Feature Matcher Accelerator
18-545 Advanced Digital Design
Fall 2015

Brandon Perez
bmperez@andrew.cmu.edu

Jared Choi
jaewonch@andrew.cmu.edu

December 15, 2015

Contents

1 Project Overview **4**
- 1.1 Project Description . . . 4
- 1.2 Project Motivation . . . 4
- 1.3 Goals and Notable Accomplishments . . . 5
- 1.4 Tools and Components . . . 6
  - 1.4.1 Avent Zedboard and Xilinx Zynq 7000 Series SoC . . . 6
  - 1.4.2 ARM Cortex-A9 and NEON SIMD Engine . . . 6
  - 1.4.3 Linux Kernel (Xillinux) . . . 6
  - 1.4.4 Kinect V1 and Libfreenect . . . 6
  - 1.4.5 Xilinx Tools . . . 7

2 Project Design **8**
- 2.1 System Level Block Diagram . . . 8
- 2.2 System Overview . . . 8
- 2.3 System Flow . . . 9
- 2.4 Hardware Subsystems . . . 11
  - 2.4.1 Generic 2D Convolution . . . 11
  - 2.4.2 Bayer to Grayscale Conversion . . . 11
  - 2.4.3 Gaussian Filter . . . 12
  - 2.4.4 Sobel Filter . . . 13
  - 2.4.5 Non-Maximum Suppression . . . 14
  - 2.4.6 CORDIC . . . 15
  - 2.4.7 Orientation to Bin . . . 15
  - 2.4.8 AXI/AXIS Protocol . . . 15
  - 2.4.9 Processor to FPGA Interface . . . 17
- 2.5 Software Subsystems . . . 19
  - 2.5.1 Software Stack Diagram . . . 19
  - 2.5.2 Linux Kernel (Xillinux) . . . 19
  - 2.5.3 Xilinx AXI DMA Driver . . . 20
  - 2.5.4 AXI DMA Linux Driver . . . 20
  - 2.5.5 Libaxidma . . . 22
  - 2.5.6 Libfreenect . . . 23
  - 2.5.7 Libusb and Usbcore Linux Driver . . . 23
  - 2.5.8 Feature Descriptors . . . 23
  - 2.5.9 Feature Matching . . . 24
  - 2.5.10 Userspace Application . . . 24

3 Results **25**
- 3.1 Demo . . . 25
- 3.2 System Throughput and Performance . . . 26
- 3.3 Resource Utilization . . . 27
- 3.4 Software Comparison . . . 28

4 Personal Statements **30**
- 4.1 Brandon Perez . . . 30
- 4.2 Jared Choi . . . 32

5 Statement of Permission **34**

List of Figures

1 System Level Overview 8
2 Example Image Kernel Calculation 11
3 Bayer Pattern for the Kinect 12
4 Gaussian Filter Kernel 13
5 Sobel Filter Kernels 13
6 Sobel Filter Kernels 14
7 AXI4-Stream protocol. 16
8 The Zynq-7000 series processing system. 17
9 Software Subsystem Overview 19
10 Demo Screenshots 25
11 System Resource Utilization 27
12 Performance Comparison Bar Graph 29

List of Tables

1 System Performance 26

1 Project Overview

1.1 Project Description

Our project was to build a hardware accelerator system for feature matching. Feature matching is a step in the larger algorithm of visual odometry. Visual odometry is an algorithm used to estimate the motion of a robot based data from camera sensors. The basic idea is to look at two consecutive frames and figure out which features in each frame match. Once matches have been found, the relative motion between the two frames can be estimated by aggregating the motion of all features that were matched. The purpose of feature matching is to identify all of these matches between the two consecutive frames. We will discuss these algorithms in more detail in later sections.

The overarching goal for our project was to be able to perform the calculations in real time, streaming data directly from a camera to be processed. We sought to accelerate each step of the algorithm with the appropriate piece of hardware, which in most cases was an FPGA, but in other cases consisted of using a CPU or a SIMD engine instead. The feature matching algorithm lends itself nicely to a pipelined implementation, so we sought to accelerate each stage of algorithm, one at a time, accelerating as much as possible.

We used the Xilinx Zedboard as the basis of our system. The Zedboard is an FPGA system on chip (SoC) that consists of programmable logic (PL) FPGA portion and a ARM processing system (PS) with a NEON SIMD engine. To get the data from the camera, we used Xilinx's AXI DMA IP block to stream the data from DRAM out to the PL. Our hardware on the PL performs the heavy lifting on preprocessing the image and extracting the features, and sends this data back to the PS with DMA. With this information, we then send the data to the NEON SIMD engine to perform feature matching.

For our project demo, we used an Xbox Kinect as the camera that fed input data to our system. After we finished feature matching, we also overlaid the feature points (corners) and the feature matches, and streamed this out to a VGA display. The image displayed had all of the corners colored green, and also colored motion in the frame with one of four colors, all in real-time. In a real application, we would not perform this display, but instead send the feature points and feature matches to the next part of the system.

1.2 Project Motivation

When we started coming up with ideas for our project, we knew one thing for certain: we were not going to be doing a game. While the building a game can be fun, interesting, and very educational, it was simply something that did not interest us a lot. Instead, we wanted a project that would showcase the power of an FPGA, and what it is truly capable of. Thus, we steered toward a project that would be application and research oriented. Jared worked on with a robotics research team over the summer, where he first got the idea of accelerating this algorithm with an FPGA. We eventually decided on using the FPGA to accelerate feature matching.

Feature matching, and the larger visual odometry algorithm that it fits within, are used in a wide variety of applications. The main application is in robotics, where visual odometry is used to track the motion the motion of a robot over time. This naturally is used to control movement, and assist in navigating the environment. For example, visual odometry is used by the Mars Rover to help it navigate the Martian terrain. The newest application for visual odometry is autonomous

vehicles, where this will naturally be used to assist the system in driving the car, and making decisions about navigation.

The reason the FPGA is well-suited for this application is twofold. First, many of the algorithms used in feature matching are "embarassingly" parallel, in that it is very straightforward to parallelize them. While this means it is very easy to implement these algorithms on the FPGA, this alone does not justify the use of an FPGA. After all, a GPU could perform better than the FPGA if this was the sole reason. Where the FPGA really excels is in its low latency and high responsiveness. Unlike a GPU, an FPGA is very efficient at streaming data, and exploiting pipelined parallelism. Thus, the FPGA is well-suited for feature matching because we want the feature matching to happen in real time, which requires the image to be streamed.

1.3 Goals and Notable Accomplishments

Our goal for this project was to have real time feature matching on an FPGA. We accomplished this goal in its entirety, getting our system to be able to stream data from the Kinect at 30 frames per second with a 640x480 resolution image, and even overlay the data on the image in real time. Without the overhead of the display, our system could handle 110 frames per second with a 640x480 image. Along the way, we also developed a driver and software library to interface with AXI DMA, and got a full Linux and FPGA system up and running, even utilizing the NEON SIMD engine on the ARM core.

In many ways, our project was the first (or close to first) in many areas for 545 projects:

1. We were the second team to use a Linux-based system for interacting with the FPGA. The other group that previously did this was the Digital Logic Analyzer group (from 2014).

2. We were the first team to get Xilinx's AXI DMA intellectual property (IP) block working and interfaced with our hardware. The Digital Logic Analyzer group (from 2014), did use the AXI crossbar IP, but this is different.

3. We were the first team to get real-time streaming working at a significantly high frame rate (30 frames per second). The Real Time Ray Tracer group (from 2012) and the Haxorus group (from 2011) both did real-time streaming applications, but neither group mentioned the frame rate achieved in their reports.

4. We were the first team to use Xilinx's high-level synthesis (HLS) tool to help synthesize the FPGA hardware.

The reason we list out these accomplishments is not to brag about our project. Rather, this is for future teams reading this report. If you're planning on doing anything related to any of these items, we strongly recommend that you read through this entire report in detail. Many of these things we had to learn entirely on our own, and there is a pretty high learning curve for these tools. We hope that future projects benefit from what we learned, and will not need to focus as much on learning the tools, but instead be able to focus more on the design and implementation of their project.

1.4 Tools and Components

1.4.1 Avent Zedboard and Xilinx Zynq 7000 Series SoC

The Zedboard is a development board manufuactured by Avent that contains a Zynq-7000 SoC, with several peripherals added on by Avnet. The Zynq-7000 SoC is a system on chip FPGA that comes with a tightly integrated ARM Cortex-A9, with communication via a high speed interconnect, and a NEON SIMD engine. The Zedboard specifically uses the Z-7020 SoC, which has an FPGA that comes with 85 K logic elements, 4.9 Mb of block RAM, and 220 DSP slices[1]. The processing system on the Zedboard is dual-core, clocked at 667 MHz, and comes with 512 MB of DDR3 memory[2].

For our project, we decided on using Avnet's Zedboard for two main reasons. First, the development board comes with a Zynq-7000 series processing system, which has an on-chip ARM Cortex-A9 dual core processor.We knew from the start that we would be using Linux to interface with the FPGA, so this was a natural choice. Second, of all the boards offered, the Zedboard is the best documented. However, most of this documentation is not actually created by Xilinx, but instead Avnet, which is the company that manufactures the board. Most importantly, there are a large number of tutorials online[3]. The documentation is still not good, but it far better than other development kits offered by Xilinx.

1.4.2 ARM Cortex-A9 and NEON SIMD Engine

The Zedboard comes with a fully integrated ARM Cortex-A9 dual core processor. The Cortex-A9 is an out-of-order, two-way superscalar, pipelined processor that fully conforms to the ARMv7-A architecture specification[4].

The Zedboard's ARM Cortex-A9 also comes with a NEON SIMD engine. NEON is a 128-bit, single instruction, multiple data (SIMD) coprocessor extension to the Cortex-A9 processor[5]. The NEON engine has its own independent pipeline and register file, with 32 registers that are 64-bits wide.

1.4.3 Linux Kernel (Xillinux)

Xillinux is a branch off of the mainline Xilinx Linux kernel, which essentially boils down to several patch sets applied over the mainline branch. Both of these kernels are forks of the mainline 3.12 Linux kernel, so we used the 3.12 Linux kernel for our project.

1.4.4 Kinect V1 and Libfreenect

The Kinect V1 is the first iteration of Micrsoft's Kinect sensor. The sensor comes with a camera that can stream out a 640x480 resolution image at 30 frames per second with the raw Bayer pattern. Also, the kinect can stream out depth data through the on-board infrared sensor.

[1] http://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

[2] http://zedboard.org/content/zedboard-0

[3] http://zedboard.org/support/design/1521/11

[4] http://www.arm.com/cortex-a9.php

[5] http://www.arm.com/products/processors/technologies/neon.php

The libfreenect library is an cross-platform open source library for interfacing with the Kinect V1. It is maintained by the Open Kinect group, and captures almost all of the functionality one would get from the Kinect Windows SDK software.

1.4.5 Xilinx Tools

For the entirety of this project, we used the 2015.2 version of the Xilinx tools. There were times when some online examples were only compatible with earlier versions, but we stuck with the latest version because it has the latest versions of the IP blocks and more features. There are three main Xilinx tools: Vivado, Xilinx SDK, and Vivado HLS.

Vivado is the synthesis tools for Xilinx; it is responsible for taking your design and synthesizing it into a bitstream that can then be programmed onto the FPGA. The Xilinx SDK is for embedded and application code development. It comes with an Eclipse plugin, and it is used to interact with the processing system on the board. Also, it comes with a built-in code debugger, so you can use JTAG to debug the code running on the processor. The third tool is Vivado HLS. The high-level synthesis (HLS) tool allows one to write C or C++ code, have the HLS compiler infer the behavior into Verilog code, and then compile that to a hardware implementation. The HLS tool is very powerful, and allows one to specify standard interfaces for the inputs, and allows simple interaction with on-chip hardware, such as Block RAM's or the DSP's. We used the HLS tool for the development of all of our FPGA hardware.

2 Project Design

2.1 System Level Block Diagram

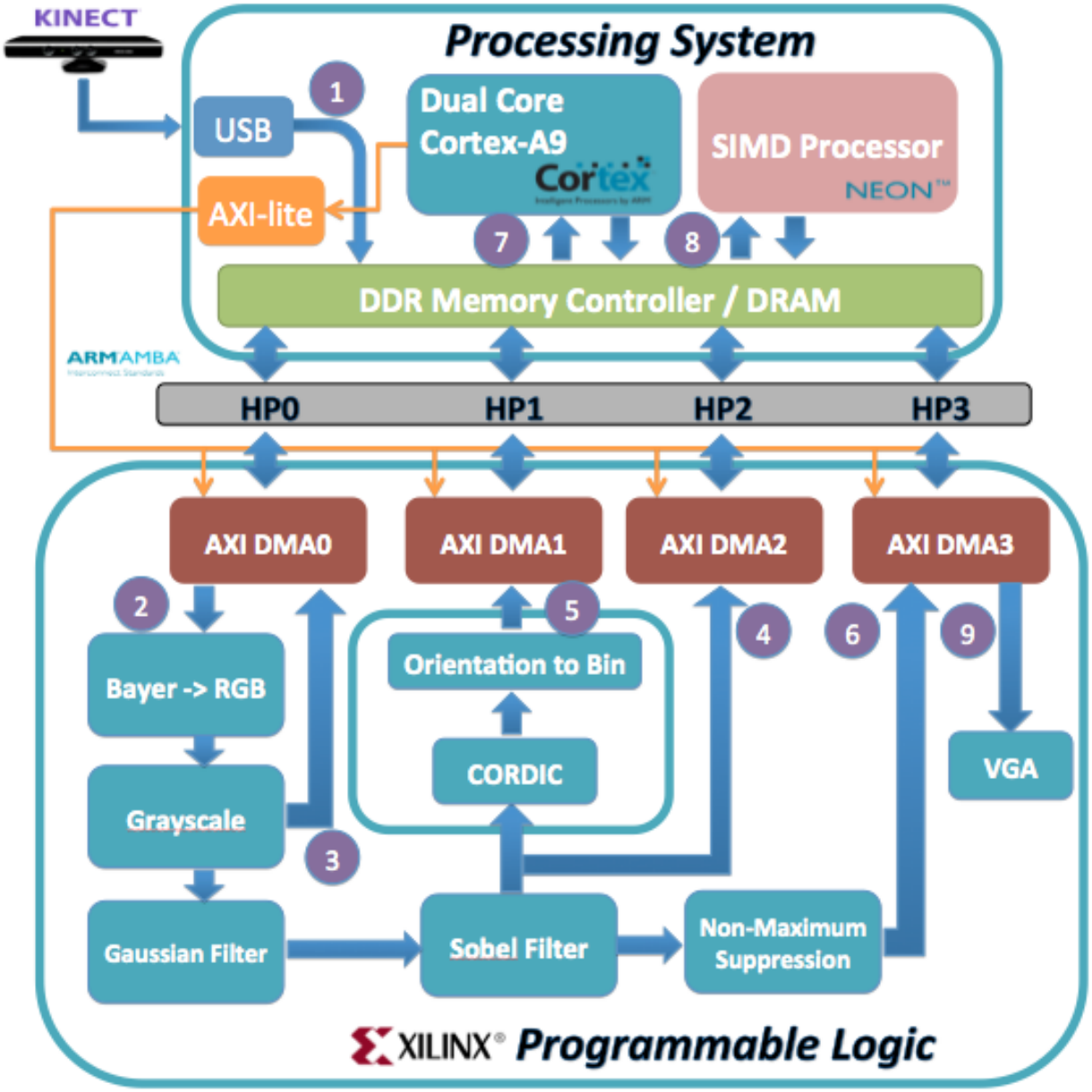


Figure 1: System Level Overview

2.2 System Overview

Above, the block diagram for our system is pictured. We are going to begin with breakdown of the system by going through how an image is processed, then we will dive into each subsystem in greater detail. As you can see from the diagram, there are quite a few different components involved in this system.

We broke our system down into essentially three separate parts. The first partition was the hardware and software, and the software partition was additionally broken down into kernel-level software, and user-level software. This essentially gave us three different levels of abstraction: low-level hardware, low-level software in the form of a kernel driver, and higher-level software in the form of userspace applications.

We carefully designed and built the interfaces between each to ensure that they were as decoupled as was reasonably possible. This ensured that we could independently change each component without greatly affecting the other. The interface between the hardware and kernel driver was the AXI DMA engine and the AXI protocol. The kernel driver acted as the interface between the hardware and userspace applications, and a custom software library acted as the interface between the userspace application and the kernel driver.

2.3 System Flow

At the beginning, we startup our userspace application, which begins the initialization of the system. First, the application requests several DMA-appropriate buffers from the AXI DMA driver; these buffers are dynamically allocated by the driver. The application then sends the buffer's address to the Kinect through Libfreenect, who then begins streaming image data from the Kinect into the buffer. Libfreenect will then asynchronously notify the application when the frame transfer completes.

Once the frame transfer is complete, the application will then initiate the DMA transfer to the PL fabric. The application will send the buffer's address to the driver, who will send it out to the AXI DMA IP block through AXI-Lite, which is used for writing to the module's register map. At this point, the DMA transfer begins, and the AXI DMA engine reads out one pixel per cycle from the DRAM buffer.

The data sent from the Kinect comes as raw image data, specifically the Bayer pattern that comes directly from the CMOS sensors on the camera. First, this is converted to RGB, and we then convert this RGB image into grayscale. This grayscale image is sent back to a separate buffer in DRAM. With the conversion complete, we then preprocess the image. We apply a Gaussian filter to the image, which smooths the image out, and removes any high frequency noise in the image.

Next, we then begin feature extraction on the image. The Sobel filter extracts the differential gradient in the x and y directions from each pixel in the image. These gradients allow us to determine which pixels in the image are corners. The score for each pixel is computed using the Harris response, and we apply a threshold to filter out only the relevant corners.

To get the final list of corner feature points in the image, we apply non maximum suppression, which serves to find us the best set of corners. In a local neighborhood of pixels in the image, we'll typically find several corners if at least one is found. Non maximum suppression serves to find the local maxima of these pixels, giving a single corner out from the region. We then stream this list of maximum corner back to another buffer in DRAM, and a list of points in the image, represented as row and column.

In parallel with the non-maximum suppression, we additionally compute metadata about each pixel to assist with feature matching. For each pixel, we send the magnitude of its gradient back to

another buffer in DRAM (this is computed by the Sobel filter). At the same time, we also compute the dominant orientation for each pixel. The orientation is essentially the direction of the corner. We use CORDIC, a fast algorithm for approximating trigonometric and hyperbolic functions, to compute arctangent. We then sort the pixel into one of several predefined bins based on its orientation, and send these bin numbers back to DRAM into a separate buffer.

Then, with all of the necessary data, feature points, orientation bin number, and the gradient magnitude, sitting in buffers in main memory, we can perform feature matching. First, we compute the SIFT feature descriptors using the ARM core for each corner feature point. The SIFT descriptors give the scale at which the gradient is maximized, seeking to make the feature point scale invariant.

Finally, with the feature descriptors for each corner point computed, we can now perform feature matching to correlate the current feature points and the previous feature points. The feature matching computation is offloaded onto the NEON SIMD engine. To compute matches, we compare all of the feature descriptors using a sum of absolute differences (SAD) to find the ones which are most similar. The NEON engine is able to effectively accelerate the SAD computation, as it can perform all of the computations in parallel. Using the SAD values, the processor then compares these values to figure out which features match.

At this point, we would send this data to whatever the next stage of the computation is, but in our demo setup, we now want to overlay this information onto the grayscale image. Thus, we overlay all of the current feature points onto the image, and use a simple rasterization algorithm to draw the lines between matched features and their previous location. We then stream out this frame to be displayed over VGA. Our display pipeline has a simple FIFO that synchronizes the DMA stream with the VGA display.

2.4 Hardware Subsystems

2.4.1 Generic 2D Convolution

The basis for almost all of our image processing algorithms is 2D convolution, or as it is referred to in the domain of image processing, image kernels. An image kernel is simply a (typically square) matrix of weights that one uses to compute a value for a given pixel. For a given pixel, you look at the surrounding pixels, apply the given weights to them, and then sum then all together to get the result of the kernel. For example:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = (0\cdot 1)+(-1\cdot 2)+(0\cdot 3)+(-1\cdot 4)+(5\cdot 5)+(0\cdot 7)+(-1\cdot 8)+(0\cdot 9) = 5$$

Figure 2: Example Image Kernel Calculation

Above, the first matrix is the image kernel, which is a 3x3 kernel. This kernel is the sharpen filter, which is used to enhance the textures in the image. The second matrix is example values for the pixels in the image; the value we're calculating for is the center pixel. The size of kernel can be anything, but it is typically a square matrix so that it is symmetrical. In our system, we use 3x3 kernels for all of our calculations.

When the entire image is sitting in a buffer, doing this calculation is pretty simple. However, when you're streaming data in, additional complications arise. In order to calculate the value for any given pixel, we need all the pixels values for a 3x3 neighborhood of the pixel. This means that we need the pixel values of the previous and next rows in order to perform the calculation. As we're streaming the image in row-major order, this means that we need a buffer to hold three rows in an image. This must also be a circular buffer, as we must cycle rows in and out of the buffer as we proceed. Before any calculations begin, we must first stream in two whole rows, followed by a 3 pixels from the third row. At this point, we reach the steady state of our calculation, and we stream in one pixel and output one value per cycle. When we reach the end of a row, we must again wait for 3 pixels to stream into the next row, then we can start the calculations again.

The other complication arises with the edges of the image. Along the borders of the image, the pixels are missing certain neighbors, so the image kernel cannot be applied to them as is. There are a few options. We can crop out the image, discarding those pixels, we can use default values for the mixing pixels, or we can mirror the previous or next row. We avoid cropping the image, because this leads to issues with displaying the image, as its size is different than the one we input. Thus, we either use default values, or mirror one of the rows to get the proper value.

2.4.2 Bayer to Grayscale Conversion

When the Kinect streams the image data out, it does not do so in 32-bit RGB format. Instead, the Kinect streams out what is known as a Bayer pattern. The Bayer pattern is the raw sensor data from the camera, the values given by CMOS sensors on the camera. To convert the Bayer pattern into a proper RGB image, you must demosaic the image. In the Bayer pattern, each pixel is represented by a one-byte value. However, these one byte values only represent one color channel for the pixel. The pixels are laid out in a Bayer pattern, where different values in the series represent

colors. For example, the first pixel might give you blue, then the next might be green, and the next is red, and so and so forth. On the Kinect, the pattern is as follows:

G	R	G	R	G	R	G	R	···
B	G	B	G	B	G	B	G	···
G	R	G	R	G	R	G	R	···
B	G	B	G	B	G	B	G	···
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 3: Bayer Pattern for the Kinect

To convert the image to RGB, we must interpolate the missing color channels for each pixel in the image, which involves looking at the colors of the neighboring pixels. These neighboring values are used for the interpolation. This process is referred to demosaicing the image. There are four distinct patterns we can encounter, and four different interpolation equations we must apply to extract the color channels for the pixel.

Our hardware implementation is based off the interpolation that Libfreenect uses to give the user RGB images from the Kinect data[6]. That is also where we got all of the interpolation cases and equations. The implementation is more or less as described, we simply keep row and column counters around to track our position in the image, and determine which interpolation to perform. As described in the generic 2D convolution, we must keep a circular buffer of three rows as we stream data in. For the edges of the image, we mirror the pixel values to fill in the missing pixels. So, for example, on the first row of the image, we mirror the second row's pixel values when performing the interpolation. Additional complications arise from streaming the data from the AXI DMA module. The minimum stream width is 4-bytes, so our module has to process 4 pixel values at a time. However, we only want to stream out one grayscale pixel at a time, so our module has to internally use a FIFO so that it can process one Bayer value per cycle.

Once we have the RGB values for pixel, the conversion to grayscale is trivial. The grayscale conversion we use is simply the average of the three color channels, so immediately after interpolating the RGB values, we sum them together and divide by three to get the average intensity.

The module streams in 4 one byte Bayer pixels per transaction from the AXI DMA module. It outputs the grayscale values into two streams. The first is the gray value in RGB format, so the value is replicated in all color channels. This is streamed back to the processor through AXI DMA, to be later used for feature matching and overlaying the feature data. The second is streamed is simply the one byte grayscale intensity value, which streamed to the Gaussian filter.

2.4.3 Gaussian Filter

Once we the image has been converted to its grayscale form, we can start processing the image. The Gaussian filter is an image kernel that is applied to images to "smooth" them out. The Gaussian filter normalizes the intensity of pixels within their local region. It acts as a low-pass filter, removing any high-frequency noise from the image. This is important because it removes artifacts from image,

[6] https://github.com/OpenKinect/libfreenect/blob/master/src/cameras.c#L453

particularly relevant for our use case because the Kinect camera is both low-resolution, and is not a very precise camera. For our system, we use a 3x3 Gaussian filter, which corresponds to the following kernel:

$$\frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 4: Gaussian Filter Kernel

This module uses the generic 2D convolution model for implementing the image kernel calculations, using a 3 row circular buffer to hold the input pixels. To deal with the edges of the image, we mirror the corresponding rows and/or columns to fill in the missing pixels. This module streams in 8-bit grayscale pixel values. This module outputs an 8-bit Gaussian value, which is the normalized grayscale intensity of the pixel, one for each pixel in the input image, to the Sobel fitler.

2.4.4 Sobel Filter

Now that the image has been smoothed out, we can now compute the gradients for the image. The Sobel filter is a series of image kernels applied to images to compute the gradient at any given point in the image. There are two kernels used: one for calculating the x gradient, and one for the y gradient. The Sobel filter computes G_x and G_y by looking at the neighboring pixels in that direction, and calculating how much the grayscale intensity changes. The Sobel Filter uses the following 3x3 kernels for calculating the x and y derivatives:

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

(a) G_x Kernel

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

(b) G_y Kernel

Figure 5: Sobel Filter Kernels

With the x and y gradients, we then compute 4 additional quantities. We compute the magnitude of the gradient by taking the Euclidean norm of G_x and G_y. Then, we compute square of each gradient, and the product of the x and y gradients. These are all used by later stages of our pipeline.

This module uses the generic 2D convolution model for implementing the image kernel calculations, using a 3 row circular buffer to hold the input pixels. To deal with the edges of the image, we mirror the corresponding rows and/or columns to fill in the missing pixels. This module streams in the 8-bit gray intensity values from the Guassian filter. It has several different output streams. It outputs the gradient magnitude, one per pixel, back to the processing system through AXI DMA. This is later used by feature matching to find matches. It also outputs the x and y gradients to CORDIC, two data values per pixel, each 4 byte integers. The final output stream is all of the gradient products, three per pixel, which are 4 byte integers streamed to non-maximum suppression.

2.4.5 Non-Maximum Suppression

With all of the gradient products, we can now figure out which pixels in the image are feature points, and give the row and column location for them. Non-Maximum Suppression consists of two parts. The first part is calculating the Harris response for each pixel. The second part is performing the suppression of non-maximum corners. The Harris response of a pixel is essentially a measurement of how much the pixel is like a corner. To determine if a pixel is a corner, we look at a window around the pixel. Intuitively, if we shift this window in any direction, and resultant window is very different from the original, then this means that the pixel is a corner. In other words, if there are large changes in all directions from the pixel, then it is likely a corner.

We can approximate this effect by using the gradient products computed by the Sobel filter. For each pixel, we look at a window of its neighboring pixels. Then, for each gradient product G_x^2, G_{xy}, G_y^2, we compute the sum of all of the neighboring pixels' gradients, producing S_{xx}, S_{xy}, S_{yy}. Note that this is equivalent to applying an image kernel of all ones to each pixel in the image. Then, with these values, we use it to form the following matrix, and use that matrix to calculate the response:

$$M = \begin{bmatrix} S_{xx} & S_{xy} \\ S_{xy} & S_{yy} \end{bmatrix}$$

(a) Harris Matrix

$$\begin{aligned} R &= det(M) - k(trace(M))^2 \\ &= S_{xx}S_{yy} - S_{xy}^2 + k(S_{xx} + S_{yy}) \end{aligned}$$

(b) Harris Response

Figure 6: Sobel Filter Kernels

The Harris response is the determinant of the matrix minus the trace of the matrix squared, scaled by a constant k. The constant k is an empirically determined constant, and makes the corner detector either more or less selective. With all of the Harris responses, we then apply a threshold, discarding any pixels whose Harris response is below a particular R-value. This will now give a set of corner feature points, but this usually gives us a very large number of feature points. To make a more manageable number of corners in the image, and to filter out additional corners, we then apply non-maximum suppression. For each corner in the image, we look at a window of its neighboring pixels. If the given corner is has the highest Harris response in its local neighborhood, then we keep the corner; otherwise, the corner is discarded. With only the local maximum corners remaining, we now have our final list of corners.

For our version of the algorithm, we use a 3x3 window to look at neighboring pixels when computing the Harris response. Our k value for computing the response is 0.125, and we threshold on an R value of 10000, so if the response is below this, the pixel is not considered a corner. For non-maximal suppression, we use a 9x9 window of neighboring pixels to find the local maxima. Like generic 2D convolution, this module must also buffer the streaming data into rows to be able to use the windows. For non-maximal suppression, this is a 9 row buffer.

This module streams in the gradient products as input from the Sobel filter. The module outputs the list of corner feature points back to the processing system through AXI DMA. The corner list is a list of row, column tuples, where the row and column are each represented as 2 byte values, and packed together into a 4 byte structure. The list is terminated by a value of -1.

2.4.6 CORDIC

With the x and y gradients, we now want to figure what the orientation of the corner is. The orientation is essentially the direction of the corner; specifically, it is the angle between the x and y gradient. To compute this angle, we need to take the arctangent of the two values. The arctangent, even in software, is an extremely expensive operation, and even more so if done naively in hardware. Without an intelligent implementation, it would difficult even to synthesize the arctangent function into hardware.

CORDIC is an extremely efficient hardware implementation for arctangent. It generalizes to other trigonometric, hyperbolic, and exponential functions as well. The way CORDIC works boils down to a binary search and a lookup table. To find the angle we're looking for, we iteratively rotate the the points until the y coordinate goes to zero. Each time we rotate, we accumulate all of the rotation angles, so when we're finished the resultant sum will be the angle between the x and y values. We calculate the rotations by using the tangent function. To speed up the calculation of rotation, we use a lookup table for the tangent function. CORDIC only uses angles whose tangent values are the inverse of powers of two; this allows for shifts to be used to scale the X and Y values, making the computation even quicker. Some example angles are 45°, 25.565°, among others.

Fortunately, Xilinx has an IP block that implements the CORDIC algorithm, and it supports the computation of arctangent. Thus, we simply use Xilinx's IP block for CORDIC. The IP block complies to the AXI4-Stream protocol, so we simply can connect our modules to CORDIC, and the protocol will handle the transfer. This is a great example of why using the standardized AXI protocol can be of great benefit.

This module streams in the x and y gradients from the Sobel filter. The module outputs the angle between the x and y gradients as a fixed point value to the Orientation to Bin module.

2.4.7 Orientation to Bin

To simplify dealing with the orientations of each feature, we sort them into eight separate bins. That way, when we go to compare the feature points, it's relatively simple to compare orientations. We simply divide the angles into eight separate bins, and assign the appropriate number to the pixel based on its dominant orientation. This module streams in the 4 byte orientation angles outputted by the CORDIC module. It outputs the orientation bin number, one per pixel, as a 4 byte integer back to the processing system through AXI DMA. This is later used by feature matching to compare the orientation of feature points.

2.4.8 AXI/AXIS Protocol

The Advanced eXtensible Interface (AXI) is one of ARM's Advanced Microcontroller Bus Architectures (AMBA). AMBA is an open standard protocol for an on-chip interconnect inside the processor, specifically for an SoC design. AXI is one of the most widely used AMBA protocols, and has adoption in hundreds of different devices. The AXI protocol supports complex interconnects of master and slave interfaces, supporting up to hundreds of masters and slaves.

The Zynq-7000 series processors comply to the AMBA4 set of protocols. There are three protocols from AMBA4 that our system specifically uses: AXI4, AXI4-Lite, and AXI4-Stream (AXIS4). AXI4 is the protocol used for large data transfers between the processing system and the programmable logic. AXI4-Lite is a subset of the AXI4 protocol and is used to small data transfers, namely control signals and writing to memory mapped control registers. We are going to skip over the details of these, as these protocols are handled by the PS and Xilinx's IP blocks. AXIS4 is a variant of AXI4 designed for streaming, and this protocol is used everywhere throughout our system, namely to stream data out of DRAM by communicating with the DMA IP blocks.

The AXIS4 protocol is designed for one-way transfers from a master to a slave. This protocol was essentially designed for streaming data transfers on an FPGA, and it is ideal for our use case. All of our modules' input interfaces are AXIS4 slaves, and of their outputs are AXIS4 masters, so our entire system uses a standardized protocol for communication. This is good for two reasons. First, we can easily replace any module in the pipeline with another that complies to AXIS4, and we would need to change very little about the rest of our setup. Second, our system is not necessarily limited to the Zedboard and Zynq-7000 SoC or even Xilinx boards; many ARM-based SoC chips use the AXI4 protocols, and so our modules would be able to easily interface with them as well.

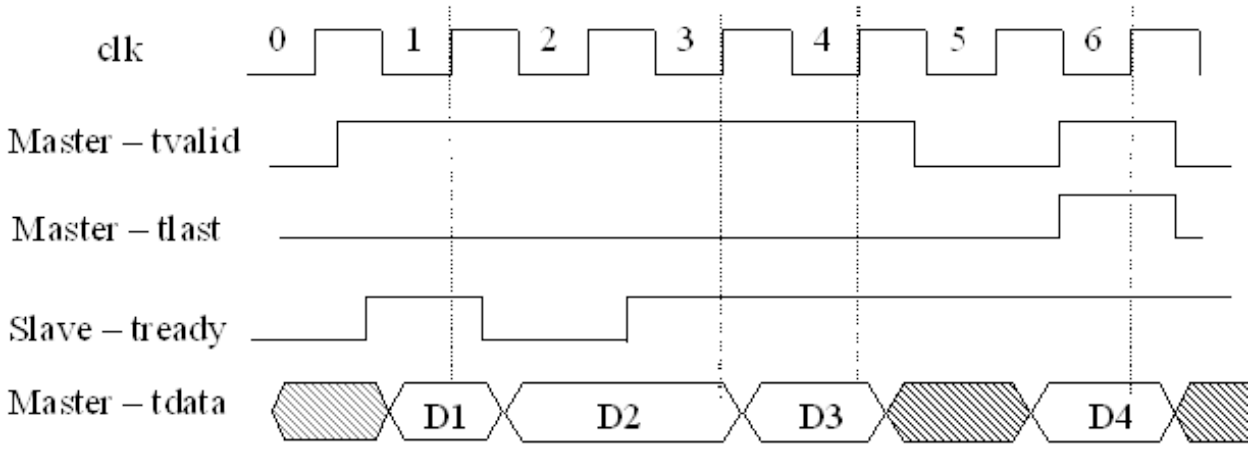


Figure 7: AXI4-Stream protocol.

Above, a typical AXIS4 transfer is pictured. This shows the AXI4-Stream interface used by Xilinx's IP blocks. Both the slave and master signals that mark when they are ready to receive and transmit data. When the master is ready to send data, it asserts TVALID to indicate the data on the line is valid. When the slave is ready to receive data, it asserts TREADY to indicate that it is ready to consume the data on the line. Thus, the data is transferred only when both TVALID and TREADY are asserted in the same cycle. The final signal, TLAST, is used for packet-based transfers, and thus is used for DMA transfers, because there is some set amount of data that is transferred. TLAST is asserted by the master when it sends the last piece of data in the packet, and this indicates the end of the transaction. One signal in the protocol is not pictured above: TKEEP. TKEEP is a one-hot encoded signal from the master that indicates which bytes of the data on the line are valid. This is only used at the end of a packet transfer, when the data being transfered is not aligned (e.g. 4-byte transfer width, but the last transfer is only 2 bytes). This signal must be

[6] http://player.slideplayer.pl/3/1291887/data/images/img28.png

set to all active-high at any other time during the transfer.

In terms of implementation, conforming to this protocol is pretty straight forward. The Vivado HLS tool handles the TVALID and TREADY signals for both the inputs and outputs to the modules. Thus, the only two signals our modules need to handle are TLAST and TKEEP. All of our transfers are always aligned to the width of the data stream, so TKEEP is always set to high. Handling TLAST is simple as well. Once, we see TLAST, we know that we've just received the last packet, so we simply need to assert TLAST once we've generated the last output that was based on the last data packet.

2.4.9 Processor to FPGA Interface

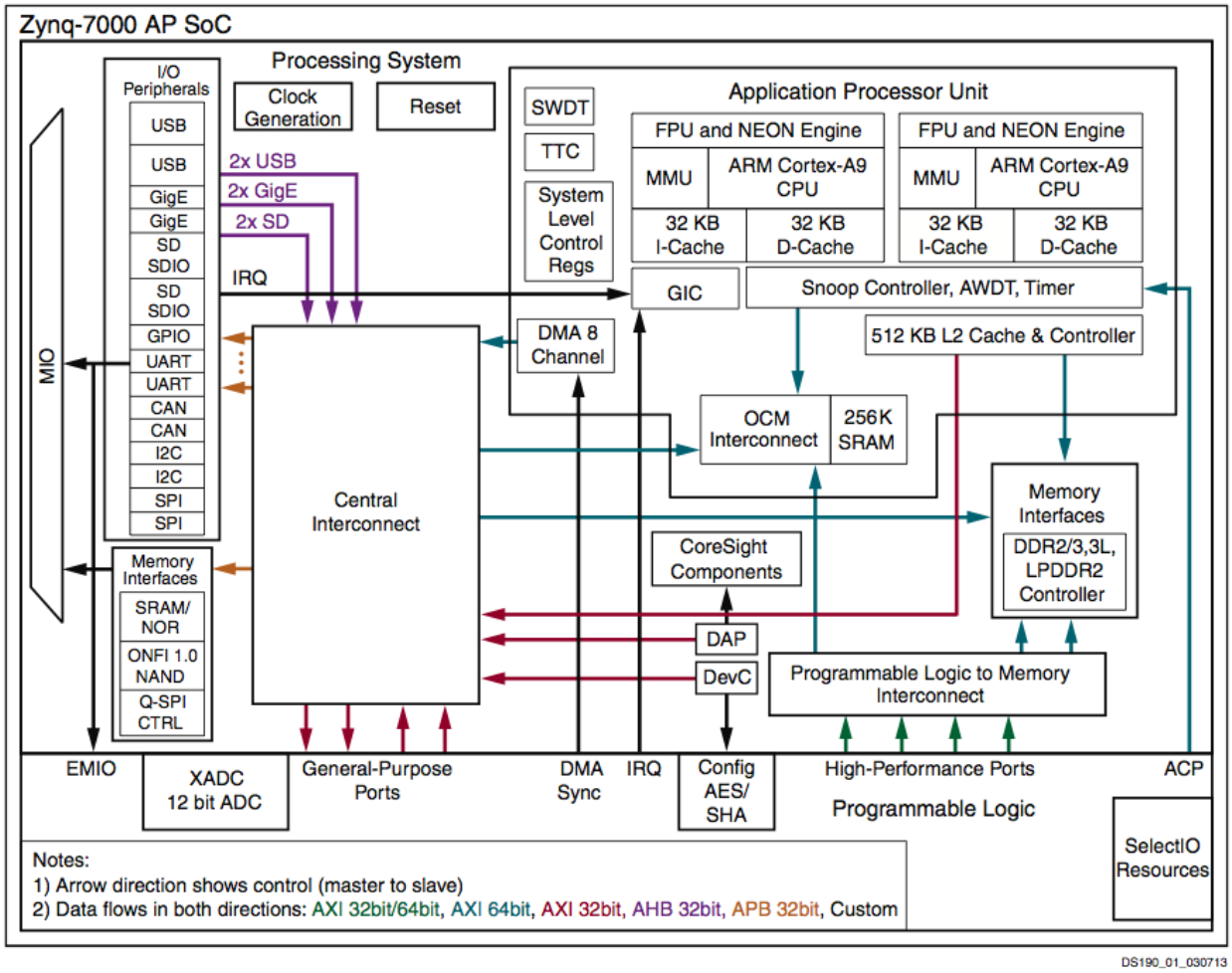


Figure 8: The Zynq-7000 series processing system.

Above, the Zynq-7000 series processing system is pictured. On the 7000 series SoC, all the communication between the processing system and the programmable logic happens through memory transfers and memory mapped I/O. There are three main interface ports out to PL from the PS: the general purpose (GP) port, the high performance (HP) port, and the accelerated coherency (AC)

[6] http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

port.

The GP port is the slowest of the three ports, as it runs through the central interconnect, sharing the bus traffic to memory with all of the hard peripherals in the PS. This typically used for smaller transfers, such as AXI-Lite control signals. The AC port is a special purpose memory port that is cache coherent with the PS. As you can see from the diagram, it connects directly into the Snoop controller and the L2 cache, allowing it to stay coherent with the processors. While this is the lowest latency, it is only the fastest when the data can fit in the cache, and the processors have an amenable access pattern. Otherwise, the cache will be thrashed between the logic on the FPGA and the processors. The third port is the high performance port, which gets a dedicated channel into main memory through a high speed interconnect. The high performance ports get dedicated access to the DRAM, allowing for very high bandwidth and throughput with memory transfers.

For our project, we choose the high performance ports for our transfers. While the AC port offers cache coherency and high throughput, image data is far too large to fit into the cache, and we will be accessing the data in a streaming fashion, so the benefits of any cache locality are minimal at best. Additionally, if we ensure that our memory buffers are uncached by the processor, then coherency is no longer the issue. The high performance ports can support the high throughput necessary to stream the large image data. Also, the HP ports have the largest number at 4, while there is only 1 AC port and 2 GP ports.

The interface between to all of these ports is AXI4. The Xilinx IP blocks for DMA, Video DMA (VDMA), and central DMA (CDMA) provide a nice layer between the modules on the PL and the memory ports. All of these IP blocks communicate with the ports through AXI4, and stream data out the rest of the logic fabric with the AXIS4 protocol. The blocks also accept input streams, and then send that data back to memory via AXI4. In terms of implementation, if our created module respects the AXIS interface, we can simply connect our module's inputs and outputs directly to the corresponding ports of the DMA IP block. For control signals and writing to the IP block's memory mapped registers, the processor goes through the GP ports by default. These are generally small transfers and they do not require the utmost speed, so these parts are the appropriate choice. Each of these block's register maps are written to by the processor through the AXI4-Lite protocol.

2.5 Software Subsystems

2.5.1 Software Stack Diagram

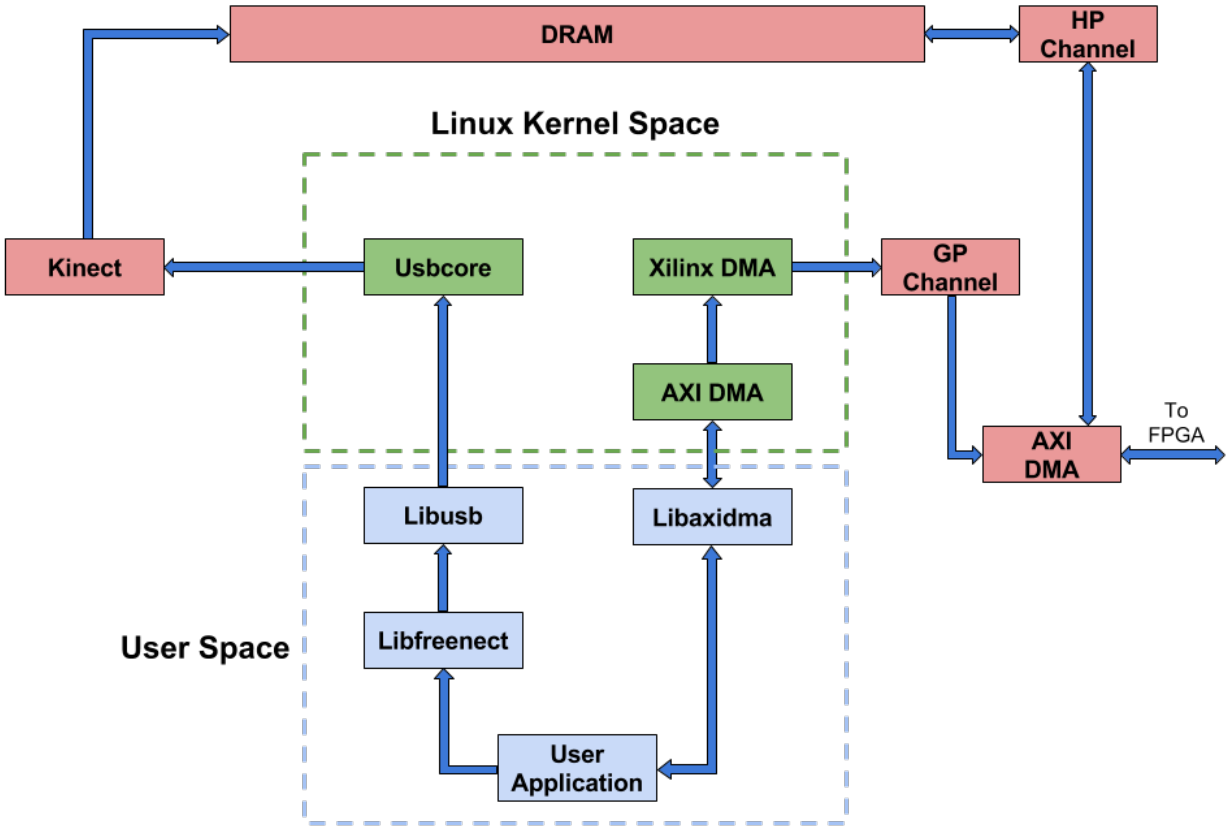


Figure 9: Software Subsystem Overview

2.5.2 Linux Kernel (Xillinux)

For our Linux kernel, we used Xillinux, which is a fork of the Xilinx mainline kernel, forked from the 3.12 Linux kernel. Xillinux notably adds support for Xillybus, which is generic FIFO interface between the processing system and the programmable logic, in the form of a kernel driver. The kernel is maintained by Xillybus, who created the eponymous FIFO IP[7].

The reason we initially chose Xillinux was because we wanted to see if we could use Xillybus for our applications. However, we soon discovered that the Xillybus design requires a separate license to use, and it was not an open source design. To integrate any of our designs with it, we would have had to use their web interface (IP Core Factory) to create our design and then test it; this little amount of control would make it too hard to fine tune our system. We stuck with Xillinux because we already had it setup and booting once we figured it out, so there was little point to switching to another kernel. It is also worth noting that Xillinux and Xillybus also support the Altera Cyclone V series of devices.

[7] http://xillybus.com/xillinux

There are three kernels in total offered for the Zedboard and the Zynq-7000 series SoC devies: Xillinux, Linux-Xlnx, and PetaLinux. Linux-Xlnx is essentially the "vanilla" kernel offered by Xilinx; this is the official mainline kernel maintained by Xilinx for their devices. Xillinux is kernel we used, maintained by a third-party vendor. PetaLinux is another version of the Linux kernel distributed by Xilinx. PetaLinux comes with a large suite of software tools that aid in setting up and deploying an embedded Linux kernel. For example, the PetaLinux tools can setup the root filesystem for the kernel, along with any needed boot loaders, and also has tools for managing device trees. We recommend that everyone, especially those who are new to embedded Linux development, use the PetaLinux kernel, as it the easiest to use and best documented.

2.5.3 Xilinx AXI DMA Driver

The Xilinx fork of the Linux kernel comes with drivers that interface with the AXI DMA, VDMA (video DMA), and CDMA (central DMA) Xilinx IP blocks. Each of these drivers knows about the layout of each IP block's memory map, which registers to write for configuration, and how to properly initiate DMA transfers.

Each of these drivers exposes its functionality through the Linux Kernel's generic DMA layer, which has been mostly standardized. The generic DMA layer requires each DMA driver to implement a set of calls, such as a device configuration function, a transfer preparation function, and a transfer submit function, along with several others. This allows the driver using this interface to be as portable as possible, having little dependency on the specific nature of the hardware. However, it is worth noting that all of the device configuration functions have many Xilinx-specific options, so the end user has to sacrifice some of their driver's portability, needing to use a Xilinx-specific configuration structure.

Unfortunately, all of the Xilinx drivers are platform drivers, meaning that their interfaces are only exposed to other kernel level code, namely other drivers. Thus, the Xilinx driver's functions are not accessible from userspace without another driver to act as the interface between Xilinx's drivers and userspace code. In order to be able to write userspace applications that can use the DMA engine, we would have to either use an existing driver, or create our own to interface. We'll discuss more about this in the next section.

A last point to note is that the Xilinx DMA drivers are both out of date and leave some features unimplemented. The AXI DMA driver in particular has not been modified since 2013, and does not implement multi-channel DMA or cyclic buffer descriptor support. Also, we briefly tried to work with VDMA, and that driver was so far out of date that they actually had some components of the register map incorrect. Thus, we had to update the AXI DMA driver to support cyclic buffers descriptors, which we used for video frame transfers. Cyclic buffers essentially tell the DMA to repeatedly read from a chain of buffers, so it is a continuous transfer.

2.5.4 AXI DMA Linux Driver

When we originally decided to use Linux, we hoped that there would already be a DMA driver in the Linux kernel that would interface we userspace. As it turns out, Xilinx does have a single driver, xvdma[8], that is exposed to userspace, but there are two issues. First, the driver is only for AXI VDMA, which is not the IP block we were using. Second, the driver essentially provides

[8] https://github.com/Xilinx/linux-xlnx/blob/master/drivers/staging/video/axivdma/xvdma.c

no abstraction to userspace, and instead exposes all of the details to the user. Essentially, it is no different than writing a baremetal (embedded) application, running without an OS and directly interfacing with the AXI DMA IP block. Without a nice abstraction, the userspace code would have to manually configure the DMA every time, which is something that really should be done by the kernel driver.

We did find a driver for AXI DMA that accessible from userspace, zynq-xdma[9], which was created by another Xilinx user, but this is essentially just a port of the Xilinx xvdma driver, with the same issues. Like that driver, the zync-xdma driver also exposes all details to the user, thus providing no abstraction. Therefore, we decided that we would write our own driver, which we knew would support the features we desired, and would expose a sane, reasonable, and well-abstracted interface to userspace.

The Linux driver is responsible for two different parts of our system. First, it exposes a memory allocator to userspace that allows them to allocate large, shared buffers that are capable of being used for DMA transfers. Second, it exposes a interface to userspace that allows applications to initiate, control, and respond to DMA. These are both achieved by exposing a character device for the driver to userspace, which the application can interact with to achieve these effects.

When allocating large DMA buffers, there are a few issues that must be considered. DMA memory buffers have two key properties that must be enforced. First, since the device and the processor both access the buffer, the memory must be coherent between them. As the processor has a cache, it may fail to write back some of its data, or some of the data in the cache may become stale. Second, the buffer must be physically contiguous in memory. While the processor has virtual memory support that allows it to see physically noncontiguous pages as contiguous, DMA devices, in general, do not have this support. Strictly speaking, it is possible to circumvent this issue by using the scatter-gather support, which Xilinx's DMA engines support. However, especially when it comes to streaming image data, this is not nearly efficient enough. Page sizes tend to be either 4 or 8 KB, while image data is typically in the hundreds of KBs or low MBs range, meaning that the transfer would need to be split up into hundreds or thousands of individual parts, slowing the throughput down substantially.

Typically, a driver would allocate a physically contiguous memory by specifying particular option to `kmalloc`; however, `kmalloc` can only reliably map physically contiguous regions up to a few kilobytes, which is far too small for image data. The driver solves this issue by using the contiguous memory allocator (CMA) support in the Linux kernel. CMA allows one to reserve a large region of memory at boot time that is reserved for allocating large regions of physically contiguous memory. This is dynamic allocator that is shared by the whole kernel, so our driver does not need to a hog a large region of memory exclusively for itself. Additionally, the CMA is completely transparent to drivers. They make the same allocation calls into the generic DMA layer of the kernel, and instead of using the normal memory allocator, the CMA is used instead. Thus, our driver can simply make a call to `dma_alloc_coherent`, which will give us a buffer that is both physically contiguous, and coherent between the device and processor. This call is portable, but on ARM this means that the page is not cached by the processor, so it only ever accesses it by going directly to main memory.

The driver exposes the DMA interface to the user through a character device, appearing as a character device file under `/dev/axidma`. The user can allocate memory buffers by invoking the

[9] https://github.com/culurciello/zync-xdma

`mmap` system call, with the character device's file descriptor. This invokes a small piece of driver code that calls the DMA allocator and then creates virtual memory mapping for the memory region, so that the userspace code can directly read and write to the buffer. This means the allocated buffer is shared between kernel space, user space, and the AXI DMA device (along with the downstream FPGA modules). Typically writes are buffered by the kernel, but due to the large sizes of images, the sharing is necessary, otherwise constantly copying the data would be too slow. The call then returns the address of buffer in the invoking process's virtual memory space.

The rest of the driver's functionality is exposed through the `ioctl` interface, which provides an arbitrary extension to the standard device system calls. When the user wants to initiate a transfer, or set some configuration, they invoke the `ioctl` system call with the appropriate command number, and the expected arguments. In the driver's interface, DMA devices are represented by channel ID's, assigned by the driver, which are unique, monotonically increasing integer values. Each transmit and receive channel of each AXI DMA device is represented by its own channel id. The driver supports several different `ioctl` commands. It supports the user querying for the available DMA channels. This will return the number of available DMA channels, along with an array of structures, one per channel, that tell what direction the channel is, the channel id, and what type the channel is. The user can search this array to find the desired channels, and in subsequent `ioctl` calls, will need only the channel id to refer to them.

Our driver supports four different types of DMA transfers. It supports one way transfers, both memory to device (write), and device to memory (read) transfers. It supports two way transfers, where you write to one endpoint, and read from another endpoint. Each of these transfers support both synchronous and asynchronous modes, blocking the invoking process until the transfer completes if necessary. Additionally, the driver supports using either an AXI DMA device or an AXI VDMA device, a result of experimenting with VDMA when we were implementing our system. The final type of transfer is a video transfer, which continuously transfers a set of buffers out to a display device. This is achieved through cyclic buffer descriptors, which tell the AXI DMA engine to repeat the transfer once the last buffer is received. For asynchronous transfers, our driver also supports notifications to userspace once the transfer is complete. This is accomplished through POSIX real-time signals, which deliver the specified signal to userspace soon after the DMA completion interrupt is received by the driver. The user can also configure which signal is sent through another `ioctl` call.

2.5.5 Libaxidma

To facilitate the process of writing DMA applications in userspace, and cut down on repeated code, we also implemented a library for the AXI DMA device. This is a thin software layer around the AXI DMA driver in userspace that abstracts the application code from the system calls. It abstracts the AXI DMA device as a object, providing generic open and close methods that initialize and destroy the object. The `mmap` system call is abstracted as a `malloc` function for the library, and the library also provides the corresponding free. All of the `ioctl` commands are implemented as their own function in the library, taking in the arguments needed for the command, and packing them into the expected structure. Last, the library also does some sanity checking from userspace, so that the user does not need to trawl through the kernel log for an error message from the driver.

The purpose of this library was mainly to speed up our code development. For starters, a lot of the initial application code we were writing was repeating code, so this was a cleaner solution. Second, the system call interface, especially to those unfamiliar with it, can be quite confusing. The

AXI DMA library serves to abstract away the details of system calls, and instead provides the user with a set of functions that correspond directly to the actions performed.

2.5.6 Libfreenect

Officially, the Kinect is only supported for Windows, though Micrsoft's Kinect SDK. Fortunately, however, there have been a few open source projects that have worked hard to crack the Kinect protocol, making it accessible from other systems. The most successful of these has been the Open Kinect[10] project. For the Kinect V1, they created the libfreenect[11] library to interface with the Kinect.

The library supports a ton of configuration features with the Kinect, but there are only a few that we care about. The library allows us to configure the Kinect to send us the raw Bayer pattern at a 640x480 resolution at 30 frames per second. It also allows us to override the library's internal buffer, so we can give it the address of a DMA buffer to place the image data into. Finally, it has callback support, so it notifies us once the transfer is complete.

The library also offers the ability to get the Kinect image data as an RGB image. However, the Kinect itself can only stream out the raw Bayer pattern, so the library performs the Bayer conversion. This conversion is extremely well documented in their code[12], and we used it as the basis for our hardware implementation for Bayer to RGB conversion.

2.5.7 Libusb and Usbcore Linux Driver

Libusb is a cross-platform, userspace library for dealing with USB devices and transfer from userspace[13]. It enables drivers for specific USB devices to be moved into userspace, allowing for much faster development, and quicker debugging time. Libfreenect uses the libusb library as the USB backend for interfacing with the Kinect, so libusb is completely transparent to our userspace application. Libusb is resopnsible for setting up the USB asynchronous transfers, and also handling the callbacks from the Linux driver.

The Usbcore driver is a large software stack within the Linux kernel for dealing with USB devices. It is a generic USB interface, and consists of several layer for dealing with different types of USB devices. This is the driver that the libusb library interfaces, and is ultimately responsible for communicating with the Kinect, and also setting up the Kinect to stream data into the DRAM buffer. This driver is also completely transparent to our userspace application.

2.5.8 Feature Descriptors

Our application used PS to calculate featuare descriptors. Once the bins, magnitudes, and feature locations are received from the PL fabric, the PS can easily compute the descriptors for each feature. For all features found on current frame, it takes 16x16 window around the feature. It then divides 16x16 window into 16 different 4x4 windows. Each 4x4 window will have its own 8-bin histogram

[10] https://openkinect.org/wiki/Main_Page
[11] https://github.com/OpenKinect/libfreenect
[12] https://github.com/OpenKinect/libfreenect/blob/master/src/cameras.c#L453
[13] http://www.libusb.org/

that represents the orientation of the 4x4 window. At the end, every feature will have its own descriptor.

2.5.9 Feature Matching

Our application used the NEON SIMD engine to accelerate the sum of absolute differences (SAD) calculation. The result is then compared with the threshold which is the minimum requirement for two features to be a match. If the reuslt is less than the threshold, it then gets compared with the most recently found minimum. If it is less than the most recently found minimum, we update the minimum; otherwise, we move on to the next feature. At the end, we'll have all the features from previous frame compared with all the features from the current frame.

2.5.10 Userspace Application

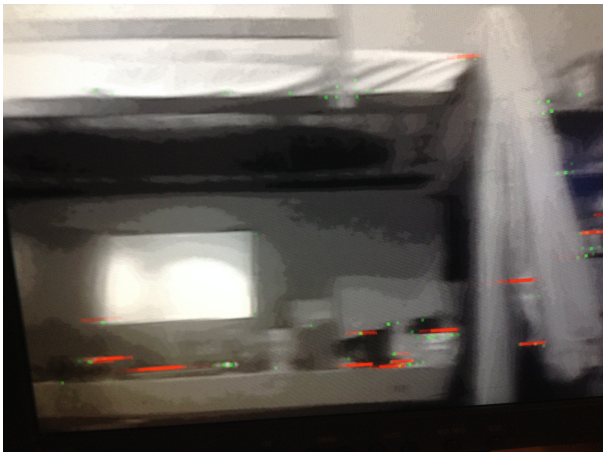
The foundation of our system is independent from the userspace application, so it is very easy to switch out and replace the application we used with any other. For our demo, however, we did use a specific application, mainly because it handled overlaying data on the image. The main purpose of the userspace application is essentially as a control master. The userspace application allocates the needed buffers for frames, initiates the transfer from the Kinect, and also initiates all DMA transfers. It also serves as a point of synchronization; it waits on the DMA and Kinect transfers to complete until it starts the next set of needed transfers.

The demo application also was responsible for overlaying the data on the image. For the corners, it was simple; we simply drew boxes around the corner feature points. To represent feature matches, we drew lines between the previous and current location of the feature. We used a simple Bresenham line algorithm to draw the lines.

3 Results

3.1 Demo

(a) Feature Points with Orientation



(b) Feature Matching, Left Motion

(c) Feature Matching, Right Motion

Figure 10: Demo Screenshots

We were able to successfully accomplish all of the goals we set for the project, so for the demo we were able to show our computations in real time overlaid on top of the image. For the demo, we used a Kinect as the camera source, and streamed a 640x480 image at 30 frames per second.

We had two displays running for our demo, pictured above. On the first display, figure (a), we overlaid all of the corner feature pointers we detected, along with the dominant orientation. This captures all of the information up to feature matching itself. On the second display, figures (b) and (c), we again overlaid the feature points, and we also overlaid the feature matches. We did extremely basic motion estimation, coloring each match a different color for either up, down, left, or right based on its motion. The line was drawn from the current feature point to its previous position in the image.

As we'll soon discuss, our system was actually capable of a higher frame rate than 30 fps; the bottleneck in our system was the Kinect camera itself. It was our speed that allowed us to perform draw the overlays on the image live. While corners can easily be drawn in real-time, drawing the feature matches on the image is always done after the fact and offline. However, since our implementation was quick enough, we were able to perform the feature matching overlays in real-time.

3.2 System Throughput and Performance

System Stage	Latency (Per Frame)	Frame Rate (640x480 Resolution)	Throughput
Kinect Image Capture	33.33 ms	30 fps	8.79 MB/s
PS to PL Transfer	1.25 ms	800 fps	234.38 MB/s
Bayer to Grayscale	3 ms	333.33 fps	390.63 MB/s
Gaussian Filter	3 ms	333.33 fps	390.63 MB/s
Sobel Filter	3 ms	333.33 fps	390.63 MB/s
Non-Maximum Suppression	9 ms	111.11 fps	130.21 MB/s
CORDIC	7 ms	142.86 fps	167.41 MB/s
Orientation to Bin	3 ms	333.33 fps	390.63 MB/s
PL to PS Transfer	1.25 ms	800 fps	234.38 MB/s
Feature Matching	11 ms	90.909 fps	106.53 MB/s
Overlaying	12 ms	83.33 fps	97.66 MB/s
Logic Fabric Total	**9 ms**	**111.11 fps**	**130.21 MB/s**
Total (Without Overlay)	**20 ms**	**50.00 fps**	**58.59 MB/s**

Table 1: System Performance

As you can see from the table, our system is able to outpace the frame rate of the Kinect. The Kinect frame rate determines the window of time in which we can compute and overlay the corners and feature matches on the image. For performance discussions, we are going to ignore the latency for overlaying, because the overlay was only for our public demo. In a real system that would use our accelerator, it would only care about getting the matches and feature points, so once we finished, we would simply send our data to the next stage of the processing pipeline. A couple of things to note about the numbers. First, all of the modules on the PL fabric are fully piplined, so their latencies are overlapped, and the total latency ends up being approximately the longest stage's latency, which is non-maximum suppression with a latency of 9 ms. This is a result of the large size of the data, so any startup time for each module is negligible.

Second, our system has two synchronization points, both when the boundary between the FPGA

and processor is crossed with DMA. This is because there is no easy way to pipeline the interaction between the PS and PL, either must wait for the other to be fully complete before they can start. Otherwise, either system cannot know whether or not it is reading invalid data from the other. The result is that the latencies on either side of the boundary are summed. The latency of getting the image from the Kinect is the window in which we have to process all of the data. For feature matching, we must wait until all modules on the PL are complete before the process can begin. This causes the latency of both to be serialized, leading to the 20 ms overall latency of our system. This gives our total frame rate as 50 frame per second. If the PL modules are considered in isolation, we can process up to 111.11 frame per second.

Another thing to note is the scalability of our system. Given a sufficient amount of resources, the latency of all of our system's modules will scale linearly with the resolution of the image. Thus, our system would be able to handle higher resolution images, such as 720p or 1080p. There is also room in our design for optimization; many of these optimizations we skipped over for the sake of time, and because we knew that the Kinect was the bottleneck. All of our design is clocked at 100 MHz, but it would be possible to clock it higher frequencies, which would cut down on the latency. The AXI DMA engine is also capable of higher speeds. The reference guide puts its maximum transmit throughput at PS to PL throughput at about 400 MB/s, and the PL to PS throughput at 300 MB/s[14]. This is also only at the default settings; other parameters such as the streaming data width, burst size, etc. can be changed to fine tune the performance. In short, while our numbers point to our system throughput at 50 frames per second, it would not be difficult to optimize our system further to achieve higher throughput and frame rates.

3.3 Resource Utilization

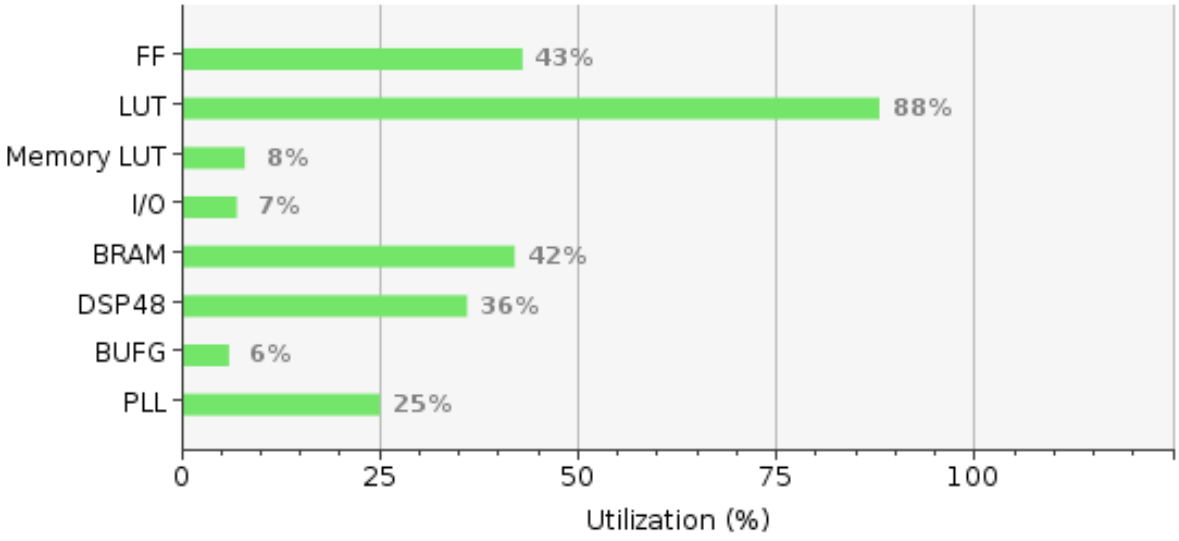


Figure 11: System Resource Utilization

These numbers come from the synthesis report given by Xilinx's Vivado tool after our system is synthesized into a bitstream. The one thing that sticks out in our resource utilization is the large number of look-up tables (LUT), or logic elements, that we use in our design. The reason for this

[14] http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

is that the cyclic row buffers that are utilized to store the incoming rows of the image are all placed inside the fabric. The reason for this is that we wanted to make the modules as quick as possible, so we stored the buffers in the logic elements. We could have also stored them in the Block RAM, but since we had many remaining logic elements, we decided to store it directly on the PL fabric.

3.4 Software Comparison

A natural question that arises with our system is: is it faster than a software implementation? Did the FPGA really provide a large performance boost? The answer to both of these questions is yes, especially when the system is trying to achieve real-time performance. The ability of an FPGA to efficiently pipeline a stream leads to significant performance gains over equivalent software implementations.

Our first reference comparison comes from a research paper on real-time feature matching for medical display devices[15]. This paper is an interesting application of feature matching. In short, the goal of this paper is to be able to assist surgeons in identifying the orientation of and regions of interest on an organ when using a laparoscopic camera, which is a camera mounted on the surgical instrument. Using a 3D reference model, the system is able to map the 3D model of the organ being operated on by matching features between the laparoscopic camera image and anchor points on the 3D reference model. The results were measured on a 2.26 GHz Intel Core2 Duo. The resolution of their images was 704x480, so it was reasonably close to our image resolution. For just extracting Harris corner features and their SIFT descriptors, it took them an average of 0.238 seconds. That puts their throughput at 5.41 MB/s, which is 1/10th of our system speed with feature matching, and around 1/26th of the throughput of our feature extraction pipeline. Even with feature matching, our system significantly outperforms this reference.

The second reference comparison is our own software implementation that we used for verification. Note that this is a custom implementation of the Harris corner detection, and it is not necessarily the most optimal implementation, so the results should be taken with a grain of salt. To process a 640x480 image, and perform only SIFT and Harris corner detection, took 600 ms on a modern Intel processor with a single thread. This puts the throughput at 1.95 MB/s, which is about 50 times slower than our project's implementation, with feature matching included. If we only consider our Harris and SIFT pipeline, it is well over 100 times slower. In both software implementations, our version of feature matching far outperforms the equivalent software implementation. Below, the comparison between the three implementations' performances is pictured:

[15] http://ranger.uta.edu/~gianluca/papers/PuCaMa_AECAI12.pdf

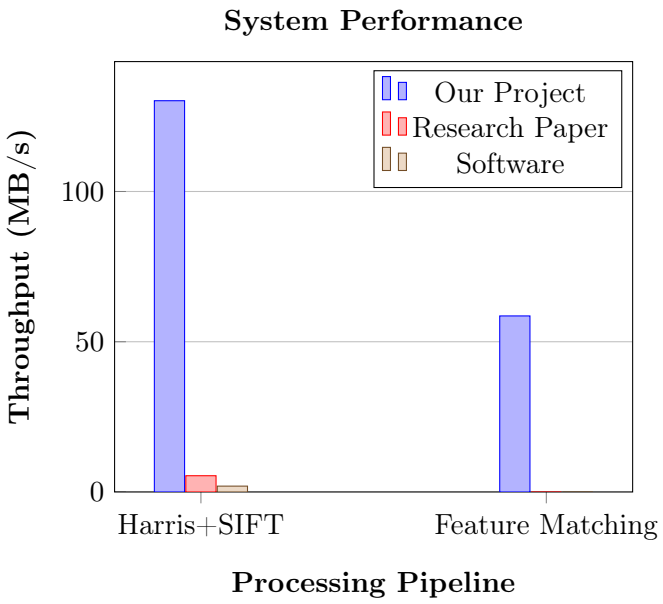


Figure 12: Performance Comparison Bar Graph

4 Personal Statements

4.1 Brandon Perez

For our project, I mainly worked on the system integration, namely integrating the processing system, programmable logic, and the interfaces between our various submodules. From the outset, we knew that we would be using Linux, so my first task was getting Linux booting. This was where I hit the first roadblock, which was understanding how Xilinx performed its system boot up. Most systems typically simply use U-Boot as the primary bootloader, but Xilinx has a special bootloader that runs before UBoot called the first stage bootloader (FSBL). The documentation on how exactly to generate this is pretty sparse, so it took me a while to figure exactly how the boot up process worked, and how to generate the FSBL from the SDK.

With the Linux setup out of the way, my next task was exploring the PS-PL interface. I spent a significant amount of time on working getting asymmetric multi-processing (AMP), which was ultimately a dead-end. Jared pointed us towards using DMA after an 18-643 lab, so I next worked on getting that up. After some research, I figured out that we would need to write out own driver. I first started by getting Xilinx's test driver working, then I moved on to writing my own driver. Around the time my driver was near completion, Jared was also getting many of the HLS modules up. He would request additional features, and I would add them to the driver. Eventually, I created a clean userspace interface for him to use, and once he had enough of the HLS finished, he was able to use this to rapidly test and integrate his modules.

I consider our project to be a success, and I'm quite proud considering the number of obstacles we faced, and how quickly we were able to get our system up. Jared and I were extremely effective as a team; the small size of our team allowed us to be constantly in sync. I attribute the success of our project to a few main factors: hardwork, good team dynamics, frequent communication, and perseverance. Among these, I think the last was most critical to our success. There were a fair number of large roadblocks our project faced, but we never gave up trying to get over them. In the face of poor documentation, a lack of knowledge, and difficult problems, we would always try just that much harder to solve them, and it paid off big. Jared worked especially hard at the end of the project. Both of us worked hard, but Jared implemented a ton of our system towards the end that took our project to the next level. Also, I was extremely skeptical of Xilinx's HLS, from the beginning our project to even a good way through it. Jared, however, never relented when I tried to convince him that Verilog would be more efficient, and in the end this paid off huge. If we had decided to use Verilog instead, there's no way we could have implemented so much so quickly.

That being said, our project and work was certainly not without its flaws. There was huge lag time in our project, nearly 2 months, that it took for both of us to learn the Xilinx tools, and get the system setup and ready for development. Unfortunately, this was essentially unavoidable, given how much we simply had to learn to get a system of this complexity up and running. Nonetheless, it was something that we did not plan for, which brings me to the next flaw. We never had a truly solid schedule and design in place. At the beginning of the semester we did try to come up with a solid design and schedule. However, the problem with the schedule was that the learning curve time threw our initial schedule so far behind that it essentially became useless. With our design, we were constantly learning so many new methods and new things that it constantly invalidated previous iterations of our design, so that too became obsolete. This is not entirely our fault though; it is much impossible for us to come up with a great schedule and plan with the information we had at the beginning of the semester. We were essentially trying to hit a moving target.

The last flaw deals personally with me, rather than team as a whole. In hindsight, my workload was definitely a little to heavy for a capstone semester, and I wish I could have devoted even more time to the project. During this semseter, I was also taking compilers (15-411), was the chair of Build18, and was also an executive officer for HKN, all of which ate up good portions of my time, though Build18 the most. It was not for a lack of trying, though, because at the beginning of the semester I was also taking 18-370, was slated to be an 18-240 TA, and was working on research project. Once I started to comprehend how much work my semester would be, I quickly started dropping some of these, though 18-370 later than I would have liked. Nonetheless, even with my adjustments, I think my schedule was still too heavily loaded for a capstone semester.

While I do like 18-545 as a whole, I think there are definitely some improvements that can be made. The labs in their current form are not particularly useful, and eat up valuable design time early on. The labs do not really teach much because they are completely unstructured. It would be far more useful if the labs were structured like tutorials. The labs are also pretty irrelevant to most projects. It would be useful to have some labs on setting up the display, booting Linux on the a Zynq-7000 system, or something similar. I would advise that the labs for this class should be updated and revamped.

Along the same lines, I think there is too much logistical overhead, and little benefit from it. While I understand the reason for weekly status reports, their benefits seem rather small when we rarely ever receive feedback on them. The same goes for the weekly presentations. I think there are far too many and they just simply distract from the project. It would be far more effective if we instead had meetings directly with the instructions. I think there should be no more than a beginning of term, midterm, and end of term presentation.

The last issue I have with the class is that it is a little too independent. I understand that part of the point of the class is experience designing a project on your own, but the class would benefit from more TA and instructor intervention. The class does very little to teach us about the Xilinx tools, and I think a lot of groups waste their time teaching themselves things that could have been taught to them much quicker. This then takes away time they could spend building their project. I think the class should have more explicit instruction with the Xilinx Tools. Along the same lines, the TA's could be more involved in helping the teams out. I understand that there was little precedence for what our project was doing, so there was little that the TA's could do to help with our issues. That being said, the TA's for this class seem far too disengaged, and I think that they should be taking a more proactive role.

4.2 Jared Choi

For our project Feature Matcher Accelerator, I worked mostly on the application side - this includes all the modules for the Harris corner detector and SIFT. Before I got to implement the system, I spent the first two months learning Vivado tools - this includes Vivado, Vivado SDK, and Vivado HLS. Fortunately I was taking 18643 taught by Prof. Hoe, and the class taught me how to use most of the tools. I went through all the Vivado tutorials(from Xilinx, AVNET, and various developers) that I found on the internet. After about a month of playing around with the tools and testing stuff out, I began implementing our system late October.

I started with the Harris Corner Detector which has 5 different submodules. Even though all my modules were written in C++, I couldn't simply copy my software code to Vivado HLS. In fact, software codes gave me inefficient designs with high latency and poor resource utilization. It turns out Vivado HLS requires the users to provide an explicit design similar to RTL. But once I got used to working with Vivado HLS, the rest was very straightforward.

While I was able to implement the entire Harris Corner Detector on the PL fabric, I couldn't implement SIFT in PL fabric. Instead I had to implement half of it on the PL fabric and the other half on the Processing System - using both Dual ARM processor and Neon SIMD engine. This was because feature matching and feature descriptors were easier to implment and faster on the PS. I made it even faster by implementing feature matching in Neon SIMD engine inside the PS. Initially I used the Neon Library but it was too slow so I tried out the Neon intrinsics. Intrinsics didn't meet the timing requirement so I wrote the assembly code which met the timing requirement and worked out perfectly.

There were couple roadblocks but overall I was able to get the entire system up and running in a month. We could've easily gotten visual odometry to work; however, we had trouble with xbox kinect. Xbox Kinect can only send either RGB or depth data but not both. I thought about using two kinects to get both data but since we were behind schedule, I decided to spend time polishing what we had and work on the demo.

Overall the project was a success. Brandon and I worked great as a team. Brandon got the AXI DMA driver to work from Linux - this made integration and verification easier. In addition, Brandon was able get the VGA to work with the DMA driver which made real-time streaming possible. We were first very skeptical about the Vivado tools - especially Vivado HLS. At one point we thought about giving up on Vivado HLS and writing our own System Verilog codes. After talking with Prof. Hoe and couple of his students, I decided to stick with Vivado HLS, and it definitely paid off.

There are many things that can be improved in this class.

1. Lectures are pointless - I get the idea that this class wants to teach us what it's like to work on a project at a company. However, I think this class should teach us more on how to use Xilinx boards and Xilinx tools.
2. Labs are outdated - teach us how to use I2C, UART, DMA, or HDMI.
3. Weekly report/presentations are pointless - I really don't understand why you guys make us do this. I understand doing this once a month or at the end but this class definitely takes it to another level.
4. No Feedback/Help - I was shocked that nobody really knew how to use Vivado tools. Since nobody knew how to use it, we got no feedback or help on our project. Luckily, I was able to

get some help from couple Phd students and from Prof. Hoe.

5 Statement of Permission

The members of this team, Brandon Perez and Jared Choi, hereby give permission to any members of a non-profit or educational group to use our documents and code in their projects, provided that no members of the non-profit or educational group benefit financially from these documents and code, and that the original authors of the works are credited. In addition, we give permission for this report to be posted on the 18-545 course website. We are not responsible for any generated intellectual property cores of Xilinx, which were provided to us as part of the end user license agreement from Vivado, and are subject to Xilinx's permissions.

There is no warranty for any of the contents, programs, or intellectual property of this project. The content of the project is provided "as is", without warranty of any kind, either expressed or implicit, to the extent permitted by applicable law. Should any content of the project found to be defective, the end user assumes all the risk and cost associated with servicing, repairing, or correcting the materials contained within. The end user also assumes responsibility for any damages caused, directly or indirectly, by the contents of this project.

If you have any questions about the project you can email us at:

Brandon Perez — bmperez@andrew.cmu.edu
Jared Choi — jaewonch@andrew.cmu.edu

You can find our source code repositories at the following locations:

Main Code Repository — `https://bitbucket.org/aznshodan/545_project.git`
Linux Kernel Fork — `https://bitbucket.org/bperez77/545_linux.git`
Uboot Fork — `https://bitbucket.org/bperez77/545_uboot.git`