criteria need to be met before they can be driven on the road[14].

There are a few items that require discussion before beginning an in depth look into the final design of the sign detection system. The topics of Computer Vision and image processing will be brought up, but the difference between them is very important. Image processing takes in some image and outputs a modified image in some way. Computer Vision is a more complicated idea which may use image processing in order to accomplish its goal but ultimately is not necessarily image processing. Computer Vision focuses more on what a machine can see in an image based on patterns, and takes the input image and outputs some task-important data.

Computer Vision

Research behind Computer Vision began in the 60's by Artificial Intelligence (AI) researchers, like those at MIT in 1966[18]. From this, there has been significant advancement towards incredibly accurate and advanced Computer Vision systems that can identify any number of things. These systems have used a number of different algorithms to identify what they wish to find, whether it be motion, an object, or similar images. For the purposes of this project, we require a specific type of computer vision for object detection in order to detect stop signs.

This type of Computer Vision usually has a few steps involving image processing, feature extraction, and classification. The preprocessing is fairly simple and involves a few basic functions to change the image so that features are more easily extracted. There will be further discussions of feature extraction and object classification.

Feature Extraction

Feature extraction refers to a broad idea of taking image data and converting it to a more compressible set of data that is more meaningful to a computer where features like large changes in pixels are more prevalent and small changes or noise in the image can be ignored. An example of this kind of algorithm is Histogram of Oriented Gradients (HOG), defined by a Dalal and Triggs in 2005[12]. The HOG algorithm was created to allow for efficient people detection. It operates by extracting groupings of gradients, which are changes in pixel values, into histograms in areas across the entire image. This is just one algorithm that could be used for feature extraction, and is the algorithm selected for this project. The reasons for this, will be discussed later.

Object Classification

Object Classification refers to the process of taking the processed data that came from an image, and using some algorithm or function and determining whether the image contains a specific object. This requires a pattern recognition algorithm that needs to be trained on a labeled data set. One commonly used classifier is called a non-linear Support Vector Machine (SVM) created in 1992 by Boser, Guyon, and Vapnik[11]. This is an algorithm that allows for classification of data using a kernel method to handle non-linear classifications for more accurate classifications. The specifics of the SVM which is used in this project will be discussed in the design section.

Hardware Acceleration

Hardware Acceleration is a process in which algorithms can be accelerated by parallelized processes. This acceleration is done on specific type hardware, usually on a Feild Programmable Gate Array (FPGA). FPGAs are very good at increasing the speed of highly parallelizable algorithms. One of the most successful algorithms that has been accelerated like this is the HOG, discussed earlier. Because, this algorithm, while very effective at detecting people without using a neural network, was still very slow. Advani et al.[8] showed empirically that the algorithm could be accelerated significantly by using hardware acceleration. They also showed that FPGA implementation was the best way to implement this algorithm compared to standard computing. Hahnle et al. [13] created a system that uses the accelerated hardware capabilities of the FPGA to create a real time pedestrian detection system on high resolution images. Their system worked on full HD images and classified an image in under 150 µsec. Ngo et al. [16] also boast a very fast extraction rate of 526 frames per second per image from their results which shows an incredible speed for an an algorithm running on this kind of data size.

Ethical Consideration

The use case of this project provides multiple ethical considerations that must be addressed. The goal of our project was to design a system that could be used in a car to assist drivers. This has been a controversial topic in the past few years because cases have come up where companies, such as Tesla, have had cars' autopilot fail causing the driver to die[**?**]. While we do not intend to do any detection in our project, we also know that it is crucial that accurate data is processed and ready for prediction in a time that is reasonable. If system takes to long to process and image it may not longer be relevant.

currently no car on the market that allows the driver to not pay attention. There are 5 levels of automotive driver assistant[14]. We can only achieve level two automation with currently regulations and technology. This means that the vehicle can perform some task but the driver must be engaged at all times. This is important to note for anybody developing this technology. If the system makes a prediction, what will it do? It can notify the driver so that they have more info or it could try to make a change itself which reduces the things the driver has to worry about but also increases the risk.

Design Requirements

For this system, we need to be able to take an image and extract the features that would allow us to determine if there is a standard American stop sign in the image. Because stop signs often are not uniform we need the system to work on all sign regardless of the level of vandalism that might be on the sign. If the sign is clear to the user, it should be clear to our system.

Because we want this system to be useful in a driver assisted setting, we want this system to be able to pull the important features out of a 640x480 image in real time. This means that each image needs to be processed completely before the next image comes in. We want our system to work on cameras that operate at 30fps. This means that the image must be classified in under 3.3ms. We also need it to have a acceptable level of accuracy, somewhere around 75%. We need to the system to produce the same features, as a python implementation performing the same tasks. It is important that we produce data that is very accurate but it is probably more important that we do it quickly.

Design Alternatives

During the design stage of our project we looked into many different solutions and different iterations for each part of the project. We will be discussing each of these below.

Camera type

The camera was one of the first things we looked at in the entire project. As it is the only input we have for our system it was important to choose a camera that had good enough specs while also being simple enough to integrate into the final system. We looked primarily at two options a USB Camera, and FPGA specific camera.

USB Camera

A USB camera or webcam could be used and would operate as a UVC or USB Video Class Device and would operate by sending video in a multitude of different video formats and color formats [3]. This would be difficult to do because many of these devices are designed for a consumer who doesn't care about specifications or about what communication protocol is being used. This means there is no data on which format or how data is being sent from the the device and what we could use to decode and use the data for feature extraction and classification.

FPGA Camera

The FPGA compliant camera is a camera created by the producers of the selected System on a Chip (SoC) and is designed to provide direct uncompressed digital picture data and the formatting of this data is well documented and allows the camera to be easily integrated into the project. However because of this special camera is more expensive and camera specs may be lower than what we could find for a market ready camera of the same price.

Our Choice

We chose to used the FPGA camera. This camera provides more support for the hardware we are looking at. This support allows us to focus more effort on the algorithm. The price of the camera is a small price to pay as USB webcam communication adds a significant amount of complexity to our project.

Color space

The color space is how the color information is represented as we pass the image through the algorithm. The camera we have chosen in the sections above uses the bayer color scheme. With is a modification of the RGB color space. it can be useful in some circumstances to convert the pixels into a different color space before processing them. This conversion, while adding some computational time, can lead to an increase accuracy of the system.

Grayscale

This removes all the colors from the image uniformly so that the image is made up of shade of gray. Reducing the image to grayscale reduces the performance of the algorithm by 1.5%[12]. Because of this, we decided to look into other solutions like RGB or LAB which allow for full color imaging.

RGB

The RGB color space consists of all possible colors that can be made by the combination of red, green, and blue light. It's a popular model in photography, television, and computer graphics.

LAB

Cie-L*ab is defined by lightness and the color-opponent dimensions a and b, which are based on the compressed Xyz color space coordinates. Lab is particularly notable for it's use in delta-e calculations.

Our Choice

We chose to use the RGB color scheme, because [12] showed that the color scheme does not have a large impact on the accuracy. This also means that we have to do minimal color conversion before feeding the pixels in to the HOG algorithm as we could simply select a common camera with RGB output.

Feature Extraction Algorithm

The purpose of this algorithm is to take an image and create feature that could be useful to a machine learning algorithm to detect something in an image.

Histogram of Oriented Gradients

HOG starts by taking an image and converting it to gray-scale or by taking the maximum color value for that pixel and making that the gray-scale value. Once in gray scale it moves over the image from left to right then top to bottom looking for gradients in the image. A gradient is a place were the first pixel moves from a low pixel value to a high pixel or a negative gradient when a pixel moves from high value to low value. For each pixel a direction vector is created that represents the direction of the gradient for that pixel. The once the left to right and top to bottom gradient vectors for the pixel are created they are combined to get the overall gradient vector for that pixel.

Once every pixel has an overall gradient vector there are put into groups of 3x3 (something we need to decide) and gradient vectors for that group are put together. Also a bin value is determined so that if two pixel vectors are within say 10 degrees of each other they are added together to make a larger vector in that direction. This results in a block of pixels having a star of vectors in the middle where there might be one or two vectors that are clearly the longest. This start of vectors is the feature that is fed into the support vector machine.

Scale Invariant Feature Transform

This is an algorithm that is commonly used in shape tracking and shape detection. This algorithm starts by finding a series of interest points in an image. These interest points can be found through a edge detector line detection, other methods. It then creates a feature for each one of the interest points by looking at the gradients around the interest points. It then takes these gradient collects them into a histogram and compares them to all the found features of the object.

The algorithm is very effective at tracking object objects in images. It is used in robotic application for mapping rooms by finding fixed points in the image and watching how they move in subsequent images. This is very similar to our ultimate goal. Despite the fact that it could work well for our goal This algorithm does not seem to be easily implementable on an FPGA and would not be as fast to run as the HOG transform. It may also not be as effective in low light level images due to a potential lack of interest points.

Our choice

We chose to use the HOG algorithms because it has been shown to be effective at detecting different kinds of objects and is much better at detecting objects that are not sharply defined. While stop signs are often sharply defined, they also do sometimes have obstructions that could lead to worse performance on SIFT. HOG is also much easier to implement in hardware and would allow for more a speed up.

Classification algorithm

The purpose of this algorithm is to take a list of features from an image and decide whether that image contains the object we are looking for. There are many types of classification algorithms we could have used but we looked at two algorithms that have been show to work on the HOG features.

Support Vector Machines

An SVM is a machine learning algorithm that takes in a set of features from an image and using data from a large data set can determine where an object is in that image. It does this by turning each image that it receives into a point on a plot. It then uses the plot point it already knows the answer to to create a linear function to separate the images with the object and without the object. Once this line is created all the algorithm needs to do is check if the new data point is above or below the line to determine if it contains the object. This algorithm is commonly used in conjunction with HOG and relatively common algorithm, with lots of resources. Tends to provide better results than K Nearest Neighbors (k-NN), another classification algorithm we will discuss after this. However, This algorithm is a little bit more complex to implement than k-NN and may require more training data in order to work well

K Nearest Neighbor

K Nearest neighbor takes a plot of values just like the SVM but in order to determine whether the image contains the object it takes a some amount k of the nearest neighbors to the near plot point. If the majority of those neighbors are classified to contain the object then the algorithm returns a positive identification. This algorithm is relatively simple implementation, and simple to understand. However, May not be as effective as other algorithms due to its simplicity.

Our Choice

We will be implementing the SVM algorithm for the project. We chose this algorithm because we believe that it will provide better results than the k-NN and the resources available will be enough allow us to overcome any difficulties related to implementation.

SVM implementation

When programming the hardware processor to run with the FPGA there are two options for running software on the processor. The software can be run on either the bare metal or using a Linux distribution installed on the processor. Most computers that run software have an operating system that keeps track of all the parts of the program. It keeps track of memory allocation it also makes sure that cores are being used efficiently and a lot of other things under the hood.

Bare Metal

When software is run on bare metal this means that it is run without any operating system to manage it. Running the software on the bare metal process removes all the computation time used to maintain a Linux distribution running on the background, but it does add complexity to the programming that can take extra time that we do not have. Because our program is not very complicated, we do not expect the added complexity to be unmanageable. This method also requires a special ARM compiler. We would need to license this compiler to be able to write code for bare metal. If we cannot get a university discount we will be unable to proceed with this method.

Linux

Linux manages a lot of things related to the programming for the programmer. There is little overhead when it comes to system management. Because of this, using the operating system is a compelling idea. We want to spend most of our time working on the feature extraction section of the project. Linux does however reduce the speed of any program we write on it because it is managing so many things in the background. By the nature of the project, speed is very important to our final goal.

Our Choice

Both sides have pros and cons and either way would work for our project. We are going to program the SVM on Linux Operating System (OS), because the ease of development is important especially for a project with such a short time frame. Bare metal processing tools are significantly more difficult than a Linux OS and are significantly more difficult to test code with. Also the ability to get licences to good tools for developing are very expensive and the only tool that was available was not only overly complex but also had very little documentation order to get them working.

Preliminary Proposed Design

This term we worked on developing a design for the system so that we could get a better understanding for what we will be implementing next term. We cover each aspect of the design below. We start with a overview to give the reader a better intuitive idea of the design and allow for allow for an understanding of the flow of the system. After that we go into materials that will be required. Through this section the reader will get a better understanding of the physical systems that we have chosen for the design. We then go into detail about each of the parts of the system.

System Overview

For our system we chose to combine aspects of work done by Hanhle et al., Advani et al., and Ngo et al. [13, 8, 16] into a system that will allow for both fast computation and ease of development. The system overview can be found in figure 1.

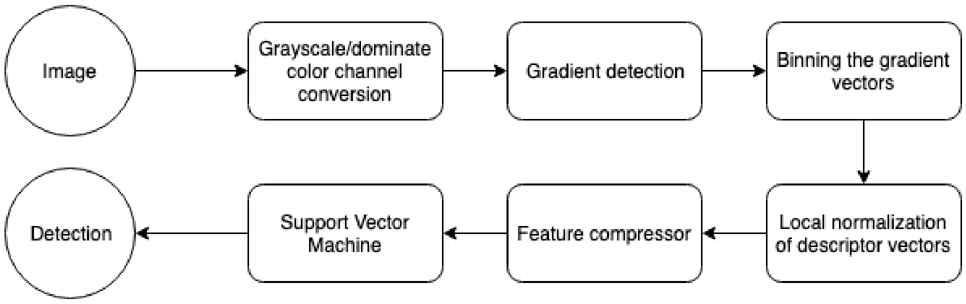


Image
Grayscale/dominate color channel conversion
Gradient detection
Binning the gradient vectors
Local normalization of descriptor vectors
Feature compressor
Support Vector Machine
Detection

Figure 1: Flow diagram for the overall system

We can see from this diagram above that there are 6 major sections of this system. We start with a camera input. The camera we have chosen outputs each of its pixels in a bayer color pattern. This means our first step is to convert these pixels into a gray scale form. We then find the gradient vector for each pixel. Next we create a histogram graph for a group of pixels and normalize that histogram. Once we have normalized it we can compress the data and move it into the SVM to classify the image. The first 5 blocks of this block diagram will be done on an FPGA. The last step, SVM. While the operation of the SVM can be sped up by implementing the algorithm on an FPGA, we have not chosen to do this because the added complexity of implementing it on the FPGA is not worth the performance gain for this project.

Materials & Economic

Our development system will be designed using a Field Programmable Gate Array(FPGA) in order to be able to quickly redesign any hardware that doesn't work or that can be improved. The current design uses a Terasic De10-standard to implement hardware and software with the FPGA and ARM processor on the board the De10 SoC retails for about $350 and would most likely be the most expensive part of the project. The De10 could however be taken from the Electrical Engineering Department's current collection to reduce costs for the project. The project will also require a camera to take input to the system, for this the TRDB D5M be used. This a camera sensor that is compatible with the de10, with a cost of $89.

Histogram of Oriented Gradients

There are 5 major steps involved in extracting HOG features from and image. All of these features will be preformed on a FPGA to allow for parallelizable computation. The first step of the process can be seen below in figure 2. The first aspect is moving the image pixel produced by the camera into gray scale.

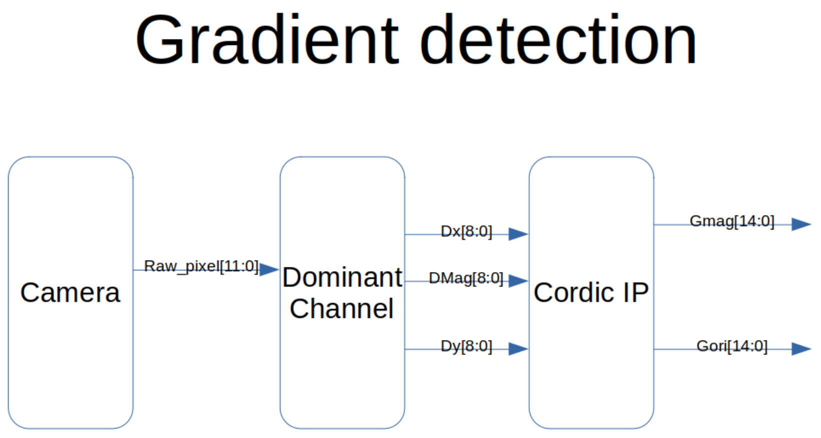


Figure 2: Block diagram for gradient detection system

The camera produces a 12 bit wide signal. This is used to transfer 4 bits for each of the 3 colors in each pixel. Instead of using the average value between all of the color channels to convert the images into gray scale we are using a technique called dominate channel conversion. The goal of this is to encode the change in color along with the change in gradient. We accomplish this by computing the gradient for each color channel and comparing them to determine which channel has the largest gradient. Then the output is 9 bit signed integer value representing the gradient of the pixel and its magnitude. The block diagram for this computation is shown in figure 3. The next step of the gradient detection is to compute the magnitude and direction of each of the gradients. The equations for computing the magnitude and direction of the gradient can be seen below in equations 1, 2.

$$\|G(x,y)\| = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{1}$$

$$tan(\phi(x,y)) = \frac{G_y(x,y)}{G_x(x,y)} \tag{2}$$

Because of the complexity that comes with computing square roots on an FPGA we will be using the Cordic IP block to preform these computations.

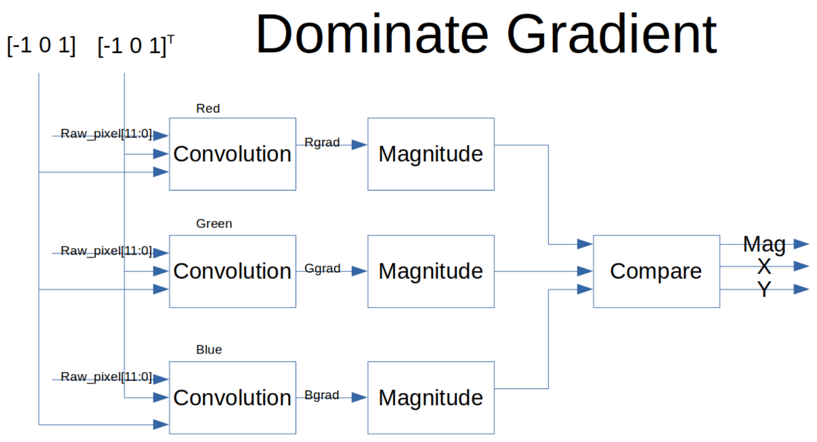


Figure 3: Design diagram for dominate gradient computation per pixel

Through this method we hope to get much more distinct gradient features because of the sharp change from red to white on stop signs. As we can see from figure 3, we start by convolving each color channel with the matrix $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}^T$ Using these matrices we can find the gradient of the color channel. We then compute magnitude using equation 1. Finally we compare the magnitudes and pass the largest one to the Cordic module. The cordic module outputs 15 bit wide signals to allow for 2 fractional bit below the decimal point as specified by the Cordic IP.

We then move on to the histogramming section of the algorithm. The goal of this section is to group all the gradients in an 8 by 8 area of the image into a histogram. This grouping removes noise from the image by making gradients that are not very strong less important and visible than gradients that are strong.

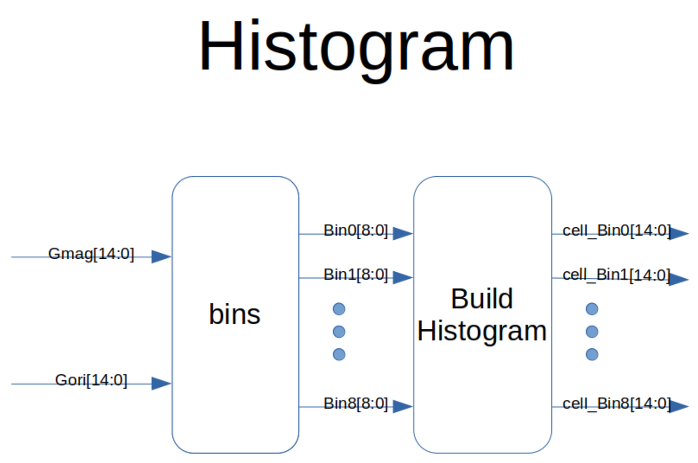


Figure 4: Block diagram for histogram creation

The purpose of the binning block is to take a gradient vector and classify it into a specific bin. To do this advani et al.[8] has create a system that holds a lookup table for bins and stores already computed values for each orientation. This provides a speed increase that is necessary to this section. Unfortunately, the information for this implementation is limited. We hope to gain a better understanding of how to preform this stage in the design through reading other sources. The output of the binning block is a 9 bit integer value for each of the 9 bins that the orientation can be put into.

The next step is to build the histogram. This is done by aggregating all the computed values for each of the pixel within an 8 by 8 window called a cell. By accumulating the orientation values together we are created groupings that will reduce the size of the data we are transferring and reduce noise in the pixel groups. The output of this block is a 9 bit integer value that represents the magnitude of all the gradients for each of the 9 orientation bins.

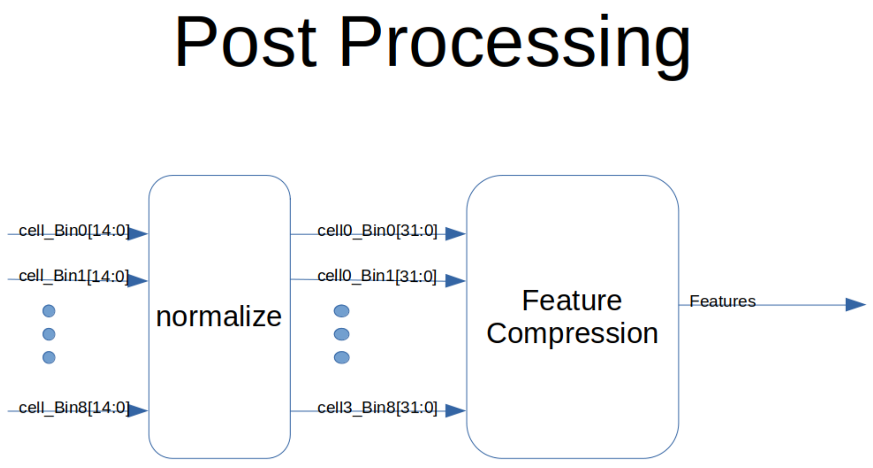


Figure 5: Block diagram for post processing steps

The final step of the process is the the post processing stage. This consists of the normalization of the histograms and the feature compression. Normalizing the histograms is important because advani et al.[8] has shown a significant increase in detection rate when the features are normalize. to do this we preform an *L2-norm* normalization on 4 cells together to reduce the values in histogram to manageable values. the output of the normalizing block is the 9 bin histogram values for 4 cells. After they are normalized it is important that they are compressed. This allows for faster data transfer of the features to the SVM. The block diagram for this section can be seen in figure 5.

Support Vector Machine

For this project we are focusing most of our effort on the FPGA implementation of the HOG because of the scale of the task. Because of this we plan to use a simple implementation of the SVM for testing purposes. To create this SVM we will be using the SVMlight library. This is a fast prebuilt C library for SVM's. If we are able to get a license for the ARM compiler we will run this SVM on the Cyclone V SoC.

Hardware Software Co-design

Hardware, while often defined through a language called a Hardware Description Language, is defined very different from software. The development process for the two elements of the project are unique but they must be used together to get the system to work in unison. One problem that we have encountered is how to combine these two independent approaches to allow for optimal communication. This is solved through a methodology called Hardware Software co-design. This is a way to structure project to allow for ease of development between hardware and software. For this problem we are using a tool called Quartus II. Quartus allows us to develop the hardware design in a language called VHDL, a language that allows for a detailed hardware description without having to think about the underlying logic that would be used. Quartus also allows us to add C code to our project and connect it to the VHDL design. To do this we will be using a tool called Qsys. Qsys, a part of the Quartus Design suite, allows us to define how information will flow from the FPGA to the (Hard Processing System)HPS and back. It also gives us a visualization of how each component will be connected and what kind of data can be transferred at what clock rate. It is important to keep track of details such as the clock domain of the data's origin and the clock domain of the destination. Qsys helps us to organize these details.

Final Design and Implementation

The final design is split into two separate sections. These two sections are the hardware design section implemented by Brock Harris, and the Hardware Software interconnect implemented by Will Christensen.

Hardware

Within the Hardware design of this project we have 3 stages that the data goes through prior to moving to the HPS. The first stage is being able to get the data from the D5M camera into a format that allows for further processing. This is shown in part A of figure 6. The next stage is to run an edge detection algorithm on this data, shown in part B. Finally, we compute the gradients of the pixels in the image, shown in part C. Once we have the gradients we can find the dominate gradients and pass them to the HPS for further interpretation.

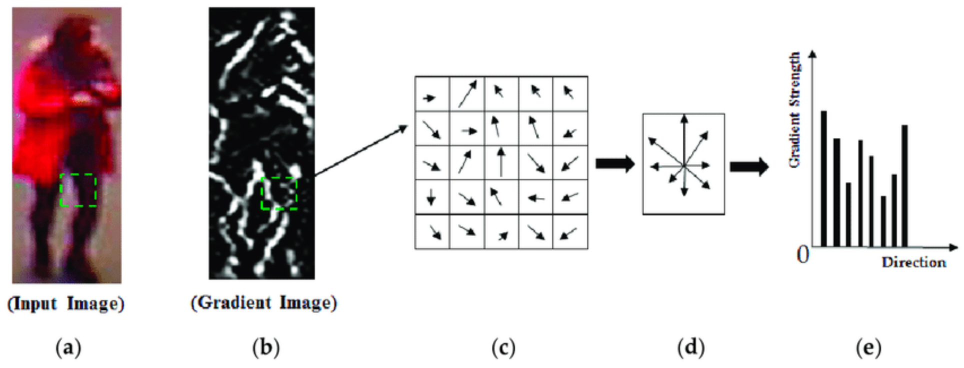


Figure 6: The step of the HOG algorithm visualized[17]

Camera

The first step in our algorithm is to interpret the data from the camera. To do this, we make use of a golden hardware reference design (GHRD) created by the manufacturer of the camera. The camera we are using produces pixels in the Bayer color format. In this format the pixels have 4 separate values with 8 bits representing each value shown below.

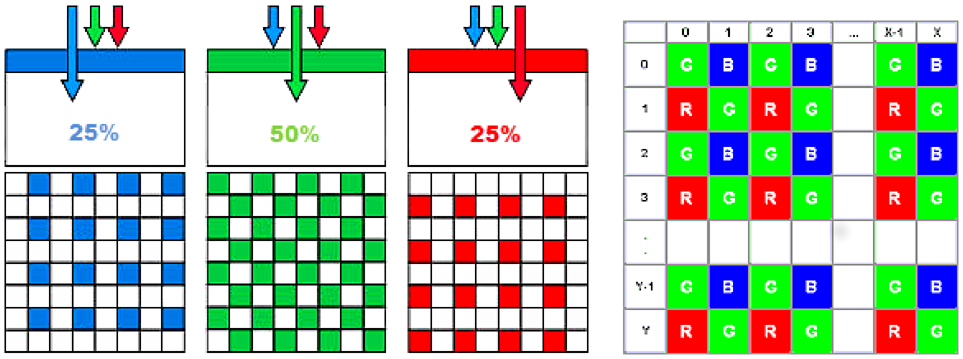


Figure 7: Bayer color pattern

We can see that in figure 7 green makes up half of the representation for each pixel. This scheme is used because the human eye is more sensitive to changes in green light compared to the other colors. This however is not needed for our algorithm and is not suitable for displaying on the VGA monitor so we must convert the values to the RBG color scheme with 10 bits representing each color channel. The RGB color scheme has equal representation for each color represented. This is done using the GHRD provided by Terasic. The block Diagram for this design is shown in figure 8.

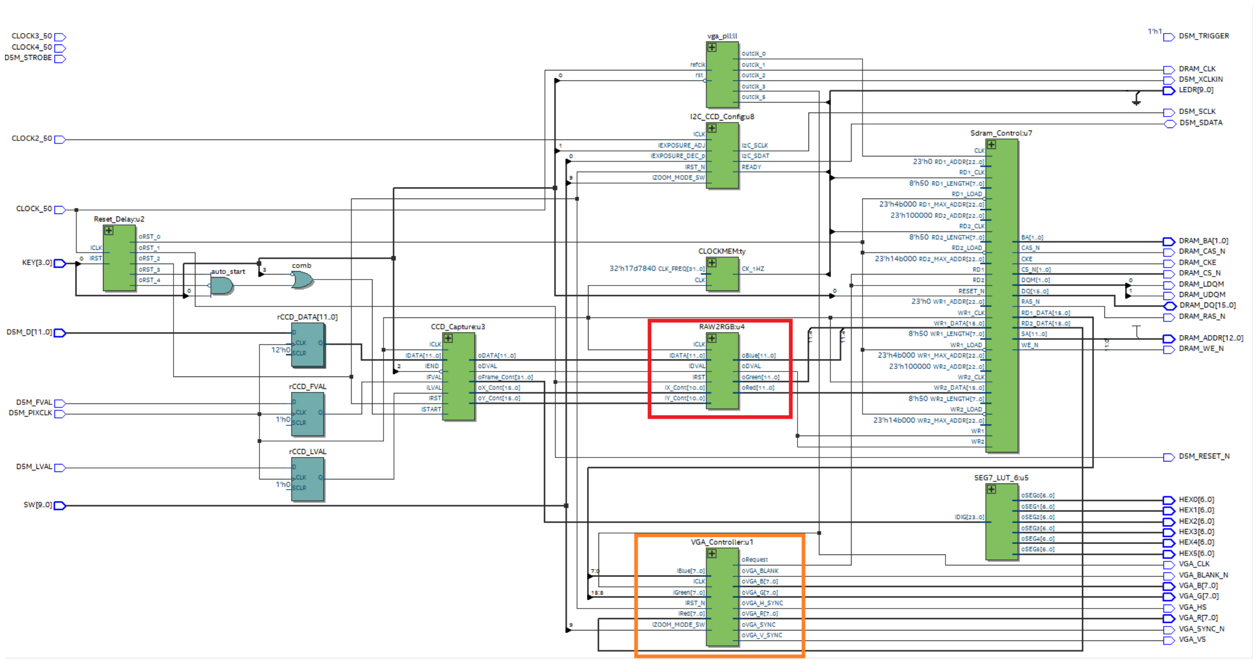


Figure 8: Block Diagram for D5M Bayer to RGB Resampler

There are 2 major functions that this module plays. It converts the Bayer pattern pixel to RGB and it downgrades the resolution so that images can be displayed on a VGA monitor which has a resolution size of 640x480, when the camera is able to capture at a 4 Mega pixel resolution. While this block diagram is large, the two parts that are most important to our project is the output from the RAW2RGB module, inside the red box, and the VGA_Controller module, inside the orange box.

As we can see from figure 8, the RAW2RGB module has 3 major outputs, the Red, Green, and Blue, pixel values represented by 12 unsigned binary values. We can can this pixel stream and the camera clock which is used to set the timing for this module in our own module.

Edge Detection

Once we have the RGB values for the pixels being streamed from the camera we can begin working on the Edge detection section. This step produced a significant challenge that we were not expecting. We went through several potential solutions until we arrive at our final implementation described below.

To calculate the edges of the image we decided to implement a Sobel filter into our design. A Sobel filter is a basic image processing technique used to show only the edges in a gray scale image. The effects of the Sobel filter can be seen in figure 9 below.

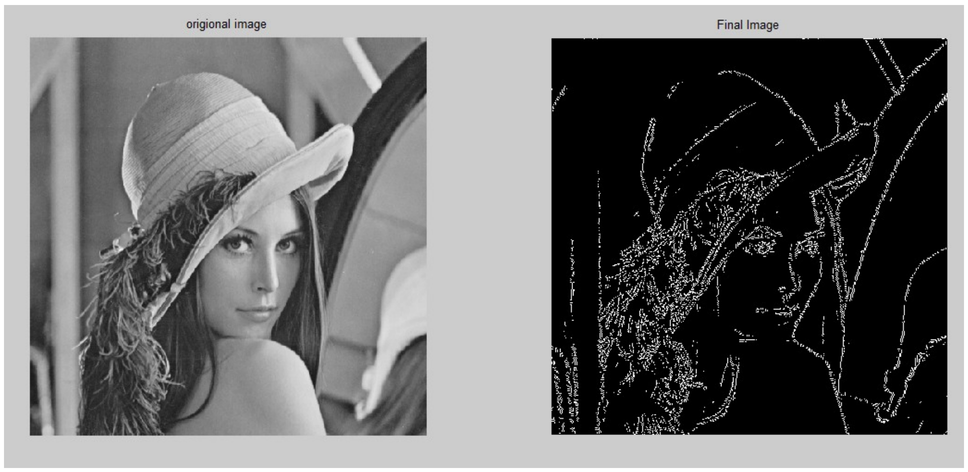


Figure 9: Before and after application of Sobel Filter[10]

As we can see from figure 9, the Sobel filter takes a grayscale image and highlights the edges in the image. This can be thought of as taking the derivative of an image because the brightness of a pixel goes up as the slope of the transition from the pixel to its neighbors goes up. This image is created by passing, or convolving, two filter matrices over the entire image. These two matrices are:

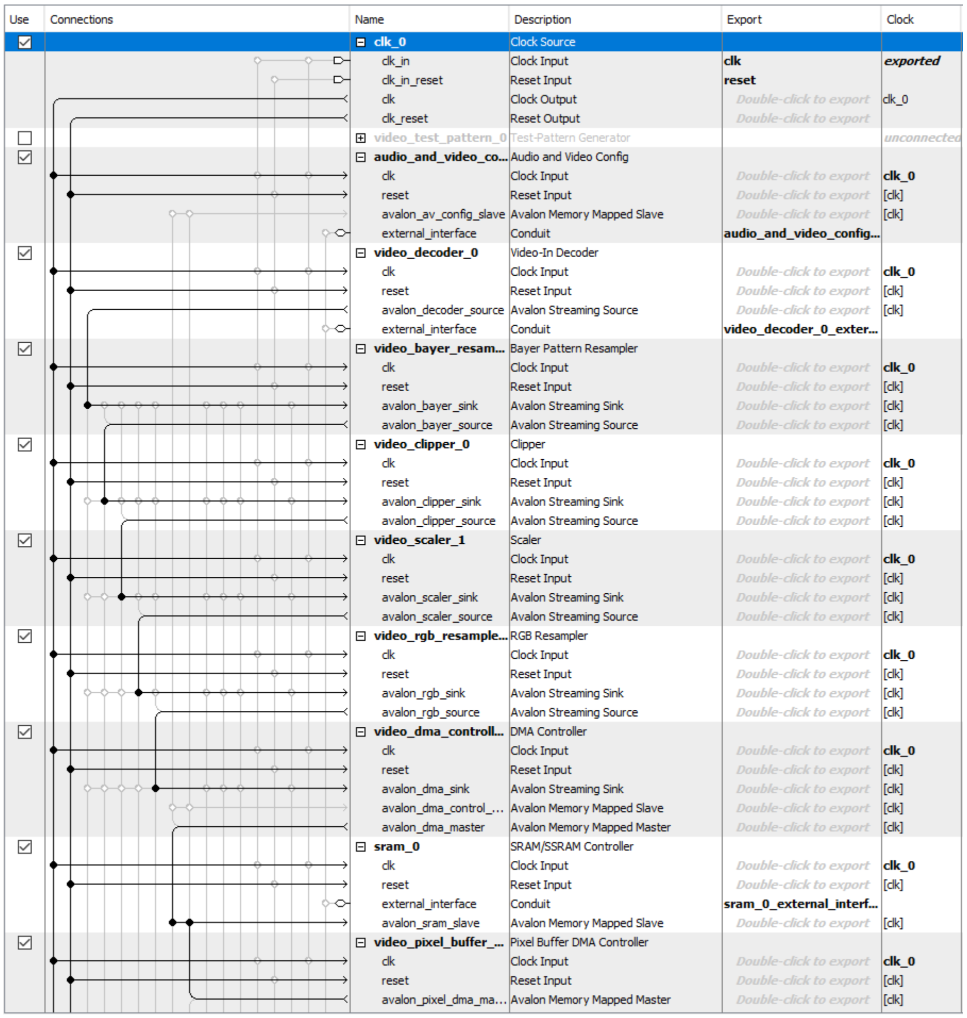
$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} G_y = \begin{bmatrix} -1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix}$$

each matrix represent a direction of change in image. if there is large pixel magnitude change between pixels when moving left to right it will be detected by the G_x filter, likewise the G_y filter will detect pixel magnitude change in the vertical direction. To compute these transition we pass the matrix over each pixel in the image multiplying its immediate neighbors with indicated weights and summing all the values together, or 2d convolution. Once each detection is computed we can pass those values onto the gradient detection step.

We looked into several methods to implementing this process. We found three potential methods for creating the Sobel filter from the camera data we had. We looked into the 2d FIR II package, Qsys, and VHDL. The VHDL method was the direction we chose in the end.

The 2D FIR II library is a package maintained by Intel for the purpose of 2D signal processing. This package performs a finite impulse response over a 2 array. Through the documentation we found language that implied that we could use this library to implement an arbitrary filter over our image, but this turned out not to be the case.

Solution two was to implement the Sobel filter using prebuilt packages within Qsys, a program within Quartus to connect modules together. Qsys provided modules that we can use to get data from our camera and it also had a module for edge detection already built. This seemed promising however the data had to go through several layers of conversion prior to doing the edge detection, and then after the edge detection it had to go through more conversions to be able to be displayed on the VGA monitor. The step that we used are shown in figure 10.



Use	Connections	Name	Description	Export	Clock
✓		⊟ **clk_0**	Clock Source		
		clk_in	Clock Input	**clk**	***exported***
		clk_in_reset	Reset Input	**reset**	
		clk	Clock Output	*Double-click to export*	clk_0
		clk_reset	Reset Output	*Double-click to export*	
☐		⊞ **video_test_pattern_0**	Test-Pattern Generator		*unconnected*
✓		⊟ **audio_and_video_co...**	Audio and Video Config		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_av_config_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		external_interface	Conduit	**audio_and_video_config...**	
✓		⊟ **video_decoder_0**	Video-In Decoder		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_decoder_source	Avalon Streaming Source	*Double-click to export*	[clk]
		external_interface	Conduit	**video_decoder_0_exter...**	
✓		⊟ **video_bayer_resam...**	Bayer Pattern Resampler		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_bayer_sink	Avalon Streaming Sink	*Double-click to export*	[clk]
		avalon_bayer_source	Avalon Streaming Source	*Double-click to export*	[clk]
✓		⊟ **video_clipper_0**	Clipper		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_clipper_sink	Avalon Streaming Sink	*Double-click to export*	[clk]
		avalon_clipper_source	Avalon Streaming Source	*Double-click to export*	[clk]
✓		⊟ **video_scaler_1**	Scaler		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_scaler_sink	Avalon Streaming Sink	*Double-click to export*	[clk]
		avalon_scaler_source	Avalon Streaming Source	*Double-click to export*	[clk]
✓		⊟ **video_rgb_resample...**	RGB Resampler		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_rgb_sink	Avalon Streaming Sink	*Double-click to export*	[clk]
		avalon_rgb_source	Avalon Streaming Source	*Double-click to export*	[clk]
✓		⊟ **video_dma_controll...**	DMA Controller		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_dma_sink	Avalon Streaming Sink	*Double-click to export*	[clk]
		avalon_dma_control_...	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
		avalon_dma_master	Avalon Memory Mapped Master	*Double-click to export*	[clk]
✓		⊟ **sram_0**	SRAM/SSRAM Controller		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		external_interface	Conduit	**sram_0_external_interf...**	
		avalon_sram_slave	Avalon Memory Mapped Slave	*Double-click to export*	[clk]
✓		⊟ **video_pixel_buffer_...**	Pixel Buffer DMA Controller		
		clk	Clock Input	*Double-click to export*	**clk_0**
		reset	Reset Input	*Double-click to export*	[clk]
		avalon_pixel_dma_ma...	Avalon Memory Mapped Master	*Double-click to export*	[clk]

Figure 10: Qsys Diagram for Edge detection

If we follow the process laid out in figure 10, we start with the input from the image.

1. Decoder: Qsys provides support for our camera and a De10 FPGA through the Video_decoder module.

2. Bayer resampler: We can then pass that data through an video bayer re-sampler module to convert the pixel values into 8 bit RGB values.

3. Clipper: We then pass those values into a pixel clipper that clips 4 of each side and 3 pixels of the top and the bottom so that the image is a resolution that is a multiple of 640x480.

4. Scaler: We then scale the image down to a resolution of 640x480

5. RGB resample: We then have to do a RGB resample to convert the pixel values from 8 bit RGB to 10 bit RGB values.

6. Grayscale: Although its not shown in figure 10 we need convert the RGB values to a single gray scale value by averaging the three together.

7. Edge detection: We then perform edge detection on the image.

8. First In First Out (FIFO) pipeline: We then need to change the clock domain. All of the previous modules were operating at 50MHz which is the default clock for the default clock for the FPGA, but the VGA operates at 25MHz. So we need to buffer the data so that it can be moved to the slower clock domain.

9. VGA controller: Finally we display the data on the VGA

We began by testing this design on a test image without using input from the camera. This worked as intended. However when we tried to read data in from the camera we found that the Camera Decoder module provided by Qsys in did not support our FPGA. This led us to believe that working with the GHRD provided by the camera manufacturer shown in figure 8 was the best path forward.

Solution 3 was a to implement the Sobel filter in VHDL and Verilog so that we could make use of the camera GHRD which we know is functional. For this implementation we looked at examples from Marco Wizker[4] and Mahboob Karimian[2].

We used VHDL to design the Sobel filter by creating a line buffer in VHDL. A line buffer is a buffer that can hold all of the values for the pixels in a row of the image. This means that our line buffer needs to be 640 pixels long as that is the number of pixels wide the VGA monitor is. We then connect the line buffers so that the output of one of the line buffers is the input of the other. When the pixels begin to fill up the line buffers the first pixel will be moved by one space each clock cycle until it is at the end of the last line buffer. This will mean that we have two rows of the picture stored. We can see a visual representation of this process in figure:

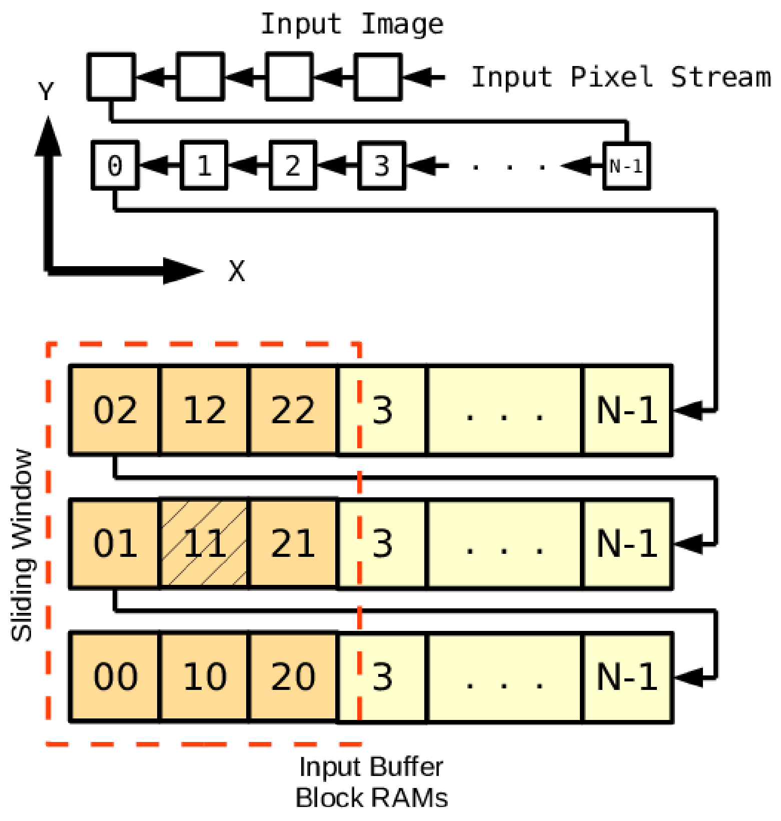


Figure 11: Diagram of using a line buffer to compute the Sobel filter[15]

From the visualization in figure 11 we can see that if two lines are stored we can use the pixels coming in as the third line. This makes the 3d matrix that we can multiple by the values in the Sobel filter.

This is the method we used to compute the Sobel filter. While this a logically sound way to compute the Sobel filter we did run out of time before being able to do extensive testing with real data.

Gradients

The next step in the algorithm is to compute the gradient magnitude and orientation for the pixels. In our original design we planed to compute the gradient for all three color channels and then compare the magnitude to determine which gradient to use. This method, suggested by Hanle et al.[13], gives a better understanding of the sharp contrast in color for our image. We start by calculating the magnitude of the gradient. We use equation 1 to compute the magnitude, but because the square root operator adds a significant amount of time to the calculation of the value we approximate it by removing the square root. Equation 3 shows what the new magnitude equation looks like

$$\|G(x, y)\| = G_x(x, y)^2 + G_y(x, y)^2 \tag{3}$$

Once we have computed this value we can compare it to the other gradient magnitude values.

The next challenge is calculating the orientation of the gradient. To calculate this we normally use equation 2. However, The arctan is very difficult to compute using the FPGA. Because of this we use an alternate equation shown below.

$$\phi(x, y) = \frac{G_y(x, y)}{G_x(x, y)} * 1000 \tag{4}$$

While dividing the two gradients and multiplying by 1000 is not the same as computing the arch we can hopefully approximate the value. If, for example, the $G_y = 1$ and $G_x = 255$ then $\phi(x, y) = 0.00392$ multiplying by 1000 allows us to have whole numbers for every possible combinations. We can use a lookup table to find a approximate angle value that matches $\phi(x, y)$. The block diagram for this calculation is shown in figure 12

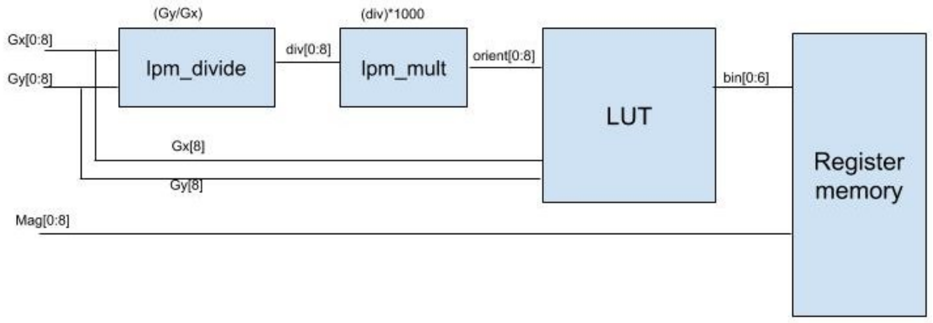


Figure 12: Computing the orientation of the gradient

Hardware Software Interconnect

The interconnect is the only part of the design that connects any custom hardware we built to any code written on the Linux OS. The interconnect is controlled from the software and uses the built in AXI bus previously discussed. The interconnect uses custom hardware which has been designated to specific memory positions within the HPS memory allocations. The interconnect has been designed to take data coming from the custom HOG hardware and is able to send the data to the HPS without having any data over written. This kind of system requires a hardware and software side which will be described independently.

Hardware of the Interconnect

The hardware was based on the Verilog in Appendix A, written by Tony Frangieh, this was the base code which was modified for the final design to allow for the data to be written from external hardware, rather than hardware within the same code block. The FIFO can be observed in the Verilog as the block defined as myfifo. This allows data to come in and exit in the same order so that no matter when data comes to the FIFO none of it is lost. This is a concern because data coming in from the hardware may not always run in a constant time or be completely synchronized with the HPS.

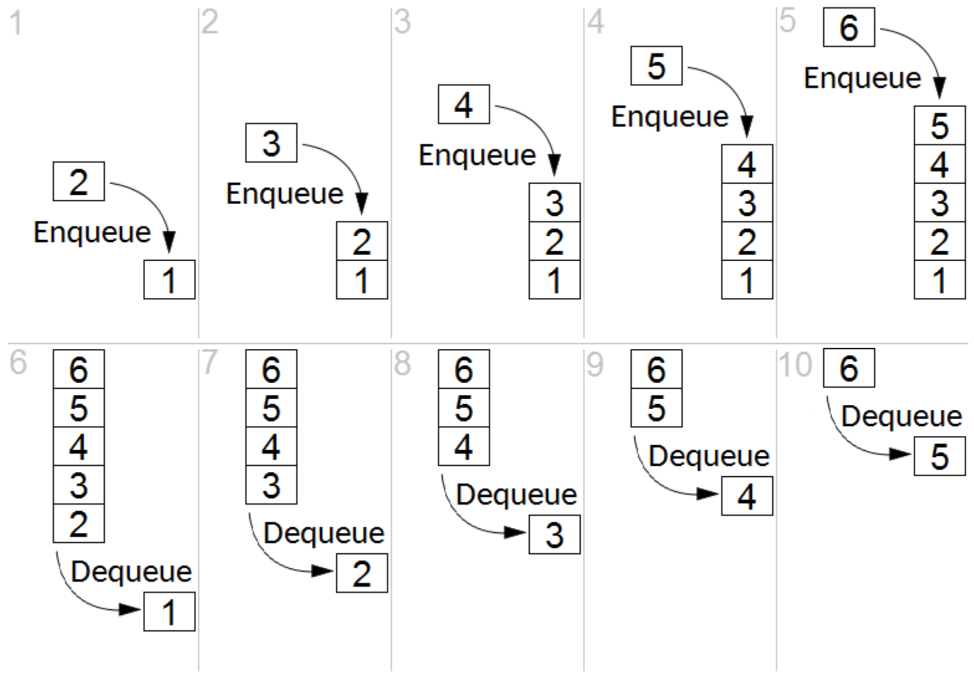


Figure 13: A visual description of how a FIFO memory works [6]

Figure 13 shows the basic idea of how a FIFO memory block works. Using this we were able to read in the data arriving from the camera and HOG with minimal loss, the only loss observed occurred when data was stored in FIFO before the software began running and was able to fill the FIFO before data was able to be dequeued from memory. While dealing with the FIFO in hardware was often more difficult the stabilization it brought was very useful.

Software of the Interconnect

The inter connect is controlled by software through a system called memory mapping. in which you access specific memory pointers in C and can read and write. The memory mapping allows the code to access any custom hardware connected to the AXI bus like this by providing the AXI bus and each piece of hardware connected to it a span of memory in settable in Qsys. The code is then able to control reading and writing data across the bus. This is the main purpose of the code in the interconnect. Not only to pass data into the system but to also control the interconnect for when and where data is being gathered.

This is why software relies on the hardware to deal with synchronization because while it can control when data is written from the HPS to the FPGA it has no real control on data being written within the FPGA. So it needs to have some hardware that can hold onto data so it is not overwritten as it would be in a simpler register structure. Thus, both sides work together to pass the data from the FPGA to the HPS in the design with the HPS controlling when data is read and the FPGA trying to store the data so that the two systems do not need to be perfectly synchronized in order to not lose data.

Performance estimates and results

Due to time limitations we were unable to meet all of our performance goals. We had 3 major goals that we wanted to achieve with our project.

1. Process images in real time

2. Produce accurate data to transfer to the the hard processor

3. The data quality must be maintained in the process of transferring to the hard processor

Real time

The Sobel filter module is able to compute the pixel value in real time as they come in. Except for an initial delay, Once a pixel buffer has been filled, each pixel is computed before the next one comes in. This means that we have met the first goal of the Project.

Data quality

When testing our Sobel filter we did observe an error in the data that was being displayed on the VGA monitor. The image that was output was full of noise and each row did not seem to be aligned.

While we believe that the data is being computed correctly we also believe that the timing is off. When a new image comes in it takes time to fill up the buffers before the first pixel can be computed. This time is not currently accounted for in our design. Unfortunately, due to timing and real world events we were not able to look into fixing this flaw in the design. This means we were unable to achieve this goal.

Data Maintenance

Through our use of the FIFO buffer and the AXI bus that can transfer data at very high speeds we were able to have values created on the FPGA transfer to the HPS without loosing data or seeing repeating data.

While we did meet some of these design goals we were unfortunately unable to put them together to a functional system.

Production schedule

Due to the nature of this project we chose to split the project into two sections. Brock Harris was in charge of hardware implementation and Will Christensen was in charge of the communication between hardware and software.

- Week 1: We started by finalizing our design block diagram overview. We wanted to have a better understanding of how our hardware would connect to the hard processor and what data we would pass between the two. We also began work on a python implementation for testing purposes.

  In order to program the hard processor on bare metal we needed to be able to use the Intel Monitor program.

- Week 2-3: During these weeks we worked on reading related papers to chose a design that would allow us to create the most efficient and simple design. Using these papers we put to get a block diagram that showed the IP components that we planed to use in as much detail as possible. We also sued these block diagrams to inform the design of the python implementation.

  While the Intel Monitor program promising four our desing there were too many problems. Due to these issues we began looking at sample designs for using the Linux distribution to program the hard processor.

- Week 4: Get VHDL design for camera and determine what parts we can use and if it will work within our system. During this time we were also trying to use Qsys.

- Weeks 5-6: Test out 2D FIR package and to make sure that it wouldn't actually work for our design. During this time we also began to learn how Qsys worked and try to figure out what steps needed to be taken to use Qsys to build our design. This required us to look into the documentation for our board and camera to determine which pins we could use and which packages we needed to convert the image in Qsys.

We were able to move data from the FPGA to the HPS using register and work with that data on the HPS. We then need to further understand and expand this transfer so that we are able to move the large amount of data to the hps.

- Weeks 7-9: During these weeks we worked on a VHDL/Qsys implementation for the sobel filter. We used several guides and tested at an implementation by Marco Wizker[4] which we need to edit heavily.

  We enabled the interconnect to be able to hold a arbitray amount of registers. This means that we can transferred much more data at a faster rate the we previously could.

Cost Analysis

There are two major costs associated with this project to make it operational. the first component is camera. For this project we opted to use the 5 Mega Pixel Digital Camera Package. this package includes the TRDB-D5M camera that can be attached to our FPGA. This package costs $80. While this specific camera choice is not necessary it is the cheapest and simplest option.

The next major cost is the FPGA SoC board. For this project we chose to used the DE-10 standard SoC board. This board is necessary because each FPGA has different configurations that mean that our design may not work on other devices. While the cost of this board is $350 Union College and our advisor Cherrice Traver graciously allowed us to use one of their boards temporally so that we could complete this project.

A computer is also necessary to program the DE-10. This is not factored into our cost analysis because most people have computers already. It is important to note, however, the software that we are using to program the DE-10 board, Quartus 2, is a very resource intensive program. It is recommended by the developers to have a computer that has at least 25 GB of available disk capacity on you computer and 6-8 GB of ram[7].

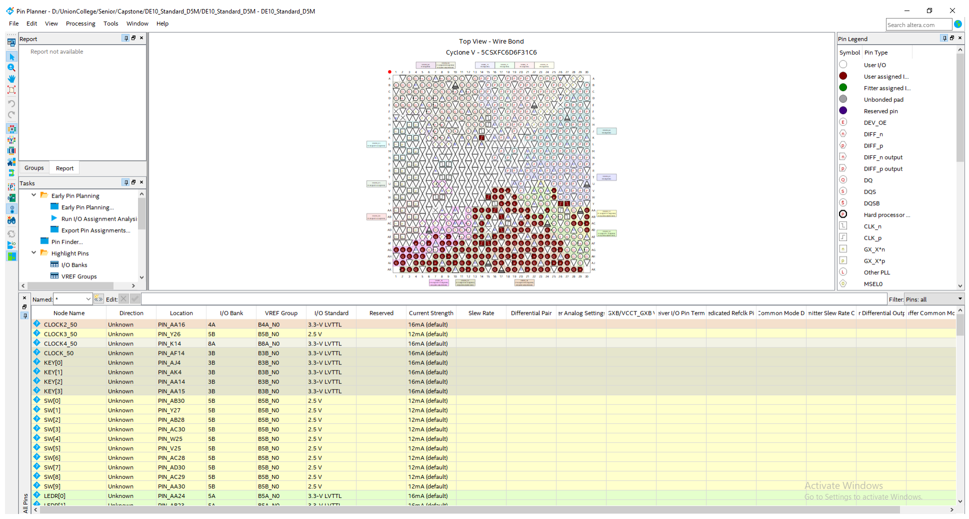
User's Manual

This user manual will describe in detail each tool and step used to run and create the system designed. Understanding the tools used for this project is most likely the most time consuming part of the process, there are a number of parts that any one working with them needs to fully comprehend. The most important knowledge you could have is that of the tools being used, Qsys, Quartus, and Linux. It's important for any one looking to further this project to have a thorough understanding of how the C code interacts with custom hardware in the HW/SW co-design as well.

Quartus Operation

Quartus is the tool provided by Intel for development on Intels series of FPGA. It main purpose is to compile hardware connections and Hardware Description Language (HDL) blocks into a fully functioning system. Quartus also provides a very all encompassing system for editing HDL and editing the connections between these files. And provides simple to use tools for synthesizing these final products into physical hardware that can be uploaded to any FPGA in their library.

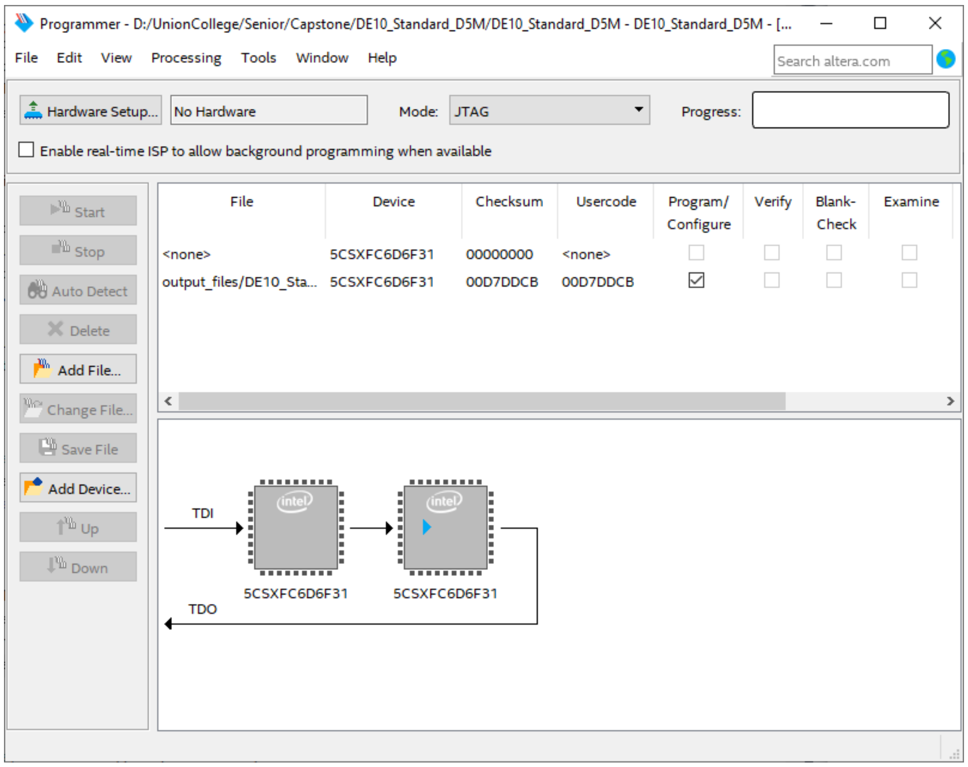
Quartus is incredibly complex, and has a multitude of tools including within it including Qsys which we will cover in the next section. Quartus' tools are the life blood of this project however they are not all used. The only tools used here were the programmer, the pin planning tools, and qsys. These tools are all very important, but the pin planner is probably one of the weirder and less documented tools. For each input that is described the pin needs to be defined and put on the correct settings for the hardware to operate correctly. It can be quite tedious but if you set it up once you shouldn't have to do it again. All the documentation required for a input to any system should be set up in a .qsf (Quartus Settings File) setup if you are using a GHRD and you can use this to help create your own .qsf or set you pins in Quartus. The pin assignments for our project can be seen in figure 14.



Node Name	Direction	Location	I/O Bank	VREF Group	I/O Standard	Reserved	Current Strength	Slew Rate	Differential Pair	er Analog Setting:	GXB/VCCT_GXB \	eiver I/O Pin Term	edicated Refclk Pi	Common Mode D	nitter Slew Rate C	r Differential Outp	iffer Common Mc
CLOCK2_50	Unknown	PIN_AA16	4A	B4A_N0	3.3-V LVTTL		16mA (default)										
CLOCK3_50	Unknown	PIN_Y26	5B	B5B_N0	2.5 V		12mA (default)										
CLOCK4_50	Unknown	PIN_K14	8A	B8A_N0	3.3-V LVTTL		16mA (default)										
CLOCK_50	Unknown	PIN_AF14	3B	B3B_N0	3.3-V LVTTL		16mA (default)										
KEY[0]	Unknown	PIN_AJ4	3B	B3B_N0	3.3-V LVTTL		16mA (default)										
KEY[1]	Unknown	PIN_AK4	3B	B3B_N0	3.3-V LVTTL		16mA (default)										
KEY[2]	Unknown	PIN_AA14	3B	B3B_N0	3.3-V LVTTL		16mA (default)										
KEY[3]	Unknown	PIN_AA15	3B	B3B_N0	3.3-V LVTTL		16mA (default)										
SW[0]	Unknown	PIN_AB30	5B	B5B_N0	2.5 V		12mA (default)										
SW[1]	Unknown	PIN_Y27	5B	B5B_N0	2.5 V		12mA (default)										
SW[2]	Unknown	PIN_AB28	5B	B5B_N0	2.5 V		12mA (default)										
SW[3]	Unknown	PIN_AC30	5B	B5B_N0	2.5 V		12mA (default)										
SW[4]	Unknown	PIN_W25	5B	B5B_N0	2.5 V		12mA (default)										
SW[5]	Unknown	PIN_V25	5B	B5B_N0	2.5 V		12mA (default)										
SW[6]	Unknown	PIN_AC28	5B	B5B_N0	2.5 V		12mA (default)										
SW[7]	Unknown	PIN_AD30	5B	B5B_N0	2.5 V		12mA (default)										
SW[8]	Unknown	PIN_AC29	5B	B5B_N0	2.5 V		12mA (default)										
SW[9]	Unknown	PIN_AA30	5B	B5B_N0	2.5 V		12mA (default)										
LEDR[0]	Unknown	PIN_AA24	5A	B5A_N0	3.3-V LVTTL		16mA (default)										

Figure 14: Pin planner for our design

If someone wants to put our design onto their own board all they would need is the programmer tool and our compiled sof file. The programmer is fairly simple to use and only requires setting up a few things as Quartus automates the system fairly well after your design is fully synthesized. there are two step required, first being hitting auto detect on the system which will set up the HPS and a random FPGA into your system. All that is required to set this up is to remove the random FPGA and replace it with the output synthesis file created by Quartus for the design. The pin planner window can be seen in figure 15



No Hardware

Mode: JTAG

Progress:

☐ Enable real-time ISP to allow background programming when available

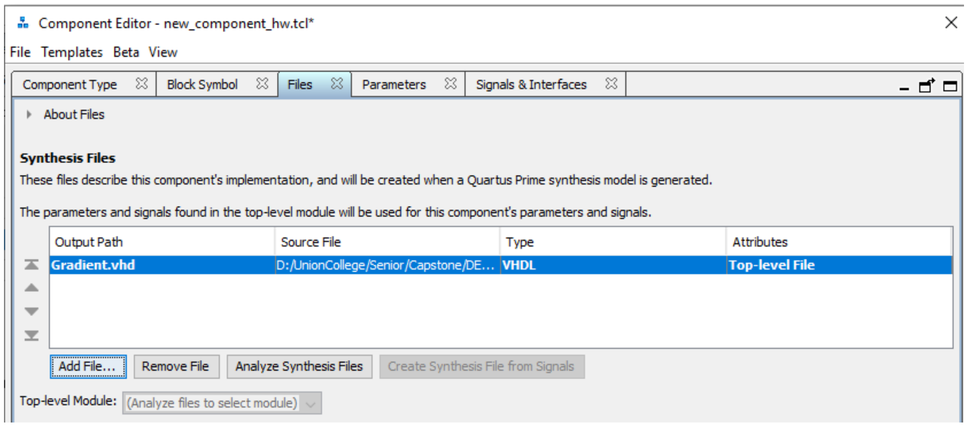
File	Device	Checksum	Usercode	Program/ Configure	Verify	Blank-Check	Examine
<none>	5CSXFC6D6F31	00000000	<none>	☐	☐	☐	☐
output_files/DE10_Sta...	5CSXFC6D6F31	00D7DDCB	00D7DDCB	✓	☐	☐	☐

Figure 15: Quartus Programmer window

Qsys Operation

Qsys is a system provided as a tool by quartus but is in itself very complex and similarly important to the project as the entirety of Quartus. Qsys is a graphical system designed to make connections between intellectual property (IP) modules simpler. But before getting a module into quartus you need to modify your component so that the it fits within the other components in the Qsys design. The inputs and outputs for specific busses need to be similarly named to those it is being connected to. Qsys makes addressing systems much easier because it often doesn't worry about the width of some busses, such as address busses.

Adding new components to Qsys is relitively easy. Qsys comes with a large library of prebuild IP compnents that can be put into the design by clicking on them and then modifying them to fit your needs. You can also create your own components by writing HDL to specify inputs, outputs, and what your componet will do between the two. Once the HDL is made it can be put into the Qsys new component which will analyze the files and create a Qsys component block with your specifications. This is shown in figure 16.



Component Editor - new_component_hw.tcl*

File Templates Beta View

Component Type | Block Symbol | Files | Parameters | Signals & Interfaces

About Files

Synthesis Files

These files describe this component's implementation, and will be created when a Quartus Prime synthesis model is generated.

The parameters and signals found in the top-level module will be used for this component's parameters and signals.

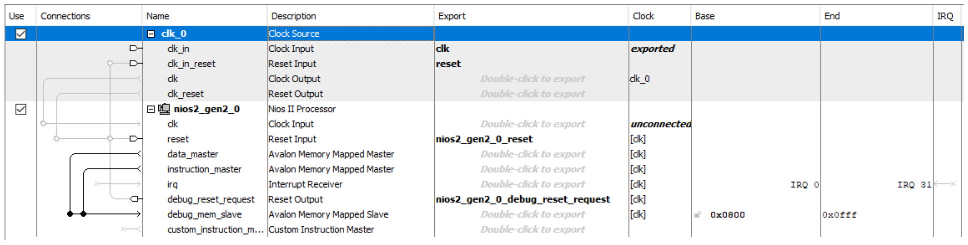
Output Path	Source File	Type	Attributes
Gradient.vhd	D:/UnionCollege/Senior/Capstone/DE...	VHDL	Top-level File

Add File... | Remove File | Analyze Synthesis Files | Create Synthesis File from Signals

Top-level Module: (Analyze files to select module)

Figure 16: How to add HDL files to Qsys components

Another important process that is done in this project is exporting pins to HDL, which allows specific module pins to connect in HDL to systems that are more difficult to fit into a Qsys document, or connect to physical inputs or output in the Quartus pin planner as described previously. This is fairly easy for each pin or bus defined in Qsys simply go to the export tab on that wire and type what you want its export name to be. The export column can be see in figure 17 and only needs to be clicked to be edited.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		clk_0	Clock Source					
		clk_in	Clock Input	clk	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Double-click to export	clk_0			
		clk_reset	Reset Output	Double-click to export				
✓		nios2_gen2_0	Nios II Processor					
		clk	Clock Input	Double-click to export	unconnected			
		reset	Reset Input	nios2_gen2_0_reset	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	nios2_gen2_0_debug_reset_request	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0800	0x0fff	
		custom_instruction_m...	Custom Instruction Master	Double-click to export				

Figure 17: We can see a sample HPS in qsys

Another important part of Qsys is setting up memory positions if a system is addressable from the HPS its address can be set simply by adjusting the values within Qsys. An example of these being set is in figure 17 that there is a column for memory address beginning and ending. This can be changed and allows us to know where this Qsys module will be putting data.

HW/SW codesign using a Linux system

In order to create a program designed to interact with the your Qsys design you will need to first set up an Angstrom Linux system on the DE-10. This is available through the terasic website. In order to boot into the Linux system change the position of the MSEL switches on the board to 101011 or Up, Down, Up, Down, Up, Up from left to right if looking at the board with the user switches at the bottom. After doing this the Linux system will boot when the SoC is turned on. Accessing the terminal within the Linux system is the next step, to gain remote access to the OS use a PuTTy terminal or similar through a JTAG connection. From here you will need to connect the SoC to an Ethernet connection, from your computer you will upload the code you have written through an SSH command. After this all you will need to do is run any C code you have written directly on the SoC through the PuTTy terminal.

Coding for Memory Mapping

The code for the interconnect accesses data from the hardware across the AXI bus using a system called memory mapping. This allows C to access the data just as you would a memory pointer. Memory mapping in C can be done with a couple lines but the main command is:

```
mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, HW_REGS_BASE);
```

mmap is the memory mapping system within C and all of the capitals are basic constants that will be discussed later but fd is a variable that represents the open file */dev/mem* which is the file refrence for all memory within the system. The **HW_REGS_BASE** represents the default LW AXI bus base at 0xff200000. This detail is documented within the DE10 standard manual.

After creation of the mmap variable you can use it to map to the exact address of the by using

```
hwsw_addr = virtual_base + ((unsigned long)(EXAMPLE_IP_BASE) & (unsigned long)(HW_REGS_MASK));
```

this creates a pointer to the exact position in memory of the AXI bus module using the **EXAMPLE_IP_BASE** as the address for the module set by Qsys. For full code of an example IP see the appendix.

Discussion, Conclusions, and Recommendations

The original goal of our project was to create a system that would be able to detect object, such as stop signs, in images in real time. To do this we chose to implement and Histogram of Oriented Gradient Algorithm on an Field Programmable Gate Array. We wanted to pass the data computed for the HOG to a Support Vector Machine. To improve the speed of the SVM we chose to implement that section on a Hard Processor System.

Do to time limitation we chose to implement only the first half of the HOG algorithm and then work on transferring that data to the HPS so that it could be computed later. We tried many different ways to implement the sobel filter starting the the 2D FIR library, then trying to implement it in Qsys, and final settling on a VHDL implementation. We also tried many ways to program the hard processor. We originally attempted to program it on bare metal using the intel Monitor system, we then tried using Quartus, and finally we settled on using the Linux OS as a base that we could then upload binary files to.

Future Work

We have quite a bit of future work to do. We first need to fix issues with FIFO memory issues, and problems with synchronization of the Hardware and Software starting at the same time. This would be a large period of work as simply getting the data interconnect working took a very long time. We also need to implement all the custom Gradient calculation blocks that haven't yet been connected together. However connecting all of this to the interconnect will also require finishing the creation of the SoC ready SVM and then using the data we have to train the system. Which would probably take very long period time to get everything right.

Lessons Learned

From the very beginning we knew that this project was going to be too big to accomplish in the two terms that we had. We decided that working on a project that was on a large scale but also was interesting and relevant to current issues was worth the risk of not making a lot of progress into the project. While, to some extent we still believe this is valid, we also would not choose to do this again.

The major lesson we learned is that it is important to set more realistically obtainable goals. We are happy with the progress that we made and the amazing amount of things we learned, but also we do know that it does not look like we made a lot of progress due to the scale and complexity of our project. We know that it reflects poorly on us that we were unable to complete a working prototype. Next time we would choose a project that is more realistically solvable with in the time constraints.

References

[1] Advanced driver assistance systems.

[2] Hello tech! mahboob karimian personal webpage, Aug 2000.

[3] *Universal Serial Bus Device Class Definition for Video Devices: Video Camera Example.* August 2012.

[4] *Image Processing with Terasic FPGA-Boards.* Jan 2019.

[5] Real-world benefits ofcrash avoidance technologies. *nsurance Institute for Highway Safety, Highway Loss Data Institute*, Jun 2019.

[6] Fifo (computing and electronics), Jan 2020.

[7] System and software requirements, Mar 2020.

[8] Siddharth Advani, Yasuki Tanabe, Kevin Irick, Jack Sampson, and Vijaykrishnan Narayanan. A scalable architecture for multi-class visual object detection. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE.

[9] Andrew.currin.ctr@dot.gov. Driver assistance technologies, Jun 2019.

[10] Ashish. Understanding edge detection (sobel operator), Sep 2018.

[11] Guyon Issabelle M. Boser, Berhardt E. and Vladamir N. Vapnik. A training algorithm for optimal margin classifiers. page 144, 1992.

[12] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005.

[13] Michael Hahnle, Frerk Saxen, Matthias Hisung, Ulrich Brunsmann, and Konrad Doll. FPGA-based real-time pedestrian detection on high-resolution images. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 629–635. IEEE.

[14] Matthew.lynberg.ctr@dot.gov. Automated vehicles for safety, Mar 2020.

[15] Craig Moore, Harald Devos, and Dirk Stroobandt. Optimizing the fpga memory design for a sobel edge detector. pages 299–300, 01 2009.

[16] Vinh Ngo, Arnau Casadevall, Marc Codina, David Castells-Rufas, and Jordi Carrabina. A high-performance HOG extractor on FPGA. *CoRR*, abs/1802.02187, 2018.

[17] Dat Nguyen, Hyung Hong, Ki Kim, and Kang Park. Person recognition system based on a combination of body images from visible light and thermal cameras. *Sensors*, 17:605, 03 2017.

[18] Seymour Papert. The Summer Vision Project. In *AI Memos 1959-2004*. MIT.

[19] Benjamin Preston. The hidden cost of car safety features.

[20] Jack Stilgoe. Machine learning, social learning and the governance of self-driving cars. *Social Studies of Science*, 48(1):25–56, 2018. PMID: 29160165.

Appendices

Appendix A - Example of HW/SW Co-design Verilog Hardware Description

Author: Tony Frangieh

```
module example_ip (
  // signals to connect to an Avalon clock source interface
  clk,
  reset,
  // signals to connect to an Avalon-MM slave interface
  slave_address,
  slave_read,
  slave_write,
  slave_readdata,
  slave_writedata,
  led
);

  // clock interface
  input clk;
  input reset;
  // slave interface
  input      [1:0]  slave_address;
  input             slave_read;
  input             slave_write;
  output reg [31:0] slave_readdata;
  input      [31:0] slave_writedata;
  output     [7:0]  led;

  // local signals
  reg [31:0] user1; // Used, MSB 8 bits connect to the LEDs on FPGA
  reg [31:0] user2;
  reg [25:0] local_counter;
reg [31:0] fifo_input;
reg        write_fifo;

myfifo myfifo_inst(
  .clock (clk),
  .data  (fifo_input),
  .rdreq (slave_read),
  .wrreq (write_fifo),
  .empty (),
  .full  (),
  .q     (slave_readdata));

// Local counter generator
always @ (posedge clk or posedge reset)
begin
  if (reset == 1)
  begin
    local_counter <= 0;
  end
  else begin
    local_counter <= local_counter + 1;
  end
end

// Fifo input generator
always @ (posedge clk or posedge reset)
begin
  if (reset == 1)
  begin
    fifo_input <= 0;
    write_fifo <= 0;
  end
  else begin
```

```
    if (local_counter == 0)
    begin
      fifo_input <= fifo_input + 1;
      write_fifo <= 1;
    end
    else begin
      write_fifo <= 0;
    end
  end
end

// user1 register drives LEDs 7:0 on FPGA
assign led[7:0] = user1[7:0];

// Avalon write state machine
always @ (posedge clk or posedge reset)
begin
  if (reset == 1)
  begin
    user1 <= 0;
    user2 <= 0;
  end
  else begin
    if (slave_write == 1) begin
      case (slave_address)
        1'b0: user1 <= slave_writedata;
        1'b1: user2 <= slave_writedata;
        default: begin
          user1 <= 0;
          user2 <= 0;
        end
      endcase
    end
    end
  end

endmodule
```

Appendix B - Example HW/SW Co-deisgn Software

Author: Tony Frangieh

```
#include <stdint.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>

#define HW_REGS_BASE    (0xff200000)
#define HW_REGS_SPAN    (0x00200000)
#define HW_REGS_MASK    (HW_REGS_SPAN - 1 )
#define EXAMPLE_IP_BASE (0x00006000)

int main() {

  void *virtual_base;
  int  fd;
  void *h2p_lw_led_addr;

  // map the address space for the LED registers into user space so we can interact with them.
  // we'll actually map in the entire CSR span of the HPS since we want to access various registers wit
  if((fd = open("/dev/mem", (O_RDWR | O_SYNC))) == -1) {
    printf("ERROR: could not open \"/dev/mem\"...\n" );
    return(1);
  }
  virtual_base = mmap(NULL, HW_REGS_SPAN, (PROT_READ | PROT_WRITE), MAP_SHARED, fd, HW_REGS_BASE);

  if(virtual_base == MAP_FAILED) {
    printf("ERROR: mmap() failed...\n");
    close(fd);
    return(1);
  }

  h2p_lw_led_addr = virtual_base + ((unsigned long)(EXAMPLE_IP_BASE) & (unsigned long)(HW_REGS_MASK));

  while(1) {
    // turn all leds off (LED2 through LED9)
    *(uint32_t *)(h2p_lw_led_addr) = 0;
    usleep(500000);
    // turn all leds on (LED2 through LED9)
    *(uint32_t *)(h2p_lw_led_addr) = 255;
    usleep(500000);
  }

  // clean up our memory mapping and exit
  if(munmap(virtual_base, HW_REGS_SPAN ) != 0) {
    printf("ERROR: munmap() failed...\n");
    close(fd);
    return(1);
  }

  close(fd);

  return(0);
}
```