An FPGA-Based Hardware Accelerator For The Digital Image Correlation engine

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Keaten Stokke
University of Arkansas
Bachelors of Science in Computer Engineering, 2018

May 2020
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

David Andrews, PhD
Thesis Director

Patrick Parkerson, PhD
Committee Member

Dale Thompson, PhD
Committee Member

Abstract

This work intended to develop a hardware accelerator for the Digital Image Correlation engine (DICe) and compare two methods of data access, USB and Ethernet. The original DICe software package was created by Sandia National Laboratories and is written in C++. The software runs on any typical workstation PC and performs image correlation on available frame data produced by a camera. When DICe is introduced to a high volume of frames, the correlation time is on the order of days. The time to process and analyze data with DICe becomes a concern when a high-speed camera, like the Phantom VEO 1310, is used which is capable of recording up to 10,000 frames per second (FPS). To reduce this correlation time, the DICe software package was ported over to Verilog and a Xilinx UltraScale+ MPSoC ZCU104 FPGA was targeted for the design. FPGAs are used to implement the hardware accelerator due to their hardware-level speeds and flexibility from reprogramability. The ZCU104 board contains FPGA fabric on the Programmable Logic (PL) side that is used for the implementation of the ported DICe hardware design. On the Processing System (PS) side of the ZCU104, a quad-core ARM Cortex-A53 processor is available that runs the Ubuntu 18.04 LTS Linux-based kernel to provide the drivers for USB and Ethernet I/O, a standard file system that is accessed through a command-line prompt, and to run the program's control scripts that are written in C. This work compares the processing time of the DICe hardware accelerator when frame data is accessed via Ethernet-stream or local USB to showcase the fastest option when using the DICe. Both methods of accessing frame data are necessary because data may be offloaded from the camera over Ethernet while it is still recording, or the frame data may be readily available in memory. By providing both a method to access frame data via USB and Ethernet, users have more flexibility when using the DICe hardware accelerator. The work presented in this paper is significant because it is the first known hardware accelerator for the DICe software.

Acknowledgements

I would like to thank the University of Arkansas and the faculty of the Computer Science and Computer Engineering Department for providing me and many other students with the tools and skills that we need to succeed. I could not have asked for a better university, department, or faculty to provide me with my education. The Computer Science and Computer Engineering Department has been my home for the last six years and I will miss my time here. I also want to extend my gratitude to Dr. David Andrews, Dr. Dale Thompson, and Dr. Patrick Parkerson for supporting me as members of my committee. I have a great deal of respect for each of these professors and I am grateful for all they have taught me during my undergraduate and graduate careers.

I want to give a special thanks to my thesis advisor, Dr. David Andrews, for guiding me through this degree. His extensive knowledge and expertise in the field of reconfigurable computing have taught me a significant amount. Without his support, I would not have had the opportunity or desire to pursue this degree. From teaching me in the classroom to pushing me in the research lab, he has been a consistent source of knowledge and wisdom. Dr. Andrews has presented me with countless opportunities that have positively shaped my life and I will be forever grateful to him.

I owe a debt of gratitude to Honeywell FM&T for presenting our research lab with the opportunity to work on the project that led to this paper. Honeywell FM&T provided my colleague Atiyeh and me with the funding and work that allowed us to pursue our graduate degrees. Working with the faculty at Honeywell FM&T, specifically with Mr. Dennis Stanley, provided us with a wealth of knowledge that made us be better engineers.

I also want to thank my colleague, Atiyehsadat Panahi, for supporting me and working with me throughout my two years of graduate school. I could not have made it this far without her assistance. Together we have accomplished many goals and solved countless problems and my education will forever benefit from our time as partners. Atiyeh is one of the hardest working and most persistent students I have ever worked with and I thank her for instilling this work ethic in me.

Finally, I would like to thank my family and friends who have all supported me every step of the way to achieving this goal. Starting and finishing this degree would not have been possible without every one of them. With far too many names to list, I want to take this opportunity to express to my friends how grateful I am for their support and motivation during the most intense part of my life thus far. As I close this chapter in my life I know that they will continue to support me in my future endeavors.

Dedication

To my mother, who pushed me to pursue the dreams I thought were impossible

To my father, who raised me to work my hardest under difficult circumstances

To my brother, who showed me that anything is possible with determination

To my sister, who motivated me to push through any obstacle

This thesis and my many years of education are the results of the support that you each have given me. Thank you for being there for me when I needed it most.

I love you all.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

// TODO: Introduction to this paper, what it entails, etc.

1.1 Motivation

// TODO: Explain the motivation behind this work and why it matters. What is significant and novel about this work? What were the driving factors behind the development of this work.

1.2 Thesis Contributions

The contributions listed below are all a direct result of the work that was achieved through the completion of this project. This research aimed to take an existing image correlation program and accelerate its performance by porting it to a Hardware Description Language (HDL) so that it was possible to target an FPGA. The result of this work is that each of the contributions listed below is significant in their own right.

1. The first DICe hardware accelerator to target FPGAs
2. A DICe design for both USB-based and Ethernet-based frame access with performance comparisons
3. A novel low-latency method for basic arithmetic and trigonometric functions in single-precision IEEE-754 standard format

The Digital Image Correlation engine (DICe) was developed by Sandia National Laboratories to provide government entities and contractors with a tool to better analyze the footage captured from high-speed cameras. One such example of the use of DICe in the field is with the Honeywell FM&T plant based out of Kansas City, MO. This plant is known as the National Security Campus and they perform sensitive work for the Department of Energy (DOE). The engineers at this facility must have the best tools at their disposal to make the best decisions when it comes to the products and materials they develop that keep our nation safe. DICe is one of the tools that they use to analyze high-speed footage to make better, safer, and more secure products. The team that uses

DICe daily has reported to us at the Computer Systems Design Laboratory that the time to process their footage is on the order of days. This means days of wasted time before they get the information they need to make a sound decision concerning their projects. By creating a DICe hardware accelerator, the time to process this data is reduced by leveraging the FPGA fabric in the ZCU104 board. With the flexibility that comes with FPGAs, due to re-programmability, the design can be updated or modified on the fly so that that the user can always be running the most up-to-date methods.

On top of developing an accelerator for DICe, this project yielded two designs that allow for accessing frame data from either a USB port or an Ethernet port. This is significant for users of DICe because each method is needed depending on the scenario. The Phantom VEO 1310 high-speed camera can record up to 10,000 frames per second. This is a significant amount of data in a short period and analyzing all 10,000 frames will take far longer than a second. This presents users with an unbalanced scale that leaves them scrambling to process the data quickly enough. This presents two scenarios that the users are faced with. The first scenario is that as the camera is recording, data can be simultaneously offloaded over Ethernet (most high-speed cameras like the Phantom VEO 1310 have support for this). This means that the data can be received by the processing software and image correlation can take place as data is being collected. This scenario is what drove the motive for an Ethernet-based design and in fact, was the sole design choice for this project for a long time. Scenario two is where the cameras are recording and the data is automatically being offloaded to some memory within a PC. This memory can reside in the internal SSD, HDD, or an external hard drive. This is what prompted the work to create a USB-based hardware design. The user can offload the data to an external hard drive and after the recording is finishing they can plug it into the ZCU104 FPGA to start processing. Both methods are desired by users and are accomplished with this work.

Lastly, a result of this project was the creation of a novel library of Finite State Machine (FSM) based methods for performing arithmetic and trigonometric functions in the IEEE-754 single-precision format. When porting the native C++ DICe algorithms over to Verilog, it was observed that a lot of simple mathematical functions were happening receptively and taking longer than expected. Even when using the native Xilinx Floating-Point Operator IP, trigonometric functions necessary to the DICe algorithms such as arcsine and arccosine were not available. This lead to the development of a custom library that performed all of the necessary functions: addition, subtraction, multiplication, division, sine, cosine, arcsine, and arccosine. This work was novel in that it didn't use any BRAM resources, which were critical in the DICe hardware design, it outperformed many previously developed libraries, and it was developed for low-latency instead of high-throughput. This work was recognized as a published long paper at the FCCM conference in December of 2019. This library is implemented in the DICe hardware design that is presented within this work.

1.3 Thesis Structure

The remainder of this paper is carefully divided into sections and subsections that categorize the content based on its relevance. Up next, in Chapter 2, a thorough background will be provided that gives an overview of image processing, an explanation of what DICe is, how and why FPGAs are used as hardware accelerators and then a breakdown of how DICe is used to set the criteria for this project. Chapter 3 will explain the hardware and software tools used to develop this project and a brief overview of how this project has evolved over the last three years while under development. Chapter 4, perhaps the most significant, will go into extreme detail to explain the DICe hardware and software designs. This chapter will provide an overview of each custom IP block that was created within the hardware design to successfully port the DICe software. The high-level code developed for the control scripts will also be discussed to shine a light on how the software design functions. The results of both the USB-based and Ethernet-based designs will be showcased in Chapter 5. This chapter will show how these methods compare to one another and their practicality based on their given scenarios. In Chapter 6, a discussion will be present that touches on the benefits of using PetaLinux for this project, when compared to the previous method of using the LightWeight IP (lwIP) stack, and also the numerous challenges that were faced during the development of this project. Lastly, the paper will end with future works to be done on this project and a conclusion in Chapter 7. Following this will be a bibliography that will present all referenced material in this paper.

Chapter 2

Background

// TODO: This section will provide background materials on related works that are supported with citations.

2.1 Image Processing

// TODO: The general things to mention about what image processing is and what it is good for. Cite papers about how image processing is used. What kind of results and benefits are produced from image processing/correlation?

2.1.1 What DICe Is

// TODO: Explain what DICe is. Who it was developed by, what applications it is used for, and why it is significant. How does DICe differ from other image processing/correlation programs? Support with citations.

2.2 Hardware Acceleration Via FPGA

// TODO: Speak to why FPGAs are used and in particular as hardware accelerators for existing software/programs/applications. What are FPGAs generally used for? What are the pros and cons of using FPGAs? Support with citations. Why did we use an FPGA for this particular project. What options did we have?

2.2.1 How DICe Is Used

// TODO: Breakdown the statement of work that was laid out in the Honeywell contract. Number of subsets, frame size, correlation methods, etc. How is DICe used by the people who wanted in accelerated? Why was a DICe acceleration necessary?

Chapter 3

Platforms

// TODO: Discuss what hardware and software was used to complete this project. Why were these hardware/software chosen. Give the reader an intro of what to expect from this chapter.

3.1 Hardware

// TODO: This section should discuss all of the hardware involved in development of this project. The KC705, VC707, ZCU104, Gigabit Ethernet, USB 3.0, SD card for the Linux-kernel boot, Etc.

3.1.1 Xilinx Virtex-7 - VC707 and Kintex-7 - KC705

// TODO: Explain how this project started on the Virtex-7 and Kintex-7, the pros and cons and why we moved to a different board.

3.1.2 Xilinx Zynq UltraScale+ MPSoC - ZCU104

// TODO: Explain what the ZCU104 FPGA is and its capabilities. What does this FPGA provide for this project in terms of the successful completion of the application.

I/O

// TODO: Discuss Ethernet and USB capabilities here. How is the workstation PC used with the FPGA in regards to Ethernet and USB here?

3.2 Software

// TODO: This section will discuss all of the software tools used for this project. Why were these software tools chosen, how were they used, and what did the provide to us in order to complete the application?

3.2.1 Vivado 2018.3

// TODO: Explain what Vivado is and why it was used to program the ZCU104 FPGA. What beneficial features does it have? How was this software used to our advantage?

Vivado 2018.3 SDK

// TODO: Discuss how the Vivado SDK was used. What benefits did it provide to us while it was used? Why isn't it used anymore?

3.2.2 PetaLinux

// TODO: Why was PetaLinux used? What features does it have? How did using PetaLinux work to our advantage? What alternatives would have been used if not PetaLinux? Inform the reader of what PetaLinux is (An Ubuntu 18.04 LTS Linux-based kernel for the FPGA). How does it work? What parts is it composed of? Etc.

3.2.3 Control Scripts

// TODO: Explain the use of control scripts here. The Python scripts that ran on the PC to the C code that ran on the ARM cores previously. How and why we transitioned to C code running on the ARM for USB and C on the PC for Ethernet.

Chapter 4

Application Design

DICe is a dense program that is composed of nearly 100 separate files and thousands of lines of C++ code. To properly port this design over to Verilog, the original application needed to be studied extensively to understand how it runs, what algorithms are used, and what key functions needed to be ported first to meet the project requirements. While the original DICe is exclusively a software program, creating a hardware accelerator for it means that the application design for this project will be made up of a software-based design and a hardware-based design. In Section 4.1 below, the hardware design that is programmed into the FPGA fabric of the ZCU104 board will be discussed in detail. This hardware design is made up of eight Verilog-based custom-developed IP blocks, each with a specific function. After, in Section 4.2, the software design will be discussed to show how the control scripts run the hardware-accelerated design. This will cover both the USB-based design and the Ethernet-based design, how the FPGA memory is accessed from these scripts, and how the high-level code works with the low-level hardware.

4.1 DICe Hardware Design

The original hardware design for this project targeted a Virtex-7 VC707 FPGA, but has since migrated to the Zynq UltraScale+ MPSoC ZCU104 FPGA. This change in hardware was due to the purchasing of new equipment for our lab and the advanced capabilities the ZCU104 has. The most beneficial feature that the ZCU104 provides for this project is the ARM Cortex-A53 quad-core processor on the PS side. This is one of the reasons that the ZCU104 FPGA is defined as a Multi-Processor System-on-Chip (MPSoC). The ARM processor allows for the ability to run high-level C or C++ code directly on the boards PS side that can be configured to transfer data to and from the PL side (FPGA fabric). With that, the ARM processor is also capable of running a Linux-based kernel that provides a file system to the user, the ability to download packages and run high-level applications and configure the FPGA with the proper drivers to use I/O ports such as the USB and Ethernet ports.

The development of the hardware design for DICe is the most significant portion of this project. The design was under development for nearly three years and continues to be refined. The block design for the program consists of the following IPs: the Zynq UltraScale+ MPSoC, a Processor System Reset, two AXI Interconnects (one for memory and one for custom IPs), six AXI BRAM Controllers, 10 Block Memory Generators (BRAM), a Virtual Input/Output (VIO) monitor for debugging, a Clocking Wizard for adjusting the clock frequency, a custom Parameters IP, a custom Interface IP, a custom Gradients IP, a custom Gamma Interface IP, a custom Gamma IP, a custom Subset Coordinates Interface IP, a custom Subset Coordinates IP, and a custom Results IP. Each custom IP will be discussed in length in the sections below and each one serves a unique function for the DICe program.

The hardware design is ready to perform image correlation when the control scripts have passed two images, the reference frame, and the deformed frame, into the BRAM and the parameter data into the BRAM. Once the application has the data it needs to perform its first correlation it will start. First, the parameter data, which is stored in three separate BRAMs, is sent to the Parameters IP, the Subset Coordinates Interface IP, and the Gamma Interface IP. The user is responsible for defining the parameters before the application begins in a file named "Subsets.txt". The parameters data is necessary to specify the parameters of the images that the correlation will perform on and the subsets, or Areas Of Interest (AOIs), within the images that are predefined by the user. These parameters are the number of pixels in a frame, the number of subsets in a frame, the subsets size, the subsets half-size, the subsets X center point, the subsets Y center point, the subsets shape, the width and height of the frame, the user-selected optimization method (gradient-based or simplex-based), and the user-selected correlation method (fast or robust).

Once all the parameter data is in the memory within the design, the Parameters IP forwards the necessary data over to the Gamma IP. The Subset Coordinates Interface IP and the Subset Coordinates IP are the next to start processing. The Subset Coordinates Interface IP is responsible for receiving all of the subsets that are defined for the correlation from BRAM and relaying that data to the Subset Coordinates IP. The "interface" IPs were created because Vivado does not allow multiple IPs to drive addresses to the BRAM blocks. This is what led us to split the parameter data up into three separate memory blocks because each IP requires different data at different times. The Subset Coordinates Interface IP works with the Subset Coordinates IP by sending it all of the needed subset data for each subset. Because the user can pre-define up to 14 subsets, the Subset Coordinates IP needs to receive the data in order when computing all of the subset coordinates. Once the subset information is retrieved from memory, it is sent to the Subset Coordinates IP. This IP receives the following data: the number of subsets in a frame, the subsets size, the subsets half-size, the subsets X center point, the subsets Y center point, and the subsets shape. Once it receives this data for a single subset, it computes the coordinates of each pixel for the subset. The provided information only tells the correlation that a subset of some size exists, but it doesn't tell the correlation where the subset is placed on the frame. The Subset Coordinates IP solves this problem by taking the subsets parameter data and computing all of the pixels and their coordinates that exist within the subset so that the correlation algorithms can locate where the subset is to do further processing.

After all of the subset coordinates have been computed, the Gradients' IP starts. This IP works together with the Interface IP and Parameters IP. First, once the parameters are set and the Parameters IP signals that it is finished, it sends the frame width and frame height to the Gradients' IP. Second, the Interface IP is responsible for sending the reference image data to the Gradients' IP. When the application initially starts, it is provided with two frames: a reference frame and a deformed frame. The reference frame can be thought of as the original frame and the deformed frame is the next image in the sequence that differs from the previous frame. When the first correlation finishes processing, the deformed frame becomes the new reference frame and a new deformed frame is loaded into BRAM over the previous reference frame because it is unneeded at that point. This is where the Interface IP comes into play. The Interface IP connects to both BRAMs and it is responsible for altering which frame is considered the reference frame and which is considered the deformed frame, because they alternate in BRAM, and sending the correct data to the corresponding IPs.

At this point, the Gradients' IP is receiving the correct frame so it can perform its computations. The goal of the Gradient's IP is to compute the gradients of the reference frame in the X direction and the Y direction. Computing the gradients within this IP means finding the difference between two pixels and their intensities. Once computed, the gradients are saved into BRAM's three and four, where three holds the X direction gradients and four holds the Y direction gradients. The purpose of computing the gradients for the reference image is so that the DICe can track motion when compared to the deformed image in the Gamma IP.

The Gamma IP is the largest IP that was developed for this project. All of the data that has been computed thus far, such as the subset coordinates and the gradients, are all used in the Gamma IP to perform the image correlation. The Gamma IP implements a variety of functions to perform the correlation; these will be discussed below in the Gamma IP section. Once the results are computed by the Gamma IP, the information is passed over to the Results IP. This IP receives the results from the Gamma IP and stores them in BRAM five. BRAM five is connected to AXI BRAM Controller two so that the results have an associated address that can then be read back to the ARM processor and stored in a text file.

When the last image correlation run is finished, all of the computed results should be saved into BRAM five. The C control script that runs on the ARM processor can read from this memory within the hardware design. The control script reads all of the results data stored in this BRAM, converts them from IEEE-754 single-precision floating-point format to scientific notation that is human-readable, and stores the final results into a text file that the user can access. The C script is responsible for formatting the text file in a manner that is comparable to the output file from the DICe GUI. It will display the number of each frame that was processed, the X coordinate, the Y coordinate, the X displacement, the Y displacement, and the Z rotation computed for that frame. For the USB-based design, the Results.txt file is computed locally on the FPGA and saved to the

USB drive from the directory where the images were read from. For the Ethernet-based design, the results will be transmitted back to the connected PC over Ethernet where they will be converted and stored on the PC in the same directory where the images were accessed from.

4.1.1 Miscellaneous IPs

The block diagram for the DICe hardware design contains a few Xilinx IPs that are not mentioned in the subsections below. This is because they do not play a significant role in the image correlation and they were not developed in-house for this project. This subsection will discuss the other IPs that are used within are hardware design with a brief explanation of each.

The Zynq UltraScale+ MPSoC is controlled and configured by the zynq_ultra_ps_e_0 IP. This IP represents the brains of the design in that it is what controls all of the boards' processors, I/O, and hardware-based features. Interrupts can be created by other IPs and driven to the zynq_ultra_ps_e_0 IP so that some processing that requires priority can execute first while temporarily pausing all other processor operations. This IP allows us to manually configure various aspects of how the board will operate such as which I/O ports are active, and most significantly to us, the processor clock speed. The ARM quad-core processors have a max clock frequency of 1.334 GHz. Although the requested clock frequency for our design is the max speed of 1.334 GHz, the Vivado tools report that the actual frequency of our processor's clocks is more in line with 1.2 GHz. This max clock frequency is necessary for the processors to be operating as fast as possible when running the C control scripts or when receiving data from an I/O port, like USB or Ethernet. This IP also allows us to generate a PL clock of 150 MHz, the reason for this will be discussed briefly.

The system reset for the hardware design is controlled by the Processor System Reset IP labeled as rst_ps8_0_100M. The reset signal from the Zynq MPSoC IP is routed to the Processor System Reset IP. The IP has a 1-bit signal labeled as peripheral_areset that is connected to the reset input port for every single IP in the design. This controls the reset of IPs, such as restarting an IP or resetting the memory in a BRAM. This reset is ultimately driven from a reset button on the board that can be pressed at any time.

To use and control all of the memory within the hardware design, two AXI Interconnects are used. Each one has a bus that is directly connected to the zynq_ultra_ps_e_0 IP, making it the master, so that it can have control of the bus interface for the design. axi_interconnect_0 is used to connect all of the AXI BRAM Controllers. This provides a relatively uniform address space for all of the memory-related IPs that exist in the address range of 0x00_A000_0000 to 0x00_A000_9FFF. axi_interconnect_1 is responsible for connecting to all of the custom IPs within the design. This is necessary because each of the custom IPs that were developed for this project is classified as AXI4 peripherals, meaning that each IP is connected to the AXI bus and has an address space associated with it. When creating and packaging a new custom IP, the Vivado tools give the user the option to specify the interface mode of the AXI4 peripheral, slave mode or master mode, and the number of AXI registers they would like associated with that IP. For all of the custom IPs in this design, the registers were left at the default setting of four. This is beneficial because these AXI-based slave registers can be written to and read from within the IP, but also outside the IP too, for example, the ARM processor. This allows the ability to have a direct communication link with specific IPs that assists in the flow of the IP and also debugging. The address range for these IPs is in the range of 0x00_B000_0000 to 0x00_B018_2FFF.

The most significant debugging tool that is available in Vivado is the vio_0 IP. This IP stands for Virtual Input/Output and allows the connection of input or output signals from anywhere else in the design. Upon running the design, a window pops up in the Hardware Manager of Vivado for the user that allows them to view the connected signals to the VIO IP. This allows for real-time tracking of signal changes throughout the design and enables the user to verify the design. The current VIO IP in the design for this project has a total of 43 ports connected to it for monitoring various signals throughout the design.

Lastly, to successfully use the VIO IP in the hardware design, it was necessary to attach a "free-running clock source" to the clock input of the VIO. This leads to the addition of the Clocking Wizard IP that is labeled as clk_wiz_0. This IP generates a dedicated clock for the VIO IP so that no errors were experienced. The Clocking Wizard IP outputs a clock with a frequency of 150 MHz so that the frequency is in line with the speed of the rest of the design. On that note, each IP in the hardware design utilizes a clock frequency of 150 MHz. This is because the library of floating-point arithmetic and trigonometric functions that were developed for this project can only run at a maximum frequency of 150 MHz. A couple of the IPs depend on this library for basic functions that are frequently called. One of the major focuses of the continued development of the floating-point library was an increase in clock frequency, but 150 MHz is currently the highest clock frequency that was achieved.

4.1.2 BRAM IPs

Block Random Access Memory (BRAM) is undoubtedly the most valuable resource for the DICe hardware design and it is used for storing large amounts of data within the FPGA. The ZCU104 FPGA contains a total of 4.75MB of SRAM-based memory that is split into BRAM and UltraRAM (URAM). URAM makes up 71% of the total SRAM-based memory on the ZCU104 which comes to 3.375MB. URAM differs from BRAM in that both ports are single-clocked for reading or writing and the URAM blocks can be cascaded together to create larger memory blocks. BRAM, on the other hand, has a read latency of two or more clock cycles and allows for true dual-port usage; this memory makes up 24% of the FPGAs SRAM-based memory at 1.375MB. For this project, both types of memory are indistinguishable and from this point on these memories combined will be referred to as BRAM.

For this project, BRAM is used for buffering frame data and holding onto predefined parameter values that specify how the image correlation is to be performed. Before the image correlation begins, the program must have a defined set of parameter values, such as image height, width and the total number of subsets, that are written into BRAM. To write to BRAM from an external source, the memory needs to be associated with an address within the hardware design. The AXI BRAM Controller is a Xilinx IP that connects a BRAM block, defined as the block memory generator IP, to the AXI Interconnect bus and provides an address range for the memory. With this IP, the memory is visible to the outside world and can be written from an external source, such as the ARM quad-core processor.

The current hardware design utilizes a total of six AXI BRAM Controller IPs and a total of 10 Block Memory Generator IPs. AXI BRAM Controller's zero and one are connected to BRAM's zero and one, respectively. Each BRAM is 512KB in size and they hold the frame data for the reference image and the deformed image (they alternate on which frame they hold). BRAM's three and four are each a size of 415.7KB and are set as standalone blocks, meaning they do not have an address associated with them. Block three holds the gradients of the reference frame in the X-direction. Block four holds the gradients of the reference from in the Y-direction. BRAM five is connected to AXI BRAM Controller two and is 512KB in size; this block is responsible for holding all of the computed results from the image correlation. BRAM's six and seven are each 193.2KB in size and are both set as standalone blocks. Block six is responsible for holding the subset coordinates in the X direction while block seven holds the subset coordinates in the Y direction.

BRAM two is connected to AXI BRAM Controller three and has a block size of 4KB. BRAM eight is connected to AXI BRAM Controller four and has a block size of 4KB. Lastly, BRAM nine is connected to AXI BRAM Controller five and has a block size of 4KB. Each of these blocks shares a common purpose in that they are dedicated to holding the parameter values for multiple IPs that need access to that data. The details of the parameter data will be discussed in more detail below in 4.1.3.

4.1.3 Parameters IP

The Parameters IP is one of the simplest IPs within the design, but it serves a crucial function. Other custom IPs within the hardware design require a variety of parameter values to proceed with the image correlation. These parameter values are the number of bits per image, the number of pixels per image, the number of subsets per image, the width of the image, the height of the image, the optimization method to be used, and the correlation routine to be used. Each one of these data values sets the parameters of the image correlation for other IPs to function. Before the start of the program, the parameter values should be predefined by the user in a text file labeled Subsets.txt. The text file lists the various parameters in order with each data value on an individual line. When the program does start, the C control script will either receive this data from the PC over an Ethernet connection, or the script will locate the file on the USB drive and extract the parameters. Something to note is that the Parameter values listed above are the only ones that are used by the Parameters IP because these values are needed by multiple IPs at any given time and they never change. The Subsets.txt file contains more data such as the subset shape, the subset size, the subsets X center point, and the subsets Y center point. The reason the Parameters IP is so crucial to the DICe hardware accelerator is that, while values like the image height and width can be hard-coded into the IPs, it allows for the user to have dynamic parameters. This grants users the flexibility to perform image correlation using different sized images and change the number of subsets for the correlation.

When the C control script running on the FPGA has the parameter values, the next task is to write the data to BRAM. The control script running on the ARM processor will then use mmap function to map a locally defined variable to the address space of the BRAM in the hardware design. This function is the key to allowing the PS and PL sides of the FPGA to communicate data to one another. Once the variable has been mapped to an address space in the FPGAs memory, it is possible to read and write to the BRAM in hardware by providing a register index value to the local variable. The control script will then begin writing all of the values listed in the Subsets.txt file into three separate BRAM blocks. BRAM two is connected to AXI BRAM Controller three and is dedicated for use with the Parameters IP. BRAM eight is connected to AXI BRAM Controller four and is responsible for providing subset information to the Subset Coordinates Interface IP. BRAM nine is connected to AXI BRAM Controller nine and is used to provide subset information to the Gamma Interface IP. Once all of the parameter values have been written into BRAM and the first two images have been received and written into BRAM by the C script, the hardware design will start the IPs for processing.

The C control script is responsible for sending a start signal to the Parameters IP so that it may begin processing. This is done by using the same mmap function as before, but this time the value of '1' is written to the Parameters IP AXI Slave register. This will write a '1' into a register that the Parameters IP is constantly reading in state one. It is important to note here that the Parameters IP, and the vast majority of the other custom IPs, were developed using Finite State Machines (FSMs) to precisely control the execution flow of each IP. This was implemented by using a case statement in Verilog that only moves to the next condition, or state if the state variable was set in the state that is currently being processed. Now, once this value has been received by the Parameters IP from the C script, it means that the Parameters IP can start processing by moving to the next state.

The Parameters IP starts with a default address value of zero that it will send to its connected BRAM. The address value defines which register should be used from the connected BRAM. The size of each BRAM block can be manually configured in the Vivado tools; in this case, each BRAM that holds parameter values is connected to a BRAM that is 4 KB in size. Each BRAM in the design has a register length of 32-bits or 4 bytes. The Vivado tools allow for these registers to be byte-addressable, meaning that rather than accessing an entire register, or row, of data at a time, a user can choose to look at each byte in the register. The Parameters IP first reads from address zero in the BRAM to receive the data for the height of the image. Next, it increases the address value by four to shift to the next register to read from in the BRAM. After, the IP cycles through two No Operation (NOP) states before reading the next value from BRAM. This is because a standard BRAM requires two clock cycles to read a value and one clock cycle to write a value. This was another motivation for using FSM-based designs for the custom IPs because each state is set to execute in one clock cycle. While most of the BRAM used is classified as URAM, which only requires one clock cycle to perform a read operation, BRAM is still used in different portions of the design and so this is a design choice that was implemented out of precaution and portability.

By the next state, the Parameters IP reads the data from the BRAM for image width.

The same cycle continues where the address is incremented by four and followed by two NOP states. This process is repeated to retrieve the remaining parameter values such as the number of pixels in the image, the number of bits in the image, the number of subsets in the image, the optimization method to be used, and the correlation routine to be used. When all of the parameter values have been received by the IP and set to their corresponding outputs, the last state of the IP sets an output signal labeled as param_done. This done signal is important because it acts as an acknowledgment signal that tells the other IPs, such as the Gradients IP and Gamma IP, that the Parameters IP is finished collecting all of the required information that the other IPs need to operate. This reason is why the Parameters IP is so critical in the design, it drives parameter values to multiple IPs so that they can start processing. When the parameters data exists in BRAM, an address would need to be provided to determine which values to retrieve and multiple IPs are unable to drive multiple address values to a single BRAM at once.

4.1.4 Interface IP

When the DICe hardware accelerator begins processing, it requires two frames to operate on. These two frames are the reference frame and the deformed frame. Cameras capture video by taking a lot of pictures in a sequence. A short video that contains five frames will display these frames in order from one to five. In this scenario, when the DICe hardware accelerator starts, it will receive frame one which will be classified as the reference image and frame two which will be the deformed image. Upon receiving the image data on the program start, the C script will write the data for the reference frame into BRAM zero using the address provided by AXI BRAM Controller zero. The C script will then write the data for the deformed frame into BRAM one using the address provided by AXI BRAM Controller one. Now, the Gradients' IP requires the data for the reference image so that it can compute the gradients based on the pixel intensity values in the X and Y directions. The Gamma IP requires the data for both the reference image and the deformed image so that the image correlation algorithms can proceed. Once the image correlation is finished for these two frames and the results have been computed and saved, the application will begin to operate on the next pair of images.

Initially, BRAM zero holds the data for the reference image and BRAM one holds the data for the deformed image. After the first correlation has been performed on the initial two frames, the first reference image is no longer needed. The initial deformed image, frame 2, will be classified as the reference frame. The C control script is then responsible for retrieving frame 3 that will be classified as the new deformed image. Because the first reference image, frame 1, is no longer needed for processing, this leaves BRAM zero open to store data. The C script will write the new deformed image, frame 3, into BRAM zero. This means that BRAM zero and BRAM one have switched the data that they retain. This poses an issue for the rest of the IPs that they are connected to. If BRAM zero is connected directly to the Gradients IP to send the data of the reference image for processing, by the second round of correlation the Gradients IP, along with the Gamma IP, would receive the wrong frame of data. This is where the Interface IP steps in.

The Interface IP is directly connected to BRAM zero and BRAM one and controls the flow of frame data to the IPs that require it. It acts as an interface between the frame data and the IPs that need the frame data. This IP is essential in the design because it verifies that each IP is receiving the correct frame data and it prevents the processing time that would have been required to write and transmit all of the data in BRAM one over to BRAM zero. Internally, the Interface IP has been called the "ping-pong buffer" because it manages the back and forth cycle of frame data. To add to this, the Interface IP is also essential for allowing each IP to specify the data they require at a particular address. For example, the Gradients' IP could be processing the gradients for the reference frame and it could be operating on pixel six in register seven while the Gamma IP is operating on pixel one in register two. This IP enables the other IPs connected to it to operate independently. This idea of independent operation of custom IPs is explored more in the Future Works in Section 7.1.

The Interface IP operates by maintaining constant communication between the Gradients IP, the Gamma IP, BRAM zero, BRAM one, and the C control script on the ARM processor. The IP starts when the C script on the ARM processor writes to the Interface IPs slave registers. The C script will first write to the first AXI slave register that the

Interface IP has to notify the IP that a new frame has been received. This start signal allows the IP to move to the second state which then waits on signals from the Gradients' IP and the Gamma IP to coordinate which images to transmit. Both IPs will transmit the addresses of the data they require to the Interface IP. The Gradients' IP should be the first to notify the Interface IP that it is processing and needs more frame data with the grad_busy signal. After the Gradients' IP has received all of the data for the reference image, the IP will signal that it does not require any more data and will transmit the gradients data to the Gamma IP so that it can start processing. The Gamma IP will request the frame data for the reference image and the deformed image so that it can start processing. This is the default sequence for the first two frames when the first round of correlation begins. After this, the C script will write to the second AXI slave register and increment the value by one each time a new frame is written into BRAM. This variable name is labeled as frame_counter in the Interface IP and it enables the IP to keep track of which frame needs to be classified as the reference image and which frame needs to classified as the deformed image. The C script will continually update this register to reflect the total number of frames that have been written to BRAM and the Interface IP will continually flip which BRAM input is classified as the reference and deformed image with some clever if-statements.

4.1.5 Subset Coordinates Interface IP

The Subset Coordinates Interface IP works closely with the Subset Coordinates IP to manage the retrieval of the data for each subset that the user has predefined. An image correlation run can have anywhere between 0 and 14 subsets, as defined by the statement of work for this project. A subset can range in size from 2x2 pixels to 41x41 pixels. Currently, the DICe hardware accelerator only supports square and circular subsets. The DICe GUI can support thousands of subsets that vary in size and shape. The difference between subset definitions in the DICe hardware accelerator and the DICe GUI can be further explained in the Future Works in Section 7.1. The DICe hardware accelerator was designed with flexibility in mind for the user. The user can define different quantities and sizes of subsets prior to each image correlation run. This means that the hardware design has to account for these changing parameter values before each run.

When the user has predefined the parameter values, one of the first actions that the C script does is to write these data values into three separate BRAM blocks. The first BRAM has been covered in the Parameters IP above in Section 4.1.3. The second BRAM block that contains the data of the parameters is BRAM eight. This memory block is directly connected to the Subset Coordinates Interface IP so that it can manage and relay all of the subset data to the Subset Coordinates IP for further processing. This dedicated BRAM block of parameter data is necessary because the Subset Coordinates Interface IP will be fetching subset data continuously from the memory block based on when the Subset Coordinates IP needs it. The dedicated block assures that both IPs have the data that they need when they need it in order to process the subset coordinates for the Gamma IP. The Subset Coordinates IP is responsible for taking the subset parameter values and computing the location of each pixel in the image so that each subset can be located. The Subset Coordinates Interface IP is responsible for controlling the flow of this data and sending the right subset to the Subset Coordinates IP when it has requested new data.

The Subset Coordinates Interface IP starts processing when it has received the parameters_done signal from the Parameters IP. The IP then spins in state zero while waiting for a change in the coord_new_subset signal that notifies the IP that a new subset from the Subset Coordinates IP has been requested. When the Subset Coordinates IP starts and requests a new subset, the Subset Coordinates Interface IP jumps to the next state. In this state, the IP computes the address value of the current subset, in this case, the first one. It sets the address for the current subset and relays that address to BRAM eight to locate the register that contains the first data value needed, the subset X center point coordinate. Note that this IP uses the similar two NOP cycles to successfully read a value from BRAM. Upon retrieval of this data, the cycle continues with the IP going to the next state to retrieve the subset Y center point coordinate. After, the subset size value is fetched from BRAM, followed by the retrieval of the half subset size, and lastly the retrieval of the subset shape value. Once all of this data has been collected, the outputs feed the data to the Subset Coordinates IP. The Subset Coordinates Interface IP will automatically jump back to state zero where it waits for the next signal that notifies it to fetch the data for another subset. This cycle continues as long as there are subsets in the BRAM. Upon the retrieval of the last subsets data, the Subset Coordinates Interface will pause in state zero and cease to process.

4.1.6 Subset Coordinates IP

// TODO: Explain how the subset coordinates are computed and used.

4.1.7 Gradients IP

// TODO: Explain the original Gradients IP and how it computes the gradients in the X and Y direction sequentially.

4.1.8 Gamma Interface IP

// TODO: Explain the Gamma Interface IP, why it is used, and how it manages data that Gamma IP needs.

4.1.9 Gamma IP

// TODO: Explain the Gamma IP, the algorithms and functions it uses and the arithmetic library it implements.

4.1.10 Results IP

After each round of image correlation, the Gamma IP produces a series of values for each frame that was processed. Currently, the computed values include the X displacement value, the Y displacement value, and the Z rotation value. Each of these values is provided in the IEEE-754 single-precision floating-point format. The Gamma IP is responsible for sending a signal to the Results IP called results_done each time a round of image correlation is finished. This pushes the Results IP to the next state where it starts to save each of these three result values into BRAM five. Using the same FSM-based architecture for the Verilog code, the results are each pushed into the BRAM block and the address increases to move to the next register for the next value. When the Results IP reaches the second to last state, it jumps back to state one where it waits for another signal from the Gamma IP to notify the IP that more results need to be saved.

On the last round of image correlation, the Gamma IP will send an additional signal to the Results IP called gamma_done. This signal notifies the Results IP that all image correlation is finished and the last round of results needs to be saved. After cycling through the FSM to save the last round of results, the Results IP will use a signal called results_done to write a value of '1' to its first AXI slave register. This register can be read by the C script on the ARM processor to acknowledge that the image correlation is finished. The Results IP is connected to BRAM five which is connected to AXI BRAM Controller two. This provides an address space that the C script can use to read all of the data from the BRAM. From this point, the C script takes over by retrieving all of the computer results, converting them into scientific notation that is human-readable, and lastly formatting them into a text file that can be analyzed by the user.

4.2 DICe Software Design

// TODO: Discuss the use of a Linux kernel on the FPGA and how it worked to my advantage. Discuss the C scripts that were designed to run on the FPGA at the time of boot-up to start the application. What it does and how it operates.

4.2.1 BRAM Access

// TODO: Explain how BRAM is accessed from the Linux kernel.

4.2.2 USB Access

// TODO: Explain how the images on the USB flash drive are accessed from the Linux kernel.

4.2.3 Ethernet Access

// TODO: Explain how the images are streamed over Ethernet using the host PC, and Ethernet driver on the FPGAs Linux kernel.

Chapter 5

Results

// TODO: Provide details on the results of the application. Was it successful and how so? Compare between the USB and Ethernet designs. Was there any part of this project that wasn't successful in terms of the results?

5.1 DICe USB-based Design

// TODO: Provide results and an explanation of the USB-based design where images are fetched from a local flash drive.

5.2 DICe Ethernet-based Design

// TODO: Provide results and an explanation of the Ethernet-based design where the images where fetched from a host PC over Ethernet.

Chapter 6
Discussion

This section provides a discussion on some topics that were not covered in previous chapters. This project was complex and time-consuming to develop due to the dense DICe source code. The DICe hardware accelerator evolved many times as a better understanding was gained of the source code and the hardware that was used. The biggest evolution of this project, and one this paper highlights, is the use of the PetaLinux tools to create a Linux-based kernel on the ZCU104 FPGA. Using PetaLinux drastically changed the way this project was approached and brought several significant improvements to the design of the application. The previous method of using the LightWeight IP TCP/IP stack to use the Ethernet interface was clunky, slow, and error-prone when programmed onto the MicroBlaze soft processor. The complexities of this project also led to a variety of challenges that will be discussed. From working with 32-bit floating-point numbers and Python-based control scripts to scaling up the design to support multiple frames and migrating between the KC705, VC707, and ZCU104 FPGAs, this project had many complex problems that needed to be solved before the application design progressed.

6.1 PetaLinux vs. lwIP

When development for this project started the only two FPGAs that were available were the Kintex-7 (KC705) and the Virtex-7 (VC707). These FPGAs do not contain a PS side like the ZCU104 board does. This means no hard processors, GPUs, or hard IP. The KC705 and VC707 were challenging to work with because every I/O port that was needed for use had to be manually setup and configured within Vivado before synthesis and writing the bitstream. These FPGAs leverage a soft processor known as the MicroBlaze to act as the central control for a design. With these FPGAs, work on the core of the hardware design was able to continue at a steady pace. The biggest set back that was faced with these boards was designing a functioning hardware design that supported Ethernet I/O. A working Ethernet design was finally established after months of problem-solving, but only on the KC705 FPGA. Once this step was achieved, the next step was determining how to interface with the Ethernet port. This is where the LightWeight IP stack comes into play.

The LightWeight IP (lwIP) stack is an open-source TCP/IP stack that is designed for use with embedded systems. Many manufactures, like Xilinx, use the lwIP stack for their systems to provide a full TCP/IP stack that enables Ethernet communications while reducing the number of resources that standard stacks use. At its start, this project utilized the lwIP stack because it was available as an example in the Vivado SDK and also because it was one of two methods to provide a functioning TCP/IP stack to the FPGA. The Vivado SDK tool provided a method to target the MicroBlaze soft processor with C code to implement the lwIP stack. This provided a working template that could then be modified to suit the needs of this application. Using the lwIP stack started with a simple echo server example that communicated with a Python client on the workstation PC.

Once the fundamentals of this code were fully understood, both scripts were then modified to handle the transferring of images. The PC-side Python script was responsible for accessing the images, transmitting the pixel values over Ethernet to the FPGA-based server, and handling the handshaking between the PC and FPGA to delegate when new frames should be sent and when results were being received. The FPGA-based server was responsible for receiving every pixel value for each image, converting the pixel values to 32-bit IEEE-754 single-precision floating-point format, and writing these values to the corresponding BRAM so that the hardware design could act on the data. The FPGA-based server-side was also responsible for generating start and stop signals to individual custom IPs to coordinate the image processing when new frames were received. Lastly, the FPGA-based server-side would be notified from the custom IPs when the image correlation was finished so that it could read the computed results from the BRAM and send them back to the workstation PC for formatting.

The process of using the lwIP stack to transmit data between the PC and the FPGA worked, but poorly. After extensive modifications to the lwIP server parameters that compile into the Board Support Package (BSP) file, the maximum Ethernet speed that was achieved using the Kintex-7 FPGA was 56 Mbps. While disappointingly slow when compared to the maximum possible Ethernet speed this board is capable of at 1000 Mbps, it was all that was available at the time for this project. An extensive amount of work went into modifying the hardware design for the soft Ethernet IPs and modifying the lwIP settings that ultimately configured the Ethernet interface that was used. On top of the slow Ethernet speeds, the modified C file that represented the server on the FPGA was problematic in various ways. Initially, the C file provided the infrastructure that was needed to establish communication via Ethernet between the PC and the FPGA. Upon further development, it became a barrier rather than an access point for our data. When a high volume of frame data was received by the server, it had a hard time keeping up with processing. This is because the MicroBlaze soft processor was responsible for running the lwIP stack to receive the frame data, converting all 103,936 pixels from every image to 32-bit IEEE-754 single-precision floating-point format, writing the data to BRAM, and starting on the next frame. To simply put it, the MicroBlaze core was over-exploited due to its ability to run C code.

The complication with the MicroBlaze processor led to a handful of issues that were experienced on the server-side of the FPGA. The most common theme was timing issues. The C script ran into timing issues when trying to receive frame data, convert pixels to write to BRAM, and communicating with the custom IPs. The clue that led to the discovery of the MicroBlaze processor being overused was that by adding simple print statements to the C script, that would print out to the connected serial port on the PC, seemed to resolve a variety of timing issues. This is because a standard print statement takes a long time. After all, the code needs to process and after it has to display the output to the user through the terminal. By adding these print statements to the C script during times of intensive processing, it essentially added a time delay to the program that allowed the MicroBlaze processor to catch up on its processing. However, by adding these time delays to one of the primary control scripts, the overall computation time for the image correlation was drastically increased.

Of the many issues faced with the lwIP, the biggest barrier that was faced was the process of converting the individual pixels in an image to the IEEE-754 single-precision floating-point format. Originally, this computationally intensive process was performed on the workstation PC with the Python client script. Python was used because it is a great language for quick scripting and testing of programs. With a handful of lines of code, the Python script was able to open a .tif image, iterate through each pixel in the image, and convert each one to the IEEE-754 single-precision floating-point format to send to the FPGA-based server. However, this code effectively turned a maximum three-digit number into a 32 digit number, which in turn is approximately 10.6 times more data to transfer to the FPGA per image. A temporary solution was to convert this 32-bit binary number into a decimal number. For example, if a pixel value is 100 (pixel values range between 0 and 255), it then needs to be normalized by dividing the pixel value by 255 which equals 0.392156863 in this case. This number, 0.392156863, when converted to the IEEE-754 single-precision floating-point format, is equal to the following 32-bit number 00111110110010001100100011001001.

Now, this binary number when converted to decimal equals 1,053,345,993. The difference here is that the 32-bit binary number takes an entire byte for each digit which means that it is equal to 32 bytes that are transmitted per pixel. When the 32-bit binary number is converted into a decimal number it only requires a maximum of 10 digits which is equal to 10 bytes of data to be transferred per pixel. When the decimal number of 1,053,345,993 is transferred to the FPGA and written into BRAM, which is composed of an array of 32-bit registers, it is automatically represented in its binary format which is the original 32-bit IEEE-754 single-precision floating-point format that is needed of 00111110110010001100100011001001. This method of transmitting a decimal number, that ultimately represents a 32-bit binary number, uses approximately 3.2 times more data per pixel than sending the original three-digit pixel value. While this is an increase in the amount of data sent, it relieves the MicroBlaze soft processor of having to convert each pixel value for each image it receives. This trade-off ultimately pushed more of the processing load onto the PC that transmits the image data but allowed the MicroBlaze processor to execute its remaining tasks flawlessly. Of course, this issue is bigger when realizing that the maximum transmission rate of the Ethernet cable was 56 Mbps.

When taking a step back from this method, it was obvious that the DICe hardware accelerator would not be much of an accelerator at all. While the image correlation time was improved with the hardware design that was programmed into the FPGAs fabric, the time required to pre-process images and transmit them to the KC705 proved to be too costly. At this point, the idea of using the USB port was never considered due to the high cost in engineering time it took to develop a working Ethernet port and because neither the Kintex-7 or the Virtex-7 had USB ports that were capable of the USB 2.0 or 3.0 standards. After detailing the list of problems that were faced during the development process in a formal report the proposed problem and solution were relatively simple. The FPGAs used for this period of development was unsuitable for the task that was presented and the solution was to take our existing design and target a new FPGA that could meet the requirements for this project.

Enter the purchase of the Zynq UltraScale+ MPSoC (ZCU104) FPGA from Xilinx. This FPGA came equipped with 1 Gbps Ethernet, USB 3.0, a quad-core ARM Cortex-A53 processor, a dual-core ARM Cortex-R5 real-time processor, and an ARM Mali-400 MP2 GPU. The ZCU104, when compared to the KC705, has twice as much BRAM, twice as many Digital Signal Processing (DSP) blocks, and 1.546 times as many logic cells. To simply put it, the ZCU104 FPGA blew the previously used FPGAs out of the water in terms of capability and available resources. Eager to put this new equipment to use, the original Verilog-based DICe hardware design was then minimally modified to target the new FPGA and programmed to do so. With so many new capabilities, the next few weeks were spent on researching and understanding exactly what this FPGA was capable of so that the DICe design could maximize this potential.

The first step was to evaluate the I/O ports in terms of data access to the images that needed to be processed. The ZCU104 immediately provided two ports, USB 3.0 and Gigabit Ethernet, both of which are capable of the data transmission this project required. Both options were explored extensively before the decision was made to develop a design for both. The reasoning behind this is that the ZCU104 FPGA didn't require extensive hardware designs in the FPGA fabric to access these I/O ports. The ZCU104 has Physical (PHY) IP on the board that gives the ARM processors direct access to these ports. This hardware-based approach was a significant advantage over the KC705 and VC707 where the I/O ports needed to be manually configured with soft IP within the hardware design. While the decision to use both I/O ports seems counterproductive given that USB 3.0 can transmit at speeds of 5 Gbps and the Gigabit Ethernet port is only capable of 1 Gbps, both options were valuable in terms of the users at Honeywell who oversaw the original statement of work. They reasoned that sometimes cameras are in-use and need to instantly offload images for processing via Ethernet to get results as quickly as possible. Other times, the cameras record over a long duration and the image data collected is stored within some memory medium, such as an external hard drive.

The second step was to determine what software intervention would be needed to access the memory on the FPGA so that it could successfully be written to and read from, and how to properly access the USB and Ethernet I/O ports. After a short amount of time, the answer was glaringly obvious that the solution was to use PetaLinux. PetaLinux is a tool provided by Xilinx to deploy Linux-based solutions on its FPGAs. The tool provides the infrastructure to deploy a command-line interface, application templates, device drivers, a variety of libraries, and a bootable system image on their FPGAs. Xilinx provides plenty of documentation on how to use this tool to get the ZCU104 FPGA to boot with a Linux-based kernel. The kernel of choice for this project was Ubuntu 18.04 LTS because of its familiarity.

In under a week, the SD card was prepped with the bootable image and Linux-kernel to run on the FPGA. This tool immediately provided the drivers to access the USB 3.0 and Gigabit Ethernet I/O ports. A problem that previously took months of work and effort to develop was setup and running in a fraction of the time. Testing the Ethernet port immediately yielded increased speeds up 950 Mbps, nearly 17 times more throughput when compared to the speeds achieved on the KC705. The kernel allowed for the installation of libraries and compilers to configure the ARM processor to compile and run C code. This enabled the deployment of the C control scripts locally on the FPGA whereas previously the control scripts were deployed on the workstation PC and had to transmit a variety of acknowledgment signals to the FPGA to proceed with processing. The quad-core ARM Cortex-A53 was leveraged to its fullest extent by creating a C script that utilized multiprocessing to pre-process frames before they needed to be written to BRAM. The struggle to access the FPGAs memory was significantly reduced by using a C function called mmap to locate and access available memory in the hardware design.

The PetaLinux tool provided every feature, library, and mechanism that was needed to fully utilize the hardware on the ZCU104 FPGA in a short amount of time. Low-level hardware designs that were previously needed to activate these features suddenly turned into a high-level software code that was far more familiar. Development and testing time was drastically reduced by the ability to locally compile and run C code on the FPGA without the need to resynthesize and reprogram the FPGA. With the functioning hardware design already programmed into the FPGA fabric and the ARM processor booting up the Ubuntu kernel that is accessed through the serial port with the command-line prompt, the time to test and deploy changes to the high-level control scripts running on the ARM processor were minuscule. The drivers needed to access the Ethernet and USB ports were enabled with the simple click of a button, literally. Ultimately, the move to the Zynq UltraScale+ MPSoC FPGA unlocked a plethora of features and abilities that were not previously available. In a short amount of time, more progress was accomplished with the deployment and testing of the DICe hardware design than was in months of work with the KC705 or VC707 FPGAs. The ZCU104 FPGA coupled with the PetaLinux tool provided a platform that has significantly improved the quality of the work presented in this paper.

6.2 Challenges

// TODO: Provide a list of challenges faced during this project: dense DICe source code and lack of documentation, Ethernet struggles, use of different FPGAs, setting up the Linux kernel, DICe hardware accelerator design challenges, floating-point numbers, low BRAM resources, etc.

Chapter 7

Conclusion

// TODO: Finish off the paper by discussing the objectives of the thesis, if they were achieved and how they were achieved. What was accomplished? Why is this work significant?

7.1 Future Work

This project has many reasons to explore future work due to how dense and complex the source code for DICe is. The first avenue to pursue would be to develop the DICe hardware accelerator such that it retains all of the features this GUI has. Based on the requirements this project was given, the DICe hardware accelerator that was developed for this project only runs one analysis mode that focuses on tracking. The DICe GUI has two other analysis modes that could be explored for the hardware-accelerated design. These analysis modes are subset-based full-field and global. Paired with these analysis modes are the additional optimization and correlation methods that come in simplex and robust. These analysis modes were not developed into the DICe hardware accelerator because they were unused by Honeywell that provided us with a statement of work for development. Their sole focus was on the implementation of the tracking analysis mode which is what this project uses.

There exist several features that DICe is capable of but that are not implemented in this work. The DICe GUI supports image obstructions while this implementation does not. This feature allows the user to select regions of the image that are obstructing the movement within the frame. For example, a frame may have a component such as a gear that spins, but a metal mounting rack for the gear could cover up a section of the frame that blocks a portion of the spinning gear. This feature allows the user to get more precise results. Another feature that could be further developed as future work would be the subsets. Currently, the design explained in this work only supports circular and square subset shapes. The DICe GUI allows the user to view an image and create their subset shape that is uniquely tailored to the content in the frame. The subsets could also be improved to support a greater quantity and a larger size. Currently, this design only allows for a max subset size of 41x41 pixels and a max number of 14 subsets, these were based on the given requirements.

One of the given requirements that were unmet for this project was the size of the images used. The DICe design in this work supports a frame of the size 448x232 pixels but needed to support a frame size of 896x464 pixels. This means that the DICe hardware accelerator only supports an image size that is one-fourth the size of the size specified in the requirements. The reason for this shortcoming was simply due to the lack of BRAM resources. The hardware design for this project utilizes nearly 100% of the available BRAM that ultimately restricted the image size to be stored locally on the FPGA to two 448x232 sized images. Two frames are needed to be stored on the FPGA because one frame is the reference frame and the other is the deformed frame. These two frames are needed so that the algorithms used in DICe can compare two images to compute the results.

A factor that could greatly improve the speed of the DICe hardware accelerator would be the use of 10 Gbps Ethernet speeds. The hardware design developed for this project utilizes 1 Gbps Ethernet speeds, although tests show this to be closer to 950 Mbps. Faster Ethernet speeds would allow the throughput to be greatly increased when transferring frame data from the PC to the FPGA for processing. This option was unavailable during the development cycle of this project due to the equipment used. The ZCU104 FPGA supports tri-speed Ethernet which allows for support of 10/100/1000 Mbps Ethernet speeds. The workstation PC used to transfer data to the FPGA contains an Intel Corporation Ethernet Connection I217-LM (rev 04) Ethernet controller that only supports 1 Gbps transfer rates. The use of this equipment led to a maximum Ethernet rate of 1 Gbps for this project.

With more development time available for this project, it would be possible to pipeline portions of the hardware design, both IPs and the code within them. As development continued for this project, portions of code were recognized and marked to be reviewed at a later date to explore the potential of pipelining the code. On a larger scale, some of the custom IPs that were developed have been marked as well due to their potential in pipelining with other IPs that currently run sequentially. When developing this project, the primary objective was to create a functioning DICe hardware accelerator that met the give requirements. While the possibility of a speedup through pipelining was recognized, it was never acted on due to the numerous other features and priorities that were needed for this application.

7.2 Conclusion

// TODO: Summarize the work that this thesis presents. Tell them what you told them!

Bibliography