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.

©2020 By Keaten Stokke
All Rights Reserved

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

Listings

Chapter 1

Introduction

Digital Image Correlation (DIC) is an optical method implemented by computers that receive image frames as input and measures the deformation on an object's surface without contact. Two-dimensional image correlation can provide data on the strain and displacement of a series of images in the X or Y direction while three-dimensional image correlation can provide the same data in the Z direction. Rather than individual pixels providing a reference point for measuring the change of an object from frame to frame, the neighboring pixels of that point are used to provide a reference window, or subset, that can provide far more accurate measurements for analysis. A wide variety of techniques for DIC exist, such as cross-correlation and deformation mapping, that can provide invaluable data on the provided input. DIC is becoming more common in everyday applications such as automotive use for self-driving cars to process their environment and avoid obstacles or industrial applications that analyze small components for abnormal wear, tear, and defections. The increase in modern applications that depend on DIC to properly function means an increase of the computational devices needed to perform DIC in a suitable time frame, especially with Real-Time Systems (RTS) where accurate results are expected in 30 milliseconds or less.

Field-Programmable Gate Arrays (FPGAs) have long been used for their flexibility to create reprogramable designs that target physical hardware for accelerating application performance. For high-level applications that have frequently changing parameters, Application-Specific Integrated Circuits (ASICs) are rendered useless by their inability to implement functional changes in their hardware designs. FPGAs provide an option to developers to create applications that can be modified and reprogrammed in the boards Configurable Logic Blocks (CLBs) to achieve near-true hardware acceleration without the expense of manufacturing and redeploying physical ASIC chips. The use of FPGAs have been around for decades, but are recently making a big come back due to the large volume of high-level applications that require both accelerated performance and configurable

designs. Many modern FPGAs contain components such as multi-core hard-processors, Graphics Processing Units (GPUs), and various I/O ports that can interact with the low-level hardware designs in the FPGAs fabric. This makes modern System-on-Chip (SOC) FPGAs more capable for processing-intensive applications than ever before.

The work presented in this paper combines the use of FPGAs for accelerated hardware performance and the Digital Image Correlation engine (DICe) program as the high-level application to leverage the performance boost. DICe is an open-source tool that intends to provide users with either a DIC module to implement in external applications or as a standalone analysis program. Currently, the DICe GUI only supports basic use cases for 2D and stereo DIC. Additional features can be enabled through the command-line interface to support use for additional DIC methods, such as trajectory tracking. When DICe is presented with a large volume of frames to process with multiple subsets, the time to complete DIC on the data set can be on the order of days for a standard workstation computer. This lengthy delay in producing results is unacceptable to many users of DICe, who all desire a means to produce results faster. A delay in producing and analyzing the results of DIC from DICe leads to a delay to solve the larger engineering problems that the application was meant to solve. The work in this paper is aimed at the development of a hardware accelerator for the DICe software by porting the design to the Verilog Hardware Description Language (HDL) to target FPGAs for the core of the processing of the application.

1.1 Motivation

TODO: Explain the reason 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 Related Works

TODO: This section will explore various related works that are similar to this project. How do these works relate and differ from the work in this paper?

2.2 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?

2.3 DICe

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

Chapter 3

Platforms

This section is dedicated to discussing the variety of software and hardware platforms that were required to complete this project. On the multiple workstation PCs in the lab, both the Windows 10 and Ubuntu 18.04 LTS operating systems were used for development in programming the FPGAs low-level hardware design and creating the high-level software to interact with it. The Windows 10 OS provided consistent development of the FPGAs hardware and software designs due to the provided Graphical User Interface (GUI) that was simpler to install and use when compared to a Linux-based OS. The Ubuntu 18.04 LTS OS was required to use the PetaLinux tool to implement and configure a Linux-based kernel on the ZCU104 FPGA. Three different variations of Xilinx FPGAs were used for application development and testing; the Virtex-7 (VC707), the Kintex-7 (KC705), and the Zynq UltraScale+ MPSoC (ZCU104). The PCs used during the development cycle of the DICe hardware accelerator varied in terms of hardware resources, which ranged from four-core to eight-core CPUs and 8 GB to 32 GB of RAM. For this project, the hardware contained in the PCs is insignificant because they were all capable of Gigabit Ethernet transmissions which is the only factor the PC plays in the results of this application.

When programming the FPGAs hardware designs, Vivado 2015.4 and Vivado 2018.3 software suites were utilized because Vivado is developed by Xilinx which manufactures the FPGAs that were used for this project. Xilinx provides the only software suite that is capable of interacting and programming the listed FPGAs. When programming the FPGAs initial software designs, the Vivado Software Development Kit (SDK) versions 2015.4 and 2018.3 were both used. The Vivado SDK is different from Vivado in that it is based on the Eclipse Integrated Development Environment (IDE) that is used to compile high-level C and C++ code. Before PetaLinux was used to implement software onto the FPGAs, the Vivado SDK was used to program high-level codes onto the FPGAs processors directly. Designing and testing the DICe control scripts and analyzing the default frames for DICe to process required a plethora of library packages and software

that were installed on both operating systems. Python, C++, and C were among the high-level software languages that were used to interact with the FPGAs processors and their low-level hardware designs. The design decisions for using all of these platforms, both hardware, and software, are explained in the sections below.

3.1 Hardware

Making a hardware accelerator for a software application will require hardware, but what kind of hardware to choose isn't as obvious. Generally speaking, to implement a hardware accelerator one would need to use either a powerful workstation PC, a Graphics Processing Unit (GPU), a High-Performance Computer (HPC), an FPGA, or an Application-Specific Integrated Circuit (ASIC). Each of these methods comes with pros and cons that can make it difficult to accelerate a software application. The method best suited for accelerating an application depends entirely on what the application is doing during processing. Workstation PCs are great for handling a wide range of frequently used software, such as word processors and internet browsers. GPUs benefit the user when graphics processing is a top priority to push images and video as fast as possible, such as with video editing and video games that drive monitors. In terms of expense, HPCs sit above PCs and GPUs for processing because they utilize multiple machines or components that are connected to act as a single system. These devices are a good option for solving intensive problems with large data sets that can be executed in parallel. True hardware acceleration starts with FPGAs due to their reconfigurable fabric that can implement a software application as logic gates. Logic gates are the foundation of modern computing hardware and an application that can exploit these building blocks has greater potential for faster processing than what software is capable of. ASICs are the pinnacle solution for hardware acceleration by creating a physical circuit to perform a dedicated task.

On one hand, powerful workstation PCs and HPCs qualify as hardware accelerators because they have more capable components internally than standard computers. They could have upgraded CPUs with multiple cores (beyond standard quad-core processors), upgraded RAM, extra GPUs, and in the case of HPCs multiple machines could be aggre-

gated together to tackle a single problem. On the other hand, these devices don't always meet the requirements to be considered as hardware accelerators because they typically still run the high-level software application on top of some Operating System (OS) that controls the hardware. This presents a barrier that prevents the software application from fully utilizing the available hardware resources. The software application can be modified to leverage the hardware of the system, such as multi-threading and multi-processing, but this will still require the OS to manage these processes. So rather than accelerating an application by targeting hardware, the application may be accelerated by more available hardware resources. HPCs are very expensive due to the vast amount of components required to create a single system and they are generally used for specialized processing tasks. Standard PCs, even with upgraded equipment from a Commercial-Off-The-Shelf (COTS) PC, cannot truly accelerate applications because they are designed with general-purpose processors that are designed to handle a wide range of tasks instead of a single specialized task. Neither of these methods offers a suitable solution when attempting to create a DICe hardware accelerator.

GPUs are specialized circuits that are designed to rapidly manipulate and alter memory to accelerate the creation of images that is intended as an output to a display device, such as a computer monitor or television. These components can be implemented in standard PCs and even HPCs to accelerate the processing of graphical-based data. However, their limitation is in the name in that their sole intention is for graphics processing that drives a display. This is useful if the application demands it, but they offer little if the application is out of this scope. While GPUs are suited for image processing applications, the DICe software application does not do image processing that needs to be driven to a display. The image processing algorithms in the DICe hardware accelerator focus on processing the image so that objects can be tracked from frame to frame with the output data being presented in a series of numerical values. If DICe took in video input and applied some sort of filter to the image to be displayed to the user, then this would be a different discussion. Because video output is not one of the features in the DICe application, the use of GPUs does not provide a solution for the development of a

DICe hardware accelerator.

Opposed to a general-purpose processor, ASICs are customized integrated circuit (IC) chips that are designed for a highly specific purpose. Today, ASICs are common in everyday devices that range from computers to key fobs. Common technical terms, such as microprocessors and flash memory, are all composed of ASIC designs that were all developed for a highly specific purpose. ASICs represent the purest form of hardware acceleration because the application designs that are developed for a specific function are directly manufactured into a physical integrated circuit. All software needs hardware to run on. By this logic, something that can be developed in software can always be developed in hardware. An application will almost always perform better when it is developed directly into hardware rather than software because there is less functional overhead, such as the required break down of high-level code to assembly language instructions to binary for a CPU to process. The benefits of using ASICs are widely known, as are the obstructions of developing them. ASIC development requires highly specialized equipment that can produce sub-micron level circuits and facilities to support the equipment via clean rooms. The process of developing ASICs, especially custom chips, comes with significant overhead in the engineering time to design the chip, in the manufacturing time to fabricate the chip, and in the time to test for chip verification. All of this overhead translates to cost. Lastly, once an ASIC has been produced and is in use, it cannot be modified or upgraded. This is where consumers fall victim to Moore's Law every year because, as the law states, every 18 months twice as many transistors can be packed onto a circuit. The result is the annual release of faster, smaller, and more energy-efficient devices. Many scenarios exist when the benefit and profit of developing an ASIC far outweigh the cost, such as the mass production of millions of processors for mobile devices and computers. However, the scenario of developing an ASIC to implement a DICe hardware accelerator is one where the costs to do so far exceed the benefits of production.

With all prior methods proposed for accelerating the DICe application being unsuitable, the last option of leveraging FPGAs is the premise of this paper. FPGAs are ICs that, by design, are to be configured after manufacturing; this is where the term

"field-programmable" derives from. FPGAs are far more flexible than ASICs in terms of development because they can be reprogrammed over and over to run other application designs. Just like with ASIC design, FPGAs can be configured by using a specialized computer language, known as a Hardware Description Language (HDL), to describe the behavior and structure of circuits. The two most common HDLs in use today are Verilog and VHDL; this project uses Verilog to implement all IPs in the hardware design. HDLs provide a tool for developers to perform functional simulations of the circuits design and synthesis to create a netlist of the design's description. The netlist specifies the physical electronic components to be used in the circuits design and how they will all be connected. After the netlist is generated via synthesis, the software tools that are used to program the FPGA will run a series of Place and Route (PAR) algorithms to determine the optimal place to position the components and route them together. In terms of cost, standard FPGAs are nearly equivalent to a COTS PC which is a perk for this project, but they require the engineering know-how skills to be able to program them. FPGAs differ from ASICs in that they contain programmable logic blocks and interconnects which is beneficial for prototyping and development, even for ASIC designers before they fabricate their chips. For this reason, FPGAs are typically used for low production designs whereas ASICs are used for high production designs. They are relatively low cost, provide flexibility for testing and prototyping through reprogramability, and get near true hardware speeds due to their Configurable Logic Blocks (CLBs). For all the reasons listed above, FPGAs were chosen as the hardware platform for the DICe hardware accelerator.

3.1.1 Xilinx Zynq UltraScale+ MPSoC FPGA

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?

Xilinx Virtex-7 and Kintex-7 FPGAs

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

I/O

The introduction of the ZCU104 FPGA unlocked a range of new features that were utilized for this project. Specifically, the hardware that this FPGA is equipped with enabled the implementation of Gigabit Ethernet, USB 3.0 access, and hard-processor control with minimal development. The hardware-based IP of the ZCU104s Input/Output (I/O) ports provided substantial ease of use for data transfer when compared to the VC707 and KC705, which required low-level software designs to access the I/O ports. After months of development, accessing data via Ethernet wasn't achievable on the VC707 FPGA. While data access was achievable on the KC705 FPGA, the maximum speed attained was a sluggish 56 Mbps. When programming the ZCU104 with an Ubuntu 18.04 LTS Linux-based kernel via SD card, and after modifying a few configuration files, the minimum attainable Ethernet speed was on average 950 Mbps which is nearly equivalent to the speeds of Gigabit Ethernet which is 1000 Mbps, or 1 Gbps. Also, the ability to access a connected USB 3.0 flash drive was as easy as literally checking a box in the PetaLinux configuration settings within the terminal prompt on the PC. This opportunity allowed the exploration of both an Ethernet-based and USB-based data access method for the DICe hardware accelerator. The VC707 and KC705 FPGAs do not come equipped with a USB 3.0 port which prevented the option of sufficient USB data access. While both the VC707 and KC705 FPGAs come equipped with an SD card to boot a PetaLinux-based kernel, this was an undesirable approach due to the overhead incurred by the MicroBlaze soft-processor that ran the kernel.

3.2 Software

This section will provide information on all of the major software applications that were used to develop the DICe hardware accelerator. A few minor software applications were used while developing this project that will be mentioned here briefly, but not extensively due to their minimal use and lack of significance for the DICe design. Notepad++ is a free and open-sourced text editor program that was used on the PCs to write the various high-level software programs that were needed to interface with the FPGA and the files on the PC. All Python, C, and C++ scripts that were developed

for this project were written in Notepad++ and compiled using the PCs terminal with the proper libraries installed. Wireshark is a network-protocol analyzer program that was used when testing the Ethernet transmissions between the PC and the FPGA. This program provided a GUI to monitor and trace packets as data from the PC was sent and received over Ethernet to and from the FPGA. GParted is a free partition editor that was used to partition and configure the SD card for the ZCU104 FPGA so that it could boot the Linux-based kernel. Lastly, PuTTY is a free SSH and Telnet program that was used to connect to the FPGAs serial ports to provide a terminal-like interface that assisted with debugging the high-level software that ran on the FPGAs processors.

The major software applications used for the development of this project, and the focus of this section, were the ones needed to create application designs that could target the FPGAs. Throughout the development of this project, each FPGA that was used was manufactured by Xilinx. To interface with the FPGAs processors via software designs and the FPGAs fabric via hardware designs, the Vivado software suite developed by Xilinx was used because these applications are made specifically for development on their FPGAs. Xilinx provides Vivado to users to develop low-level hardware designs that are meant to be programmed onto the FPGAs fabric. Two iterations of the Vivado suite, 2015.4 and 2018.3, were used for this project to interact with the different FPGAs used. To create high-level software designs that target the FPGAs processors, the Xilinx-made Vivado SDK and PetaLinux SDK tools were used. These programs provide the necessary tools to create high-level software applications that are then targeted on the FPGAs processors. Each of these major software applications, along with the control scripts, will be explained further in the sections below.

3.2.1 Vivado 2018.3

Developed by Xilinx, the Vivado Design Suite is used for synthesis and analysis of HDL designs. Vivado is classified as an IDE that allows users to develop low-level hardware designs that target Xilinx FPGAs. This suite comes with a plethora of Xilinx-developed IP that can be integrated into designs to reduce development time. Vivado also enables users to develop their own HDL-based IP for application customization. Hardware designs

in Vivado can be created as a series of HDL files that are linked together or by using the built-in block diagram GUI which enables users to drop in IP blocks and manually connect signals together. When a design is completed, Vivado can generate a bitstream file that is used to program the FPGA with the design. When the design runs on the FPGA, a hardware manager tab is available to users to monitor the FPGAs temperature during processing and a live view of signal values if a VIO or logic analyzer is included in the design. Vivado was used as the primary tool for developing the design for the DICe hardware accelerator. The software suite provides all of the necessary tools and features to create hardware designs, test hardware designs by running simulations, synthesize hardware designs for specific FPGA hardware, program the developed designs onto the FPGAs for execution, and providing an interface to debug Designs Under Test (DUT).

The tool provides design validation which enables the user to verify that the created hardware design is correctly configured and free of any major design flaws before simulation or synthesis. Users can create testbenches for their designs that allow them to simulate the functionality of their applications. A testbench is an HDL-based file that essentially wraps around the hardware design and provides it with a series of inputs that will be executed and outputted to the user when a simulation is run. Running simulations within Vivado is an important tool for users to be able to test the correctness and functionality of their design before synthesis. Simulation, however, is just a tool for functional testing of a design and it does not guarantee that a design will pass synthesis. Synthesis is perhaps the most important feature that Vivado provides. The synthesis process will take the users' design, either in the form of HDL code or a schematic, and turn it into a netlist. This step is critical because the netlist is the file that is responsible for mapping and connecting logic gates and flip-flops together within the FPGAs fabric. In simpler terms, synthesis is responsible for transforming a software design into the necessary hardware components to physically represent the application. Place-and-Route (PAR) is the step that occurs after synthesis and uses algorithms to determine the optimal way of placing the components defined in the netlist within the FPGAs fabric and routing them all together.

When coupled together, the Vivado Design Suite and the Xilinx FPGAs used provided the foundation for the development of the DICe hardware accelerator. When the development of this project first started, Vivado 2015.4 was used to create the hardware designs and program the VC707 and KC705 FPGAs. This iteration of the Vivado software provided all of the required infrastructures to interface with the 7 Series FPGAs. When the development of this project started in early 2018, upgrading to a newer iteration of the Vivado Design Suite was not necessary because Vivado 2015.4 provided all of the needed capabilities for the available FPGA hardware. However, this changed in mid-2019 when the ZCU104 FPGA was purchased for the continued development of this project. Vivado 2015.4 was incapable of interfacing with the newer Zynq UltraScale+ MPSoC FPGAs which prompted the upgrade to Vivado 2018.3. The key differences between Vivado 2015.4 and Vivado 2018.3 are support for a wider range of newer FPGAs, an upgraded GUI, and upgraded Xilinx-developed IP. In terms of the hardware design for the DICe application, the upgrade to the newer Vivado Design Suite only changed the processor used from a MicroBlaze soft-processor to the quad-core ARM Cortex-A53 processor. The only inconvenience caused by upgrading to Vivado 2018.3 was recreating the original hardware design that was developed in Vivado 2015.4. Many behind the scenes changes that were made to Vivado prevented the direct porting of an older project design to the newer software.

Starting with the project creation, Vivado enables the user to choose a target FPGA and HDL for the hardware design. This project ended with targeting the ZCU104 FPGA and Verilog as the HDL. While differences do exist between the VHDL and Verilog HDLs, Verilog was used for this project due to the familiarity and the syntax of the language. There was no ultimate engineering design decision that favored the use of one over the other, it just came down to personal preference. When creating the hardware design of the DICe application, the block diagram GUI provided an easy way of implementing IP developed by Xilinx into the design as well as adding in custom developed IP blocks. The block diagram GUI also provided a clear visual flow of the applications IPs, signals, and how they all connected together, which was very beneficial as the design grew in size.

The bulk of the IP created for this project revolves around Vivado providing the ability for users to create and package custom IP. This feature is what enabled the porting of the C++-based DICe algorithms to Verilog and ultimately to target the hardware on the FPGA. Individual unit tests and simulations were performed on each custom-built IP by developing testbenches tailored to the functionality of each IP block.

Early on in the development of the DICe hardware design, the most common design error was the failure for certain functions in the custom IPs to meet the timing requirements required by the synthesis process. One of the many jobs that synthesis performs is to verify that the hardware design meets the timing requirements that are set by the clock speed on the FPGA and all of the connected hardware components. When the hardware design is transformed into a netlist, it represents the application in terms of physical logic gates and flip-flops that exist in the FPGAs fabric. When the netlist is targeting the FPGAs hardware, it verifies that when an output signal is generated it can transfer the data to the input of the next component in the required amount of time that is physically required to send the data. This concept in static timing analysis is known as setup and hold slack which is defined as the difference between the data required time and the data arrival time. This concept is what led to developing all of the custom IPs with a Finite-State Machine (FSM) architecture. Each custom IP built for this project implements an FSM that is dependent on the systems clock and the defined state variable to transition from one state to the next. This architecture allows for a function to be broken into multiple states that each requires one clock cycle to execute. The benefit of this is that when a static timing analysis report is generated and a timing fault is detected, it can be traced back to a specific state within an IP block. Once the source of the timing fault is found, it can be resolved by providing it with more states to complete its execution.

All of the features explained above detail why the Vivado Design Suite was such a significant tool for the development of the DICe hardware accelerator. The Vivado 2018.3 program provided the interface and tools required to create an application hardware design and implement it in the fabric of a Xilinx-based FPGA. The core algorithms and

image processing functions of the DICe software were analyzed and reprogrammed using the Verilog HDL so that Vivado could synthesis them into a netlist for the FPGA to run. While Vivado 2018.3 was used extensively to create the DICe hardware design, it wasn't responsible for creating the high-level software that runs on the FPGAs processors. The DICe hardware design was built around the core features of the DICe software, but it isn't capable of executing the image correlation on its own. For the hardware design to be able to perform image correlation, it requires parameter data to specify the bounds of the images and subsets it will operate on and the frames that are to be processed. For image correlation to start on the FPGA, all of this required data needs to be present within the FPGAs BRAM. This is where the Vivado 2018.3 SDK tool provided assistance.

Vivado 2018.3 SDK

The DICe hardware design was developed to implement the core image correlation algorithms that are utilized within the DICe software. The hardware design has no means of retrieving the data it requires to start processing. For this project, the Vivado 2018.3 SDK tool was used to create high-level software designs that run on the FPGAs processors and interact with the hardware design in the FPGAs fabric. Its these software designs that are responsible for retrieving the parameter and frame data from the FPGAs I/O ports and writing the data to BRAM. The SDK provided a GUI that enabled the development of an application directly onto the MicroBlaze soft-processor used in the VC707 and KC705 FPGAs and the quad-core ARM Cortex-A53 processor in the ZCU104 FPGA.

The Vivado 2018.3 SDK provides a development environment for high-level software applications. The tool is based on the open-sourced Eclipse IDE and can be installed independently of the Vivado Design Suite. It does more than the standard Eclipse IDE in that it can import Vivado-generated hardware designs, create and configure board support packages (BSPs), supports single-processor and multi-processor development for FPGA-based software applications, and comes with off-the-shelf software references designs, like the LightWeight IP (lwIP) application, that can be used to test the applications hardware and software functionality. The SDK is the first application IDE to deliver

true homogeneous and heterogeneous multi-processor design, debug, and performance analysis. The primary feature that the Vivado SDK provided for this project is the compilers that optimize C and C++ code and generate assembly code from them. These compilers are responsible for enabling high-level software designs to be targeted on the FPGAs processors.

The only comparable application to the Vivado SDK is the Vivado High-Level Synthesis (HLS) program. This software is used to create IP by enabling C and C++ code to be directly targeted into the Xilinx FPGAs fabric without the need to manually create a Register Transfer Level (RTL) design. This means that the HLS tool is capable of generating low-level hardware designs from C and C++ code, but it is incapable of generating high-level software applications for the FPGA processors. The use of HLS for the DICe hardware accelerator was explored as an option to accelerate the development of the hardware design but was ruled out due to the custom nature of the DICe GUI. The original DICe application is composed of 98 C++ files, each with custom functions tailored for the image correlation process. Because of the complexity of the DICe source code and the custom functions, classes, and types created for the application, HLS was determined to be unsuited for converting the entire application to a hardware-based design. Lastly, because HLS is incapable of programming high-level software designs on the FPGAs processors, the use of this tool was ruled out for this project.

For the initial design of the DICe hardware accelerator, the Vivado SDK was used to implement the provided lwIP reference design on the MicroBlaze soft-processor. The LightWeight IP is an open-source TCP/IP stack that is designed to minimize resource usage for embedded systems. The reference design provided by the Vivado SDK was a simple echo-server application. When programmed onto the FPGA, and with the FPGA connected to the PC via Ethernet, the application would simply echo back any data that was sent to the processor from the PCs command line. This simple client-server application was generated in C and provided a basic template to enable Ethernet transmission between the FPGA and the PC. The lwIP echo-server was then heavily modified to suit the needs of the DICe hardware accelerator. A control script running

on the PC would act as the client that would initialize the connection with the FPGA, transmit parameter and frame data as needed, and then receive the results from the FPGA to format it into a text file. While the lwIP echo-server application provided a sound starting place for Ethernet-based I/O, it came with more challenges than it was worth. After numerous tweaks to the BSP and configuration files for the lwIP application, the max transmission rate achieved was only 56 Mbps. To make matters worse, there was no way to safely disable the "echo" feature of the application without it compromising the rest of the design. This meant that for as much data that was transferred to the FPGA, the same amount of data was echoed back to the PC which had to be ignored. More details can be provided on the lwIP echo server in Section 6.1.

3.2.2 PetaLinux

PetaLinux is an embedded Linux Software Development Kit (SDK) that is developed by Xilinx to target FPGA-based System-on-Chip (SoC) designs. This SDK tool contains everything necessary to build, develop, test and deploy embedded Linux systems. The PetaLinux tool is composed of three key elements: pre-configured binary bootable images, a fully customizable Linux kernel for the Xilinx FPGAs, and the PetaLinux SDK which provides the utilities and tools to automate the daunting tasks of configuration, build, and deployment of the software application. The PetaLinux tools enable the user to deploy a Linux-based system on their FPGA platform that provides a bootable system image builder, a command-line interface, device drivers, libraries with templates, GCC tools, and various debug agents. Although using PetaLinux to deploy Linux-based systems on MicroBlaze-based FPGAs is possible, it was not a feasible solution when the VC707 and KC705 FPGAs were used for development; this is discussed in more detail in Section 6.1.

Leveraging the PetaLinux SDK for the high-level software development of this project was first considered with the addition of the Zynq UltraScale+ MPSoC FPGA. This FPGA was a far more capable device when compared to the previous FPGAs used for this project. The introduction of the quad-core ARM Cortex-A53 processor on the ZCU104 was too valuable of a resource to leave unused. It has far more processing power than the MicroBlaze soft-processor and it could be used to deploy the control scripts that

handle the FPGAs I/O ports, image pre-processing, and controlling the DICe hardware design locally. The ZCU104 FPGA provides PHY IP that controls the I/O ports, such as Ethernet and USB, so using PetaLinux to assist in interfacing with the I/O was not a requirement. However, as Section 6.1 details, working with the FPGAs I/O ports through the Vivado SDK and lwIP echo-server proved to be a barrier to unlocking the full 1 Gbps Ethernet speeds that the FPGA is capable of. In addition to that, there were no reference designs or applications that supported data access via the USB 3.0 port.

Using the PetaLinux SDK tools required a PC running a Linux-based OS. A spare PC in the lab was completely wiped of all contents and reformatted to run the Ubuntu 18.04 LTS Linux-based OS. This OS was chosen because it was found to be a common OS to use for Vivado and PetaLinux development and it is probably the most well-known Linux distribution. Upon further research, it was discovered that the Ubuntu 18.04 LTS kernel was a popular option for deploying on the Zynq UltraScale+ MPSoC series of FPGAs. So with that, the Ubuntu Linux distribution was selected for the OS of the PC and for the FPGAs kernel due to the many resources that were available for this type of development. Once the OS was installed on the PC, the installation of the PetaLinux tools required the installation of dozens of library packages. Once this step was completed, the real development with PetaLinux started.

First, a program called GParted was installed on the PC that provided a GUI for partitioning memory drives connected to the PC. To deploy the Ubuntu kernel on the FPGAs processors, it requires that a few of the configuration switches physically located on the top of the FPGA must be properly set to prompt the FPGA to start its boot-up sequence from the SD card slow. The SD card is to be partitioned into two sections labeled BOOT and ROOTFS (root file system). The BOOT partition requires a size of at least 500 MB, a FAT32 file format, and the setting of the boot and iba flags. The ROOTFS partition requires a size of at least 1 GB+ and requires the EXT4 file format. Starting with the ROOTFS partition, a tarball file that provides the minimal Ubuntu 18.04 kernel for ARM-based processors was downloaded from an online website at: https://rcn-ee.com/rootfs/eewiki/minfs/ubuntu-18.04.3-minimal-armhf-2019-11-23.tar.xz. Once this

tarball file was extracted, it was then copied to the ROOTFS partition. The extracted tarball file contains the core Ubuntu 18.04 LTS kernel that the FPGA will run on the quad-core ARM Cortex-A53 processor. It contains nearly identical directories to the root directory in the Ubuntu OS such as: bin, boot, dev, home, lib, media, sys, usr, and var. Next, the ROOTFS is granted root at 755 permissions via the command line. At this point, this partition is completed and is ready to be deployed.

The BOOT partition requires a lot more work to configure correctly when compared to the ROOTFS partition. The BOOT partition is responsible for providing the FPGA with the required information to properly boot the contents contained in the ROOTFS partition on the FPGAs processor and for programming the hardware design into the FPGAs fabric. The first step in configuring this partition is to create a PetaLinux project with the following command to target the FPGA in use: petalinux-create –type project –template zynqMP –name PROJECT-NAME. This creates a folder directory that will contain the PetaLinux files required to configure and build the boot image. Assuming that the hardware design is already completed by this point, the next step is to export the hardware designs hardware description file (.hdf) and bitstream file (.bit) from Vivado 2018.3 to the PetaLinux project directory. Afterward, the following command is used to configure the hardware design into the boot image: petalinux-config –get-hw-description. The next two commands are entered after to properly package the PetaLinux project with the hardware design and the bitstream: petalinux-package –boot –format BIN –fsbl images/linux/zynqmp_fsbl.elf –u-boot images/linux/u-boot.elf –pmufw images/linux/pmufw.elf –fpga images/linux/*.bit –force, petalinux-package –boot –fpga bitstream.bit –u-boot –force. Next, the boot image needs to configured to implement support to boot from the SD card: petalinux-config. This command pulls up a menu within the terminal that allows the user to select on Image Packaging Configuration, then Root filesystem type where an option to select the SD card is to be checked.

For the standard deployment of a PetaLinux-based project on the FPGA, the only remaining step is to run the following command: petalinux-build. However, because this project requires the use of the Ethernet and USB 3.0 I/O ports, the boot image must be

further configured to add the driver support for this hardware. The following command is executed to configure the kernel properties of the boot image: petalinux-config -c kernel. A menu is then displayed to the user in the terminal to add device driver support; the following options were checked for the successful installation and support of the I/O devices drivers: support for Host-side USB, EHCI HCD (USB 3.0) support, USB Mass Storage support, ChipIdea Highspeed Dual Role Controller, ChipIdea host controller, and Generic ULPI Transceiver Driver. All of these adding settings need to be saved before closing the menu. Lastly, a device tree file labeled as "system-user.dtsi" needs to be modified to add support for the I/O ports. The code for this can be examined below in Listing 3.1.

```
/include/ "system-conf.dtsi"
/ {
  model = "ZynqMP ZCU104 RevC";
  compatible = "xlnx,zynqmp-zcu104-revC", "xlnx,zynqmp-zcu104", "xlnx,
   zynqmp";
  aliases{
    ethernet0 = &gem3;
    usb0 = &usb0;
  };
};
&sdhci1 {
    status = "okay";
    xlnx,has-cd = <0x1>;
    xlnx,has-power = <0x0>;
    xlnx,has-wp = <0x1>;
    disable-wp;
    no-1-8-v;
};
&gem3 {
    status = "okay";
    phy-handle = <&phy0>;
    phy-mode = "rgmii-id";
    phy0: ethernet-phy@c {
```

```
        reg = <0xc>;
        ti,rx-internal-delay = <0x8>;
        ti,tx-internal-delay = <0xa>;
        ti,fifo-depth = <0x1>;
        ti,dp83867-rxctrl-strap-quirk;
  };
};
&usb0 {
    status = "okay";
};
&dwc3_0 {
    status = "okay";
    dr_mode = "host";
    snps,usb3_lpm_capable;
    phy-names = "usb3-phy";
    maximum-speed = "super-speed";
};
```

Listing 3.1: Configuration of the FPGAs Ethernet and USB device drivers in the system-user.dtsi file

At this point, the petalinux-build command can be executed which creates the final boot image files that are required to be in the BOOT partition of the SD card. The BOOT.BIN and image.ub files are to be copied directly over to the BOOT partition of the SD card. The SD card can now be ejected, to safely remove it from the computer, and inserted into the SD card slot on the FPGA. Picocom is a program that was installed on the host PC to monitor serial port connections. When the FPGA is plugged into the PC with the Micro-USB to USB wire and turned on, the boot-up sequence can be shown through the serial port. This program also allows the user to interact with the Linux-based FPGA system through a command-line interface. Using the command-line interface enables the user to interact with the file system on the FPGA just like they would through the terminal in the desktop-based OS. Once an active Ethernet cable is plugged into the FPGA it is possible to download and install libraries, packages, and programs on the FPGA through the command-line interface.

Using the PetaLinux tool for this project was significant because it provided the means to interact with the FPGAs processors and I/O ports with no modifications to the underlying hardware design. When the VC707 and KC705 were initially used for this project, they each required extensive hardware-based designs that acted as the device drivers. These designs were complicated, sparsely documented, and only achieved on the KC705 FPGA. The process to develop Ethernet communication between the host PC and the KC705 took more than two months of persistent work to enable and was only capable of 56 Mbps speeds. In under a week, the well documented PetaLinux tools were used to create a software design that targeted the FPGAs processor and I/O ports with complete success. The full functionality of the USB 3.0 and Gigabit Ethernet ports was unlocked through the device drivers provided by the PetaLinux SDK. Access to the quad-core ARM Cortex-A53 processor is possible through multiprocessor programming in C directly on the FPGA through the command-line interface and a few installed packages to compile the high-level code. The potential to use the ZCU104s processor is discussed in Section 3.2.3 below with the development of the DICe control scripts.

3.2.3 DICe Control Scripts

The DICe hardware accelerator is divided into two sections which are the hardware design that runs on the FPGAs fabric and the software design that runs on the FPGAs processors and host PC (if needed). The hardware design will be elaborated in Section 4 below. This section is dedicated to explaining the software design that runs on the FPGAs processor and the client script that runs on the host PC. The host script on the PC is only necessary when the Ethernet-based DICe design is running to transmit data to the FPGA. When the USB-based DICe design is running, only the software on the FPGAs processor is needed because all of the data is accessible locally through the USB 3.0 port on the FPGA.

Python was used first during development because it is a great language for fast prototyping of software applications. Using Python enabled quick development and testing of the client-server interaction between the host PC and the FPGA. Once the core functionality of the script was in place, the host PC script was then migrated to C because of

its faster performance. Python was slower when compared to C at opening the images, converting the images to the IEEE-754 format (which has since moved to the FPGA processor), and transmitting the images; this will be shown in the results in Section 5. The move to the C language provided to be useful when considering the development of the USB-based design that would execute on the FPGAs processor. C provided the means to directly access memory within the FPGA that was associated with an address. Developing a client-server network interface in C is trivial and so is interfacing with the file system on the OS to retrieve data. The only difficult task of developing in C was the required library called "libtiff" to open .TIF images. Installing and configuring this library to work properly for this project was a challenge. However, once this was set up and working to open and process images, the development cycle proceeded forward. C proved to be the single language for continued software development in that it provided all the features needed to create both the client and server control scripts.

C Client Script

Starting with the Ethernet-based DICe design, the client program is initiated by running the C script through the command-line interface on the host PC. When the program starts, a clock is defined that acts as a timestamp to keep track of the total execution time for the software application. File paths are defined as character array variables to provide the correct file directory to the data on the host PC. A simple while-loop executes that counts all of the images located in the directory that holds the data to get a total number of frames to be processed. This number is useful because it will allow the control script to be aware of how many images need to be transmitted to the FPGA and when to expect a series of computed results in return. The C control script then opens up the Subsets.txt file that contains all of the parameter data for the image correlation such as image height, image width, and all of the various subset parameters. At this point, the server control script on the FPGA needs to be executed so that it can connect with the host PC.

The host PC script then attempts to connect to the FPGAs server with the IP address 192.168.1.10 and a port number of 7. The sys/socket library in C is used to provide the

network interface through the use of sockets. The connection will automatically timeout and terminate the rest of the processing of the script if a connection is not established within 15 seconds. If the connection is made, a print statement is displayed to the user to notify them that a successful connection between the PC client and the FPGA server has been established. The connection itself uses the Transmission Control Protocol (TCP) method to send packets. TCP is used over the User Datagram Protocol (UDP) method of connection because, while TCP is technically slower, it comes with the assurance that packets are delivered to the receiver. TCP implements a "back and forth" communication between the client and server which includes acknowledgments of received packets and re-transmissions of packets that are lost. Using UDP would be faster in transmitting all of the data for this application, but would instantly fail if just one-pixel value didn't make it to the server. Even if the server was aware that a given number of pixels were dropped during transmission, it would be a tedious task of isolating which ones would need to be recovered. Due to the simple nature of socket programming, C code for the functions used will not be provided so that space is saved for more important functions.

The first thing the host PC will transmit to the FPGA is the entirety of the Subsets.txt file so that all of the parameters can be loaded into the BRAMs. The Subsets.txt file is opened from its file directory and a loop will go through each line to read the data into a local variable. The Subsets.txt file is responsible for holding all the parameter data that the image correlation process needs to operate. The file contains a unique value on each line that represents these parameters, which are all subject to change. The first value in the file is the width of the image, which is 232. The second value is the image height which is 448. The third value is the total number of pixels per image which is 103,936. The fourth value is the total number of bits, which is 32-bits multiplied by the number of pixels which is 3,325,952. The fifth value determines the total number of subsets that are defined by the user. The sixth line denotes the optimization method to be used during the image correlation, in this case, the value is 0 which represents the gradient-based optimization method. The optimization method chosen can be performed in either Fast-mode which iterates over the provided subsets 25 times, or Robust-mode which

iterates 100 times over the subsets; our DICe method implements only the Fast-mode. The seventh line denotes the correlation routine to be used during image correlation, this value is also a 0 which represents the Tracking routine. After this, each iteration of five lines represents a single subset. The first line in a subset definition represents the shape of the subset. If the value in this line is a 0, then a circular subset has been defined. If the value of this line is a 1, then it represents a square subset. The second line in the subset definition is the X-center point of the subset and the third line is the Y-center point. These pixel values determine where the center of the subset will reside within the entire frame. For a circular subset, the fourth line defines the radius of the subset and the fifth line is the value of the radius squared. For a square subset, the fourth line defines the size of the subset in pixels and the fifth line defines the subset half-size using the floor-rounding method.

The while-loop in the C script traverses through each line of the Subsets.txt file until it reaches the end. The specifications provided for the DICe hardware accelerator only required a maximum number of 14 subsets to be defined, so both the client and server C scripts have variables that account for a total of 70 subset variables. When the end of the file has been reached and all of the parameters have been saved into local char array variables, the client script will then sequentially send all of the data from the Subsets.txt file to the FPGA over the Ethernet connection. After, the client script will send a string value of "SUBSETS DONE" that notifies the server script on the FPGA to write all of the parameter values to BRAM and get ready to accept image data. The next step in processing for the host PC client script is to transmit the images to the FPGA. The images used for this project are in a Tagged Image File (.TIF) file format which is used for storing high-quality graphics. Often, this format is used for storing images with many colors, but the focus of this project is only on grayscale images where the value of the Red, Green, and Blue (RGB) spectrum of each pixel is the same. The .TIF image format is rather uncommon when compared to normal image types such as .PNG or .JPEG. However, it is used for this project because when high-speed cameras, like the Phantom VEO 1310, are used they produce a video output in a .CINE format that is then split into

individual .TIF frames. The first image is opened from the file directory that contains the frame data and a for-loop is used to iterate through each pixel in the frame.

When each pixel has been retrieved, a comma is appended to the end of the number so that each pixel number isn't appended together. The pixel values of images range from 0 to 255 in the RGB spectrum. Because a pixel value can have a length of one, two, or three, the use of a delimiter is needed to be able to keep keep the individual pixel values separated. Because all of the images are in grayscale, only the first number from the red spectrum is pulled because the number is identical to those in the green and blue spectrums. This project operates on images of size 448x232 pixels which means that a total of 103,936-pixel values are to be transmitted from the PC to the FPGA for each frame. Each of these pixel values, with a comma appended to the end of them, is stored in a variable of type char array (because of the commas) and is then sent to the FPGA over Ethernet by using the "send" function from the sockets library. This will send all of the pixels over to the FPGA in as big of packets as possible. Because the actual image of frames changes, the exact size in terms of bytes varies per frame, but the average frame is approximately 155,000 bytes. Immediately after the "send" function is called on the pixel data, a subsequent "send" function is called that sends the string "END FRAME". This notifies the server control script on the FPGA that an entire frame of data has finished transmitting and that the client control script is ready to transmit the next frame. When the server control script is ready for another frame, it will transmit the string "SEND" to the host PC script to notify it to send another frame. This process continues on for as many frames that need to be processed. It is important to note here that the server script on the FPGA is responsible for converting each of the received pixel values to the IEEE-754 single-precision floating-point format and writing these values to BRAM.

When the last frame has been transmitted to the FPGA from the host PC, the C control script will send the string "LAST" to notify the FPGA server script that the last frame has been sent. When the DICe hardware design has finished image correlation on the last set of images, it sends the string "RESULTS" to the script on the host PC to notify it that it is ready to send computed results from the image correlation. The server

script will begin to read all of the results from BRAM five which is connected to AXI BRAM Controller two. It will transmit all of the results over Ethernet to the script on the host PC script where they will be formatted. The results in BRAM five are represented as 32-bit values in the IEEE-754 single-precision floating-point format. When these values are read by the C server script, the are automatically interpreted as decimal values that are then appended together with commas as a delimiter. The client script then creates as many text files, labeled as "DICe_Solutions_#.txt", as there are subsets to write the solutions too, where the subset number represents the # symbol in the file name. Upon receiving the data, the C script will take the data, split the numbers up based on the commas, and convert the decimal number (based on the IEEE-754 single-precision floating-point format) to scientific notation to be written in a human-readable format in the solution files. The script then ends and reports the total execution time to the user.

C Server Script

The Ethernet-based DICe design requires a client script to be running on the host PC to transmit data while the server script on the FPGA receives the data and operates on it. When running the USB-based DICe design, only the server script running on the FPGA is required because it accesses the data it needs from a USB 3.0 port locally on the FPGA. The only difference between the Ethernet-based server script and the USB-based server script is how the parameter and frame data is retrieved, and how the results are saved. With the USB-based DICe design, it is required that the proper driver is installed within the FPGAs boot image so that the attached USB memory drive can be accessed from the command-terminal interface on the FPGA running the Ubuntu 18.04 LTS kernel. The process for accessing the data on the USB in C can be shown below in Listing 3.2. The Subsets.txt file is iterated through in the same manner as described before in the client control script and the values are saved to local variables and written to BRAM using the "mmap" function in C.

```
// File paths to the data on the USB drive
char imagePath[] = "/media/usbstick/Images/";
char subsetPath[] = "/media/usbstick/Subsets/Subsets.txt";
char resultsPath[] = "/media/usbstick/Results/Results.txt";
```

```
5
6  // Verifies that the memory on the USB drive can be accessed
7  int mem_fd = open("/dev/mem", O_RDWR|O_SYNC|O_CLOEXEC);
8  if(mem_fd == -1){
9    printf("Unable to open /dev/mem");
10   return 0;
11 }
12
13 // Opens the Subsets.txt file in read mode to access the data
14 FILE *fptr;
15 if((fptr = fopen(subsetPath,"r")) == NULL){
16   printf("Error! Could not open the file: Subsets.txt");
17   exit(1);
18 }
```

Listing 3.2: Reading from USB memory in C

The images located on the USB drive are accessed similarly. From this point on, the C code that defines the USB-based server and the Ethernet-based server scripts are the same. The only distinction between the two is the accessing of data locally via USB drive or receiving the data from a TCP connection. When an image has been retrieved from the USB drive, it is necessary to convert each of the pixels in the image to the IEEE-754 single-precision floating-point format before writing to BRAM. The C code used to convert the pixel values into the proper IEEE-754 format was custom developed and can be shown below in Listing 3.3. The process to convert each individual pixel and write them into BRAM can be shown further below in Listing 3.4. Once the first two full frames are written to BRAM, the DICe hardware design has all of the data it requires to begin image correlation. This process is triggered by the C script writing to the AXI slave registers, using the same "mmap" function as before, of the Parameters IP, the Gradients' IP, and the Interface IP. After this, image correlation will begin within the FPGAs hardware design, which is discussed below in Chapter 4. The server control script will then continuously read from the AXI slave register of the Gamma IP and write new frames to the corresponding BRAM when it is signaled to do so. This process will

continue until the final frames have been written into BRAM by the server control script.

```
// Struct to define the sections of the IEEE-754 value
typedef union{
    float f;
  struct{
    unsigned int mantissa : 23;
    unsigned int exponent : 8;
    unsigned int sign : 1;
  } raw;
} myfloat;
// Function to get the binary representation of the number
void printBinary(int n, int i, int high_index){
  int k;
  for(k = i - 1; k >= 0; k--){
    if((n >> k) & 1){
      pixel_ieee[high_index] = '1';
    }else{
      pixel_ieee[high_index] = '0';
    }
    high_index++;
  }
}
// Converts the decimal number to IEEE-754 format
int Dec_to_IEEE(float input){
    myfloat var;
    var.f = input;
    pixel_ieee[0] = '0'+ var.raw.sign;
    printBinary(var.raw.exponent, 8, 1);
    printBinary(var.raw.mantissa, 23, 9);
    pixel_ieee[32] = 0;
return 0;
}
```

Listing 3.3: Converting a decimal value to the IEEE-754 single-precision floating-point format in C

```
// Read from the image file...
if(TIFFReadRGBAImage(tif, w, h, raster, 0)){
    // Iterate through the total number of pixels
    for(int i = 0; i < npixels; i++){
        // Retrieve each individual pixel value from the B channel
        int pixel = (int) TIFFGetB(raster[((width * (height - moveDown)
   ) - moveRight)]);
        // Convert the pixel, in integer form, to a float
        pixel_float = pixel;
        // Normalize the pixel value by dividing it by 255.0
        Norm_pixel_float = pixel_float / 255.0;
        // Convert the resulting decimal to IEEE-754 format
        Dec_to_IEEE(Norm_pixel_float);
        // Convert the resulting IEEE-754 number into a binary string
        pixel_ieee_string = &pixel_ieee;
        // Convert the 32-bit IEEE 754 number to a decimal
        pixel_ieee_bin =(int)strtol(pixel_ieee_string,(char **)NULL,2);
        // Write the pixels to the BRAM registers
        if(fileNum == 1){
            BRAM_CTRL_0_REG[i] = pixel_ieee_bin;
        }
        else if(fileNum == 2){
            BRAM_CTRL_1_REG[i] = pixel_ieee_bin;
        }
    }
}
```

Listing 3.4: Converting individual pixels to the IEEE-754 format and writing them to BRAM in C

Once the final frame has been sent by the server control script, the C code will continuously read from the AXI slave register of the Results IP to be notified when to start collecting the computed results. When triggered, the C server script will read from BRAM five, which is connected to the Results IP and AXI BRAM Controller two, to retrieve the result value. For the USB-based DICe design, the results will be converted from the

IEEE-754 format to scientific notation locally on the FPGA and locally formatted into solution files that are saved on the USB drive. The process of converting an IEEE-754 number back to decimal is essentially the reverse process shown in Listings 3.3 and 3.4 above. For the Ethernet-based DICe design, the results are read from BRAM, where they are automatically inferred as decimals, and transmitted back to the client script on the connected host PC. A similar code block is implemented on the client control script to convert the results back to a human-readable format and placed into text-based solution files.

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. The software designs in Section 3.2.3 were previously discussed and shows how the control scripts run the hardware-accelerated design. It covers both the USB-based design and the Ethernet-based design and how the high-level code interacts with the low-level hardware. 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.

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 correla-

tion 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 Inter-

connect 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 6.3.

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 6.3. 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 within the Subset Coordinates IP.

4.1.7 Gradients IP

TODO: Explain how the gradients in the X and Y direction are computed in the Gradients IP.

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.

Chapter 5

Results

TODO: Provide details on the results of the DICe hardware accelerator. Provide a comparison between the USB-based and Ethernet-based DICe designs. Was the project successful, and if so how? Was there any part of this project that wasn't successful in terms of the results?

Provide the following results here: 1.) The total execution time to convert the pixel values of a frame to IEEE-754 on the PC in Python 2.) The total execution time to convert the pixel values of a frame to IEEE-754 on the PC in C 3.) The total execution time to transfer a single image from the PC to the FPGA before IEEE-754 4.) The total execution time to transfer a single image from the PC to the FPGA after IEEE-754 5.) The simulation time of the DICe hardware design

5.1 DICe USB-based Design

TODO: Provide results and an explanation of the USB-based design where images are fetched from a local USB 3.0 flash drive.

Provide the following results here: 1.) Transfer speeds from the USB to the FPGA 2.) Transfer speeds from the FPGA to the USB (?)

5.2 DICe Ethernet-based Design

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

Provide the following results here: 1.) Ethernet transfer speeds from the host PC to the FPGA & total execution time 2.) Ethernet transfer speeds from the FPGA to the host PC & total execution time 3.) Ethernet transfer speeds when using the KC705 FPGA 4.) Ethernet transfer speeds when using the old lwIP echo-server

Chapter 6

Discussion

This section provides a discussion on some topics that were not covered in previous chapters as well as potential future works. This project was complex and time-consuming to develop due to the density of the DICe source code, the learning curve required to develop hardware-based applications, and understanding image processing. The DICe hardware accelerator evolved many times as more knowledge was gained from the DICe source code and the hardware that was used. The biggest evolution of this project, and one this paper highlights below in Section 6.1, 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 and performance of the application. The previous method of using the lwIP TCP/IP stack to implement 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 below in Section 6.2. From working with 32-bit floating-point numbers in Python-based and C-based control scripts to scaling up the design to support multiple frames and subsets when 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

When creating the DICe hardware accelerator a variety of challenges were experienced which this section hopes to expand on. The first challenge was dissecting the

DICe source code, which is composed of nearly 100 C++ files. Navigating these files and their order of execution was a massive task to undertake. The code needed to be executed so that it was possible to trace through its execution to track down key functions. This process alone took weeks of analysis to determine which functions were called and when. To successfully trace through the code, the DICe GUI couldn't be used and the DICe source code needed to be compiled from scratch. This in itself was a difficult task due to the minimal documentation provided on the topic and the old operating systems used to originally build the source code. Dozens of library packages were required to be installed on the OS of the host PC. Even with this accomplished, using debuggers and step-through methods to analyze the code only raised more questions than were answered. This challenge was compounded by the fact that the DICe source code is updated monthly. Only after working closely with the lead developer of DICe, Dan Turner of Sandia National Laboratories were the key functions of DICe revealed. The code consisted of the following functions to port over to Verilog: computeUpdateFast(), initial_guess(), initialize_guess_4(), initialize(), interpolate_bilinear(), interpolate_grad_x_bilinear(), interpolate_grad_y_bilinear(), gamma_(), mean(), residuals_aff(), map_to_u_v_theta_aff(), map_aff(), test_for_convergence_aff(), and save_fields(). The details of these implemented functions are explained in Section 2.3.

The next challenge to solve was how to efficiently transfer data across the developed system. The original problem of creating the DICe hardware accelerator was how to deal with the vast amount of data that is produced from the high-speed cameras that are needed to be processed. The only sensible option for transferring the frame data to the FPGA was by using a Gigabit Ethernet connection between the host PC that stores the data and the FPGA for processing. This problem in itself took months of development to begin to transfer data over Ethernet at all. Three variations of Xilinx FPGAs were used with only the ZCU104 board providing the necessary speeds to transmit such a high volume of data. Ethernet was successful on the KC705 FPGA, but only up to a speed of 56 Mbps which was far too slow. Only once the option of using the ZCU104 FPGA was available was using USB 3.0 to transfer data considered. This paper explores

the differences of the DICe hardware accelerator in terms of the USB-based and Gigabit Ethernet-based DICe designs. Even once Ethernet communication was established, it wasn't an easy process to unlock its full potential. The lwIP echo server provided by the Vivado SDK enabled the use of Ethernet but, after countless modifications over months of work, the Gigabit Ethernet was never achieved. Only once the option of using the PetaLinux tools was explored was Gigabit Ethernet available to the FPGA, along with USB 3.0 which previously was completely unavailable. Once the primary method of transferring data to the FPGA was implemented, it still left the challenge of how to transfer data from the FPGAs PS side to the PL side. The data received via the I/O ports required the use of the FPGAs provided processor. The core of the DICe hardware accelerator lived in the FPGAs PL side. To connect these two sides, two primary methods were used. The first was the implementation of the AXI slave registers within each custom IP developed within the hardware design. These registers were provided addresses from the AXI bus interface within the systems hardware design which meant that the values in those registers, which could be written to from the IPs themselves, could also be read and written too from the control scripts. The control scripts leverage the "mmap" function in C to write to available address space defined by the Linux-based kernel.

Once the challenge of system-wide communication was established, another one presented itself. With direct communication between the FPGAs PS and Pl side now available, how the DICe application would resume processing after the first correlation has started became a consideration. This execution flow required modifications within the hardware design of individual custom-built IPs and to the software designs that ran in the client and server control scripts. Several acknowledgment signals were programmed into both the software and hardware designs of the application to notify the FPGAs PS and PL sides that another image should be sent and written to BRAM, when an IP should start and stop its execution, and when the correlation was finished for all provided frames. The implementation of these acknowledgment signals was time-consuming to perfect based on the time it takes to finish a round of image correlation, to send a frame to the FPGA, or to convert the frame to the IEEE-754 format. With this problem

resolved through extensive development, some problems that are inherent to the DICe software were unable to be completed. One of such problems was to give users the ability to place subsets on the frame from a GUI. Currently, all of the subset definitions need to be defined in the Subsets.txt file for them to be applied to the image correlation routines. However, it is difficult for users to look at an image and inherently know that a subset needs to be placed at a given X and Y pixel-based center point of size Z. This problem was overlooked in the short-term development of the DICe hardware accelerator because users can still work with the DICe GUI to provide them with subset definitions.

Lastly, the most significant challenge faced during the development of the DICe hardware accelerator was the limited amount of BRAM resources on all FPGAs used for this project. BRAM was by far the most crucial resource in the DICe hardware design because each BRAM block is responsible for buffering the frame data and the computed data for the gradients and subsets of an image. Developing around this constraint resulted in a 448x232 sized frame to be the largest image size eligible to be processed. The consequence of this was the DICe hardware accelerator processing images four times smaller than the required image size of 896x464 that was defined in the statement of work for this project. While the DRAM resource was abundantly available, the reading and writing access times to DRAM proved to be too slow to keep up with the hardware accelerator which resulted in timing faults.

6.3 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.

Chapter 7

Conclusion

TODO: Finish off the paper by discussing the objectives of this thesis, if the objectives were achieved and how they were achieved with supporting results. What was accomplished from this work that is significant? Summarize the work that this thesis presents. Tell them what you already told them!

Bibliography