Accelerating Reverse Engineering Image Processing Using FPGA

Matthew Joshua Harris
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/2176
ACCELERATING REVERSE ENGINEERING IMAGE PROCESSING
USING FPGA

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Cyber Security

By
Matthew Joshua Harris
B.C.S.E., Wright State University, 2017

2019

Wright State University

__________________________  
J. M. Emmert Ph.D.  
Thesis Director

__________________________  
Mateen M. Rizki Ph. D  
Chair, Department of Computer Science and Engineering

Committee on Final Examination:

__________________________  
J. M. Emmert Ph.D.

__________________________  
Travis Doom Ph.D.

__________________________  
Michael L. Raymer Ph.D.

__________________________  
Barry Milligan, Ph.D.  
Interim Dean of the Graduate School
Abstract

Harris, Matthew Joshua, M.S.C.S, Department of Computer Science and Engineering, Wright State University, 2019. *Accelerating Reverse Engineering Image Processing Using FPGA.*

In recent decades, field programmable gate arrays (FPGAs) have evolved beyond simple, expensive computational components with minimal computing power to complex, inexpensive computational engines. Today, FPGAs can perform algorithmically complex problems with improved performance compared to sequential CPUs by taking advantage of parallelization. This concept can be readily applied to the computationally dense field of image manipulation and analysis. Processed on a standard CPU, image manipulation suffers with large image sets processed by highly sequential algorithms, but by carefully adhering to data dependencies, parallelized FPGA functions or kernels offer the possibility of significant improvement through threaded CPU functions. This thesis will examine the possibilities of moving a program featuring several image manipulation and analysis operations to a hardware/software build on a modern FPGA. The paper will focus on the implementation and performance improvements of the proposed method as well as the results of moving portions of the program to FPGA hardware.
Contents
List of Figures .................................................................................................................. vi

List of Tables ................................................................................................................... vii

1. Introduction ..................................................................................................................... 1
   1.1. Introduction ........................................................................................................... 1
   1.2. Motivation ............................................................................................................ 1
   1.3. Scope and Purpose ............................................................................................... 3
   1.4. Thesis Outline ...................................................................................................... 5

2. Background .................................................................................................................... 6
   2.1. Field Programmable Gate Arrays ......................................................................... 6
   2.2. Implementation of software algorithms on FPGA .............................................. 7
   2.3 The AIRE Program & Stitching [17] .................................................................. 17

3. Methodology .................................................................................................................. 20
   3.1. Project Constraints ............................................................................................. 20
   3.2. Challenges ........................................................................................................... 22
   3.3. Image Sets Used .................................................................................................. 33
   3.4. Data Analysis Method ......................................................................................... 33
List of Figures

Figure 2.1 The division of the SIFT algorithm into hardware and software [12].......................... 11

Figure 3.1. Difference between function with Dataflow and without Dataflow Pipelining [20]... 29

Figure 3.2 Effects of loop pipelining [20].................................................................................. 30

Figure 3.3 Hardware design for Zedboard used to generate SDSOC platform......................... 32

Figure A.1 Default Parts dialog ................................................................................................. 45

Figure A.2 Run Connection Automation for block design ......................................................... 47

Figure A.3 Final Block Design .................................................................................................. 48
List of Tables

Table 2.1 Comparison of step row filtering using Vivado HLS libraries and custom coded algorithms [18]........................................................................................................................................15

Table 2.2 Comparison of data binning using Vivado HLS libraries and custom coded algorithms [18]........................................................................................................................................16

Table 2.3 Comparison of Sobel filtering using Vivado HLS libraries and custom coded algorithms [18]........................................................................................................................................16

Table 2.4 success rate of each algorithm for the tested image set (MSP 430)......................................19

Table 2.5 timing comparison of different AIRE stitching methods on several differently configured machines..................................................................................................................19

Table 4.1 Estimated times for hardware functions vs CPU timing.......................................................34

Table 4.2 Timing comparison of functions provided by the xfopencv library ....................................34

Table 4.3 Timing comparison of custom functions .................................................................................35

Table 4.4 Zedboard slice logic usage for threshold stitch hardware functions.................................36

Table 4.5 Zedboard slice logic usage for difference stitch hardware functions.................................37

Table 4.6 Timing comparison of stitching algorithms for image sets of 1024x768 ............................38

Table 5.1 Comparison between RIPL and HLS OpenCV [22].............................................................43
1. Introduction

1.1. Introduction

When a program is executed by a computer, it is often executed using the CPU with instructions executed sequentially. This can create a bottleneck especially in programs using computationally intensive algorithms such as image processing algorithms. Because of this bottleneck, there has been research into moving the most computationally intense image algorithms to FPGA boards.

The hardware-in-the-loop implementation of these computationally intense programs is used since it allows for faster computation of the expensive algorithms without moving the entire program to hardware of the board. This saves time since moving the entire program could greatly increase the difficulty of the implementation of the program due to the constraints on the FPGA board. The use of the HW/SW system allows for less to be rewritten but can still accomplish an increase in performance.

1.2. Motivation

When a hardware circuit is commissioned or bought, there is the possibility that a third party might have added a malicious addition or subtraction to the intended circuit creating a dangerous vulnerability and attack vector on the chip [1]. These attacks are often hard to detect and can be easily missed [1]. One method of detecting hardware Trojans is with the use of image processing. This method compares an original trusted
chip with an untrusted one and detects inconsistencies in the physical layout and circuit components. One program, AIRE, presented in the paper, *Image Stitching and Matching Tool in the Automated Iterative Reverse Engineer (AIRE) Integrated Circuit Analysis Suite* employs this method and uses image stitching algorithms to process the visible physical differences that might be present in the circuit [2].

The size and complexity of the circuits require a large and detailed image to be processed thoroughly and precisely. The result of this is a computationally intensive image processing and stitching process. The problem then becomes the computational intensity of running the program and the limitations to the execution speed on a CPU. Large images can take up large amounts of processing power from the CPU and thus cause the program to run for long periods of time. The solution proposed in this project is to translate portions of the program to be run on FPGA hardware and to take advantage of the device’s parallelization to create a faster hardware-in-the-loop implementation of the program. The main portions moved to the FPGA will be the image processing and stitching portions.

By moving the computationally expensive image processing to the hardware, we hope to improve the performance of the whole program. In previous academic projects, the use of the hardware in the loop has been proven to be a beneficial update to improve on the speed of the program [3]. The challenges that arise when proposing this solution are the translating of the methods used in the original program to function effectively and
correctly on an FPGA board. The use of Xilinx developed IDE’s for C++ and VHDL (VHSIC Hardware Description Language) and will be used to aide in developing the new hardware/software-in-the-loop system [4]. The original program also uses the OpenCV library (an open source image processing and algorithm development library) which will be translated for use on FPGA using the Xilinx provided libraries or using newly developed code. The FPGA board which the program will be developed for and run on will be the Zedboard Zynq-7000 [5].

1.3. Scope and Purpose

The project will focus on creating a hardware in the loop implementation of image stitching and detection algorithms using an FPGA board to allow for parallel computation on the FPGA board and resulting in higher computational efficiency. The project will first profile the functions of the original program and assess which hold the most computational time of the overall program. These will be portions of the image algorithms used in the program and whichever can and will be beneficial in moving to HW will be translated to the FPGA.

Once these candidate functions are identified, they will be rewritten to compile and run on the HW portion of the HW/SW system. The rewriting of the functions will focus on creating new versions which can be run on FPGA and are parallelizable. The project will make use of the pre-made Xilinx libraries to convert the OpenCV functions as well as newly created conversion of the original program code for the FPGA. The
parallelization and management of data piping between the software and HW functions will also be built for optimal program flow between the hardware and software components of the system.

Finally, the resulting HW/SW system will be tested and refined in order to achieve the best possible speedup using the HW/SW system. The best use of the FPGA architecture and the most efficient way to combine the HW/SW portions of the program with effective data transfer. There will be multiple tests conducted for various structures of the HW/SW system and the best one will be used as the final design of the project.

The primary purpose of moving the computationally intensive portions of the AIRE program to the HW is to increase the speed of the overall program. The project will not focus on optimizing portions of the program which will remain on the software side. The target code moved to the FPGA hardware will be optimized and parallelized for faster execution. The code moved to the hardware will be written, compiled and tested with the Zedboard FPGA as the target system.

Although the Xilinx environment supports more FPGA’s and it is possible to compile the program for multiple FPGA’s, the program will be tested solely on the Zedboard. The result will be a hardware-in-the-loop system which will improve upon the speed of the original program. The program created by this project will be faster in execution than the previous and will aid in creating a more efficient hardware trust testing environment.
As mentioned before, the end goal for the project will be a performance speedup of the AIRE algorithms when moved to a HW/SW implementation. Based on research presented in chapter 2, the amount of performance increase should be around ten times the original time to complete the AIRE algorithms. This is based on the background research uncovering that the majority of HW/SW implementations of image processing programs have about a ten time increase over their original CPU implementations.

The hardware-in-the-loop program will be useful to users who own an FPGA such as the Zedboard Zynq-700 and wish to run a faster version of the AIRE program. The AIRE program is a computationally intense program therefore, the use of an FPGA and optimized version of the program would greatly decrease the amount of time needed for the AIRE program to run. Overall, the main purpose of this project is to create a faster alternative to the original AIRE program with the use of an FPGA.

1.4. Thesis Outline

This thesis has been split into five chapters, this being the introductory chapter. The rest of the chapters will be laid out as described here. Chapter 2 will be a literature review of the background research examined in preparation for this project as well as an outline of how the project will be accomplished. Chapter 3 will outline the method used to accomplish the project. It will document the various methods tried and their results and will go in depth over the design and implementation process of the project. The limitations, both anticipated and unforeseen, of the project will also be discussed in this
section. Chapter 4 will review the results of the project. It will outline the resulting program and through tests, determine the improvements it might have over the original AIRE program. Chapter 5 will examine the results of chapter 4. the chapter will discuss the successes, failures, and possible expansion of the capabilities of the AIRE program that this project has initiated. It will also conclude the thesis and suggest areas for future work

2. Background

2.1. Field Programmable Gate Arrays

In 1985, Xilinx developed one of the first FPGA’s [6]. This new technology was based on previous PLA (programmable logic array) devices with the added benefit of being re-programmable [7]. The new design combined the fast and reliable processing speed with the ability to program different uses for the chip. The result was a versatile chip that operated much faster than a CPU but still had the programming potential not present in other ultra-fast PLAs. The first edition of this technology produced by Xilinx had only 800 gates [6].

In the next decade FPGAs evolved from a basic 800 gate chip with limited functionality to newer chips comprised of over a million gates and the ability for software programming [7]. Although their development and performance were increasing rapidly, FPGAs remained fairly unknown until 1997, when a research paper by Thompson...
uncovered the untapped potential of FPGA chips [8]. Thompson used genetic algorithms to create a lightweight sound detection program on an FPGA chip using a small area [8]. The result was an efficient program that proved FPGAs could be coded in software with the computer then systematically moving the functions to hardware better and faster than any human could. The new evolving software development of the FPGA could be used to run many computationally expensive algorithms faster than a traditional CPU or GPU [8].

Currently, this and many other experiments like it have proven that the FPGA is a viable option in the optimization of algorithms. The Zedboard FPGA designed by Xilinx contains millions of gates along with an ARM Cortex-A9 [9]. It is an FPGA built for versatility at a relatively low price.

2.2. Implementation of software algorithms on FPGA

Since the use of FPGAs in computer programs has been proven to be advantageous in increasing the efficiency of the program, researchers have begun to explore the may different areas of algorithms and programs that could be improved with FPGAs. In recent years, research into program acceleration using FPGAs has begun to look towards the algorithmically complex area of computer vision.

In one previous study by Rettkowski et. al, conducted in 2017, several different versions of a single algorithm were created and tested on both a computer and FPGA controller [10]. The algorithm used, one which detected humans using histogram-oriented
gradients, is computationally complex and so the researchers wanted to determine the best implementation. Their three options were OpenCV on an ARM processor, hardware software codesign and a full FPGA hardware implementation [10].

The results of the assessment of these three methods proved the hardware only approach to be the fastest. The hardware only approach took only 25.5 milliseconds per frame [10]. The software and hardware implementation took 700 milliseconds per frame and the OpenCV using an ARM processor took 50 milliseconds [10]. The original time for one image was 12.7 seconds. One reason why the HW/SW implementation was slower was the lack of on-board DDR memory in the FPGA chip (Xilinx Zynq) meaning the image had to be passed back and forth between the controlling computers in between steps [10]. The constant transferring of data created a high overhead to the image processing, especially in the case of high resolution images [10].

However, the use of the hardware and software implementation of the image processing algorithm still had the promising speedup of over nine times the speed of the original program [10]. The main deterrent from an even greater increase in speed using the HW/SW method lies in the transferring of image data between the FPGA and the computer. The author is optimistic that future versions of the SDSOC will be able to increase the transference speed between software and the FPGA chip [10].

In the project, the main focus is on optimizing a image processing algorithm on an FPGA chip using the hardware and software combination with a hardware in the loop
implementation of image processing algorithms. According this paper the result would be a significant speedup of greater than 9 times compared to the original algorithm. This is even before the algorithm is parallelized.

Another paper examines the result of using the Canny edge detection algorithm on an FPGA chip. The Canny algorithm, another computer vision algorithm, is redesigned to work effectively on a FPGA board. The images are divided into 64x64 bit blocks to allow them to be stored in memory on the FPGA boards [11]. This eliminates the problem of data transfer between the FPGA and the computer, a costly process according to the previous research [10]. The algorithm is then parallelized and run using an FPGA board.

In order that the algorithm process images effectively even after the images are divided into separate blocks, the researches added an adaptive selection method to predict the high and low thresholds for the image edges [11]. The algorithm determines the threshold levels for each block based on the position of the block in the original image and the gradient of the blocks own pixels [11]. This allows for the blocks to be processed independently and therefore the image processing of the whole image can be done in parallel.

In the end, the use of the ported algorithm and parallelization of the image processing allowed for a 512x512 image to be processed in 0.72 mS on the FPGA board compared to the original time of 26.112 mS on a GTX 80 GPU [11]. Also, the addition of the adaptive threshold calculation allowed for less latency, better edge detection and the
possible addition of pipelining the image when processing on the FPGA [11]. The board used for this test was Xilinx Virtex-5 [11].

Currently, Xilinx has released a hardware accelerated and parallelized version of the Canny edge detection algorithm in their xfopenv library. This will be used in the project as it has been tested to be effective. It also has been parallelized and optimized to process images by breaking it into smaller blocks.

Another paper uses the SIFT image processing algorithm to map out the key points of an image. This is an effective way to map out the location of objects in images so that image stitching can be performed. The paper focused on optimizing the portion of the algorithm which detected key points in an image [12]. The algorithm is computationally expensive and thus would be helped by porting it to a parallel implementation on an FPGA board [12]. The algorithm would also need to be rewritten to function on an FPGA board as it originally uses a large amount of memory in its mapping of image keypoints [12].

The authors of the paper created a custom algorithm for the FPGA with lower memory and with parallel computation [12]. The Gaussian blur portions and several other computationally expensive portions of the SIFT algorithm were separated while other
portions were left to be processed on software [12]. Figure 2.1 best illustrates the division of these portions.

The results were tested for speed and match rates compared to the software implementation. In the end, the match rates for the hardware plus software SIFT algorithm were only slightly lower than those accomplished using just software [12]. The redesigned algorithm was able to process 72.6 million pixels per second [12]. The speedup compared to the software only implementation was 250 times [12]. The authors also pointed out that they only optimized one portion of the SIFT algorithm. There is more that can be optimized, especially the descriptors generation phase, another highly computational portion of the algorithm [12].

In the original AIRE project the Gaussian filter algorithm is used several times. This image algorithm is similar to the SIFT algorithm mentioned above as they both rely on image convolution. Xilinx has created a hardware accelerated portion of the Gaussian algorithm already in their xfopencv library. The paper on optimizing SIFT highlights the
advantages of moving this algorithm to hardware and how its convolution can be used as alternate steps in other image processing algorithms [12].

In a research paper published by Nazma Nausheen, the SOBEL edge detection algorithm was modified to function on a FPGA board. A 10-bit version of the algorithm was redesigned to use only 8 bits so that it might be used effectively on an FPGA board [13]. Along with this, the author added several other optimizations to the FPGA implementation. On such change is the use of direct indexing of image pixels instead of redundant storage of sub windows which make up the original image being processed [13]. This allows for a decrease in memory use on the FPGA board. The resulting algorithm sped up the Sobel edge detection process and was able to analyze a 512x512 image in 0.52 mS [13]. This speed up could be increased even more in future work since the design leaves more space open on the FPGA board. The extra space could be used to parallelize the Sobelk algorithm, resulting in even faster image processing [13].

The use of the Sobel edge detection algorithm on an FPGA board is also studied in another research paper by Norsat and Kavain [14]. The paper details how this algorithm can be implemented on the FPGA chip [14]. The focus is creating a working multidirectional Sobel edge detection algorithm that will run efficiently on an FPGA chip.

Another paper explores the use of several image processing algorithms implemented onto an FPGA board using three methods. The first is using the Xilinx
proprietary Sobel filter on the FPGA. The second is the use of C++ coding and a high-level synthesis tool like Vivaldo HLS [15]. This design is the simplest and easiest method to design programs for the FPGA board. The final method is using Matlab to map the Xilinx logic path [15]. This can create a much faster program but takes more time to design than the C++ with HLS method [15]. All these methods are successfully implemented and can be further modified to fit different algorithms [15]. For this project C++ will be used since it is the most familiar language for the designer.

Another paper by Menendez Alonso et al. lays out a method of transferring images from a computer to the FPGA boards [16]. The images are divided into smaller subwindows before being sent to the FPGA individually to be processed [16]. This is necessary because the small amount of available RAM on many FPGA boards does not allow for multiple large image matrices to be worked on at once [16]. The design creates the smaller subwindows by dividing the input by sections and loading each bit into a 3x3 window using line buffers [16]. Then, the image processing can be completed by the FPGA on the small subwindow and the results can be sent back to the computer program and loaded into one complete image [16]. The implementation resulted in an effective program for image processing using the FPGA in the loop. The use of space on the board was only 30% of the available LUTS and 60% MUXes [16].

Once on the board, further optimization can be made in the data access and the algorithms run on an FPGA board. This is dealt with in a research paper by Kelefouras et
al [17]. The paper proposes a new line detection algorithm implemented on an FPGA with higher efficiency than the state of the art OpenCV algorithm. The algorithm was developed with the researchers first taking notice of the FPGA data cache sizes [17]. After examining the limits of the data caches, the windows are broken down into sub windows which can fit continuously into the data caches [17]. This greatly reduces the number of misses when retrieving data to be manipulated. In turn, a decrease in data misses causes a speedup in the total algorithm.

Also, the author further modifies the algorithm by combining steps in pixel manipulation whenever possible [17]. The OpenCV Canny algorithm used in this paper goes through four image processing kernels and multiple read writes of the arrays as they are manipulated. The author combines all four processing kernels into one to reduce the reading and writing and directly consumes any outputs into the next step in the algorithm [17]. This allows for not only less read and write times, but also the use of parallelization of the data. As data passes through one step it is immediately consumed by the next portion of the algorithm [17].

The new HW/SW program was tested on a Xilinx Virtex 5 and resulted in a huge speedup [17]. Decreasing the time for the processing of a 640x480 image from 3.31 seconds to 0.0027 seconds [17]. The large jump in performance was mostly due to the decrease in window size and the decrease of data access misses by the program. The
original OpenCV optimized code is not the newer updated code used in this project but the great increase in performance points to the success of this proposal.

Another question that comes up is if the use of OpenCV libraries is the best option for image processing on and FPGA board. This is examined in a case study paper by Cortes, Velex and Irizar [18]. The study compares three image processing methods, step row filtering, data binning, and Sobel filtering designed for an FPGA using in one case the already available Vivado HLS OpenCV libraries and in another custom algorithm created by authors of the paper [18]. The results of the algorithms were tested for development time, runtime, and FPGA resources needed.

In the case of the step row filter, the use of OpenCV libraries greatly reduced development time but ended up with a much larger processing time and a greater use of the FPGA LUTS. This can be seen in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>RTL Designer (hand-coded)</th>
<th>SW Designer (Vivado HLS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev. Time (man-days)</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Proc. Time (ms)</td>
<td>0.95</td>
<td>4.5</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
<td>145</td>
<td>370</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
<td>120</td>
<td>414</td>
</tr>
<tr>
<td>Number of Block RAMS (RAMB18E1s)</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>DSP48E1s</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.1 Comparison of step row filtering using Vivado HLS libraries and custom coded algorithms [18].
Comparing the result for the data binning and Sobel filter functions produced similar results as seen in Tables 2.2 and 2.3.

<table>
<thead>
<tr>
<th></th>
<th>RTL Designer (hand-coded)</th>
<th>SW Designer (Vivado HLS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev. Time (man-days)</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Proc. Time (ms)</td>
<td>1.89</td>
<td>2.05</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>200</td>
<td>164</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
<td>121</td>
<td>20379</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
<td>196</td>
<td>18136</td>
</tr>
<tr>
<td>Number of Block RAMS (RAMB18E1s)</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>DSP48E1s</td>
<td>-</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 2.2 Comparison of data binning using Vivado HLS libraries and custom coded algorithms [18].

<table>
<thead>
<tr>
<th></th>
<th>RTL Designer (hand-coded)</th>
<th>SW Designer (Vivado HLS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev. Time (man-days)</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>Proc. Time (ms)</td>
<td>0.398</td>
<td>0.498</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>Number of Slice Registers</td>
<td>748</td>
<td>25439</td>
</tr>
<tr>
<td>Number of Slice LUTS</td>
<td>1111</td>
<td>21946</td>
</tr>
<tr>
<td>Number of Block RAMS (RAMB18E1s)</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>DSP48E1s</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2.3 Comparison of Sobel filtering using Vivado HLS libraries and custom coded algorithms [18].

As can be determined from the charts above, the main advantage of the use of OpenCV libraries already created for FPGA’s is the greatly reduced time to develop the
image algorithms. On the other hand, the use of OpenCV libraries pre-made for the FPGA sacrifice program runtime and use a greater amount of resources than the custom developed algorithms. The Sobel filter algorithm presents the best case for using the OpenCV algorithms over custom created ones. The time spent developing using OpenCV algorithms is much shorter at one day versus twelve days developing the custom algorithm [18]. Meanwhile the performance trade-off is not too great as the custom algorithm runs at 0.398 compared to 0.498 [18]. Still, the amounts of slice registers, LUTS, block RAM’s, and DSP48E1’s are all more than doubled [18].

The use of OpenCV does also carry one great advantage over custom algorithms in that they are widely known and used by software engineers [18]. OpenCV is a popular library that has been proven to work and has an active community of developers and users behind it. The use of OpenCV not only allows an engineer to develop faster for an FPGA but also creates an easier handoff of projects since a second engineer can become familiar with OpenCV much quicker using the large amount of online information on the vision library. Also, since this paper was completed, Xilinx has improved the performance of their OpenCV libraries. For these reasons OpenCV will be used on the current FPGA project.

2.3 The AIRE Program & Stitching [2]

The AIRE program as outlined in the paper, Image Stitching and Matching Tool in the Automated Iterative Reverse Engineer (AIRE) Integrated Circuit Analysis Suite is
designed for the stitching of portions of microchips together into one complete image [2].
This helps in the image creation for microchips that have a larger surface area than the camera taking photos is able to capture in a single shot.

Due to this limitation, the photographer will take multiple pictures as it moves along the microchip resulting multiple images for a single chip. The images might be slightly off center due to movement of the chip or camera or non-uniform image capturing. This is the problem that the AIRE program works to solve by image stitching. The edges of two images are processed in order to test for a possible match. If they match, then the images can be considered adjacent to each other and can be combined. The AIRE program contains several methods with used to complete this stitching, differential, threshold, and weighted stitching.

The three algorithms use different image processing functions which will be reviews in section 3.1. The successfulness of each algorithm for an image set (MSP 430) can be seen in Table 2.3.
Table 2.4 success rate of each algorithm for the tested image set (MSP 430) Perfect denotes an image matches within 1 pixel. Good denotes image matches within 12x5 pixels [2].

Also, in the paper on AIRE, there are several times listed for different computer environments the program was run on. The image set for these tests was of 304 2048x1768 images. Table 2.4 examines these times.

<table>
<thead>
<tr>
<th>Computer</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel i7-4650U</td>
<td>Intel N37000</td>
<td>AMD A10-7850K</td>
<td>AMD TR-1950X</td>
<td>Intel i7-8550U</td>
</tr>
<tr>
<td>Cores</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Threads</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>Threshold</td>
<td>541.13 s</td>
<td>1960.20 s</td>
<td>698.29 s</td>
<td>54.78 s</td>
<td>270.03 s</td>
</tr>
<tr>
<td>Weighted</td>
<td>43.49 s</td>
<td>47.87 s</td>
<td>21.30 s</td>
<td>16.87 s</td>
<td>23.96 s</td>
</tr>
<tr>
<td>Differential</td>
<td>423.51 s</td>
<td>1028.79 s</td>
<td>353.68 s</td>
<td>19.41 s</td>
<td>110.78 s</td>
</tr>
</tbody>
</table>

Table 2.5 timing comparison of different AIRE stitching methods on several differently configured machines. The times correspond to 304 images of size 2048x1768 [2].
These times represent a threaded optimized version of the AIRE algorithm run on the author’s machines. This thesis will compare results based on new calculated times. This is to ensure that the data used to produce timing results are uniform in comparing CPU and FPGA performance.

3. Methodology

3.1. Project Constraints

This project will focus on the optimization of an OpenCV based image processing algorithm, AIRE, on a FPGA, the Zedboard, with the primary goal of improving program efficiency. The starting devices and software used to develop the project are as follows.

**AIRE** – A circuit analysis program that uses image processing to determine the architecture of a given circuit. The program is computationally expensive when run on a CPU. The goal of this project is to accelerate the program by implementing the image processing functions on hardware.

**Zedboard** - The FPGA board costs about $400 making it affordable while still being a very capable FPGA. The device has 512 Mb DDR3 RAM and the Zynq-7000 as the SOC. The Zynq-7000 contains an ARM Cortex A-9 processor and contains I/O peripherals such as 2 SPI ports and GPIO with 4 32-bit banks [8]. This allows for fast data transference between the computer and the FPGA board. One downside to using this board is the lack of support for compiling xfopencv using Xilinx tools.
xfopencv- The open source library provided by Xilinx containing more than 50 kernels optimized for use on the FPGA. The library methods are based on the existing OpenCV image processing functions which have been converted to take advantage of Zedboard capabilities such as parallelization.

Xilinx SDSOC – A IDE based on Eclipse for C++ coding and compilation for Xilinx FPGA’s. The IDE allows a user to load a hardware platform provided by Xilinx or custom platform with which they can compile a program for. The compiler is able to build according to HLS defined pragmas and move functions from software compilation to hardware compilation. It is available for use with a trial license or for purchase.

Xilinx SDK and Petalinux – These are tools provided by Xilinx that can be used to create custom platforms for the Xilinx SDSOC compiler. They are needed since Xilinx does not provide a Zedboard platform with compiled OpenCV libraries and xfopencv included. Xilinx SDK is included with the license for SDSOC and Petalinux is open source.

The main focus of the project is to convert the following image processing algorithms to run on the FPGA hardware, differential stitch, and threshold stitch.

Differential stitch

difference, bitwise XOR, and minimum and maximum location. All these functions have counterparts in the xfopencv library and the calls can be swapped out with ones to xfopencv. Only one feature used in this section is missing in the xfopencv library. This is the ROI (region of interest constructor). This constructor allowed a user to specify a mat, coordinates within the mat and a size and would initialize the cv mat to the resulting sub mat. This constructor needed to be recreated for the xf mat data structure as a new submat function.

*Threshold Stitch*

The original algorithm uses the Gaussian blur function. After running the Gaussian blur filter on the input images, they are then run through an image manipulation function created especially for the program. This will need to be completely reworked to allow for hardware acceleration and parallel processing. This method will also need to use the submat function originally created for the differential stitch.

3.2. Challenges

*Difference in calls from OpenCV to xfopencv functions*

One major change that occurs when switching from OpenCV on the CPU to xfopencv is that the size of the image mats used in all functions must be known at compilation time. This allows the FPGA to allocate memory for any of the mat data structures in contiguous memory, resulting in faster data access and therefore faster
execution. A difference of the function calls can be seen in the Gaussian blur example below.

Cv::GaussianBlur( )

xf::GaussianBlur <FILTER_WIDTH, XF_BORDER_CONSTANT, XF_8UC1, HEIGHT, WIDTH, XF_NPPC1>( )

The HEIGHT and WIDTH template options determine the maximum size able to be passed to the compile hardware function. XF_8UC1 is the data type used in the input matrix, XF_8UC1 signifies an 8-bit unsigned one channel pixel [19].

Writing code for hardware acceleration

For the custom methods which do not have any xfopencv counterpart, it is required to write new code for hardware acceleration. In these cases I refer to the available code from the xfopencv codebase as an example. The use of available HLS pragmas for parallelization and libraries aid in this task as well [19].

The threshold stitch was the major method that needed to be rewritten to hardware. The starting code was written as shown below.

    cv::Size szba((int)((double)i1-cols*(cmdargs.ovrpct+cmdargs.ovrtoler)),i1->rows);  
    cv::Mat wnd[2],gau[2],thr[2];  
    wnd[0]=cv::Mat(*i1,cv::Rect(cv::Point(i1-cols-szba.width,0),szba));  

wnd[1]=cv::Mat(*i2,cv::Rect(cv::Point(0,0),szba));

for(int i=0;i<2;i++) {

    cv::GaussianBlur(wnd[i],gau[i],cv::Size(5,5),0);

    thr[i].create(szba,CV_8UC1);

    thr[i].setTo(0);

    for(int y=0;y<szba.height;y++) {

        for(int x=0;x<szba.width;x++) {

            unsigned char px=gau[i].at<unsigned char>(cv::Point(x,y));

            if(px>=pxthr[1]) thr[i].at<unsigned char>(cv::Point(x,y))=255;

            else if(px>=pxthr[0]) thr[i].at<unsigned char>(cv::Point(x,y))=128;

        }

    }

}

In order to move this to hardware, the first step is moving the data structure for the Mat from OpenCV to xfopencv. Xilinx provides function calls to do this. After this, in order the hardware be able to perform parallel operations, the mat needs to be converted to a HLS stream data structure.
#pragma HLS DATAFLOW

for (short int i=0; i<_src_mat.rows;i++)
{

#pragma HLS LOOP_TRIPCOUNT min=1 max=HEIGHT
for(short int j=0; j<(_src_mat.cols)>>XF_BITSHIFT(XF_NPPC1);j++)
{

#pragma HLS LOOP_TRIPCOUNT min=1 max=WIDTH/XF_NPPC1

#pragma HLS PIPELINE

#pragma HLS LOOP_FLATTEN off

dst.write( *(_src_mat.data + i*(_src_mat.cols)>>XF_BITSHIFT(XF_NPPC1)) +j);
}
}

One capability lost by this is random pixel access. The new structure is a stream and the data can now only be access iteratively. This requires that lines such as

unsigned char px=gau[i].at<unsigned char>(cv::Point(x,y));

be rewritten. Using the open source code as a guide, a new method for accessing the pixels can be created while accounting for bit offset in the images. In order to do this
effectively, a smaller sliding window is created to shift through the stream data without using too much memory.

...
This portion of the for loop prepares the next right column to be loaded into the current window while also moving the line buffer onto the next column.

```c
if((col < width) && (row < height))
{
    pix_t pix = inp_img.read();
    right_col[WINDOW_SIZE-1] = line_buffer[WINDOW_SIZE-1][col] = pix;
}
...

// This design assumes there are no edges on the boundary of the image
if ( (row>=FILTER_OFFS) & (col>=FILTER_OFFS) & (row<height) & (col<width) )
{
    filt_out = thresholdFilter(window[1][1], pxthr);
}
...

if ( (row>=FILTER_OFFS) & (col>=FILTER_OFFS) & (row<height) & (col<width) )
{
    out_img.write(filt_out);
}
} // end of L2
The function checks one pixel at a time making the transition simpler. The middle pixel of the window is passed to the threshold filter function where it is put through the if else test as in the original algorithm. Afterwards the resulting stream in converted back to an xfopencv mat and then finally an OpenCV mat for the final product.

Another portion needed for this hardware accelerated function to work more efficiently is the use of Xilinx defined pragmas. SDS pragmas are read by the sds++ compiler and contain direction in the compilation of C++ into programmable logic. The SDS data access_pattern pragma option is set to sequential access which communicates to the compiler to build a streaming data access instead of a RAM [20]. This saves on the on-board RAM and allows the hardware to process large image mat structures. Also, the SDS data copy pragma specifies that data is copied between the software and hardware functions. This can be seen in the conversion from mat to HLS stream.

The other set of pragmas used are in the HLS pragma set. The HLS pragmas help reduce latency and optimize any software for the FPGA [20]. The HLS dataflow pragma allows for the functions and loops to be pipelined. This pragma directs the sds++ compiler to analyze the code and create pipelines so that data can be passed from functions immediately after being operated on and before the function has been completed.
Using the code in Figure 3.1 as an example, func_B might depend on data b which is being used and changed in func_A. If no datflow pipeline was being used, func_B would have to wait until func_A is returned before executing but with pipelining, func_B can begin executing as soon as func_A finishes its operations on the b variable.

The result is a faster parallelized program as seen in Figure 3.1.

Figure 3.1. Difference between function with Dataflow and without Dataflow Pipelining [20].
The HLS pragma *pipeline* guides the HLS hardware in parallelization pipelining \[20\]. The pipeline pragma can set the number of cycles until the initiation of the next loop. This allows for faster parallelization as shown below.

Figure 3.2 Effects of loop pipelining [20]

Other pragmas such as *HLS INLINE*, *HLS STREAM*, and *HLS LOOP_FLATTEN* all add options that can improve the latency of the data access and the overall algorithm. *HLS INLINE* controls how a function is kept in the hierarchy, if a function is inlined it may be combined with its calling function and optimized in the same way \[20\]. *HLS STREAM* causes the arrays to be stored as streams in FIFO’s instead of another structure in RAM. This decrease use of the limited RAM and increases data access efficiency \[20\]. *HLS LOOP_FLATTEN* combines outer and inner loops into the same level improving
efficiency in data access. In RTL, it takes one clock cycle to move between inner and outer loops. Combining them as one loop eliminates the cycle needed to move between them [20].

Compiling the code for the Zedboard

Xilinx currently does not feature a pre-built package to build xfopencv application for Zedboard. Therefore, a new system package needed to be created in order to compile the OpenCV libraries. Using the Vivado tool it is possible to complete this. I created the Zedboard hardware design based on the Xilinx user guide 1236 [21].
After completing a hardware definition, I used Petalinux to compile libraries for a Zedboard. The Petalinux tool requires the hardware definition created from the project above. This results in 32 bit libraries for OpenCV compiled for ARM processors.

I then used the OpenCV libraries and the given Xilinx platform files for ZCU 102 to create a new platform for Zedboard. I then used this platform as my target and was able to compile for the Zedboard using the SDS++ compiler provided by Xilinx. A more thorough step by step guide can be found in Appendix A.
3.3. Image Sets Used

The testing of the newly designed algorithm will be performed on data sets used in the original AIRE program. The data sets will be run on both a personal computer using the original program and, on the FPGA, using the new version of the algorithm.

3.4. Data Analysis Method

The project aims to increase the speed of analysis of the given data sets compared to the original computer run program. The timing on the CPU will be conducted using the C++ chrono library. This will provide timing accuracy in fraction of milliseconds. The Zedboard allows for an onboard clock to time the algorithm when executing. This clock will allow time to be estimated in clock cycles which will then be converted to time.

4. Results and Analysis

4.1. Timing Results

The resulting times from the image processing algorithms being run on the FPGA showed improvement when they could be parallelized. Tables 4.1 displays the estimated times for processing a 1920 x 1080 image on FPGA hardware vs the original times when run on a CPU (Intel® Core™ i7-7700HQ at 2.8 GHz). These hardware processing times are generated by the SDS++ compiler and have a 2.7 nS uncertainty.
The result is a drastically faster implementation of the individual AIRE functions while on the FPGA hardware. These estimates though, do not take in account the time needed to move data from the software to hardware. This can be timed in real time and the resulting times are as follows in tables 4.2 and 4.3. The comparison is between the Zedboard FPGA vs Intel® Core™ i7-7700HQ at 2.8 GHz.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Est. HW time (μS)</th>
<th>CPU time (mS)</th>
<th>Functions</th>
<th>Est. Time (μS)</th>
<th>CPU time (mS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Blur</td>
<td>8.81</td>
<td>3.12</td>
<td>Count non zeroes</td>
<td>7.98</td>
<td>1.71</td>
</tr>
<tr>
<td>Bitwise XOR/ Bitwise OR</td>
<td>7.27</td>
<td>0.366</td>
<td>Threshold Stitch</td>
<td>7.27</td>
<td>1.32</td>
</tr>
<tr>
<td>Min and Max Location</td>
<td>7.27</td>
<td>1.91</td>
<td>Mat error propagation</td>
<td>7.27</td>
<td>0.187</td>
</tr>
<tr>
<td>Absolute Difference</td>
<td>7.27</td>
<td>0.495</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1 Estimated times for hardware functions vs CPU timing.

<table>
<thead>
<tr>
<th>Functions</th>
<th>CPU Implementation (mS)</th>
<th>FPGA Implementation (mS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Blur</td>
<td>3.12</td>
<td>8.2</td>
</tr>
<tr>
<td>Bitwise XOR/ Bitwise OR</td>
<td>0.336</td>
<td>4.42</td>
</tr>
<tr>
<td>Min and Max Location</td>
<td>1.91</td>
<td>0.414</td>
</tr>
<tr>
<td>Absolute Difference</td>
<td>0.495</td>
<td>7.43</td>
</tr>
</tbody>
</table>

Table 4.2 Timing comparison of functions provided by the xfopencv library
<table>
<thead>
<tr>
<th>Functions</th>
<th>CPU Implementation (mS)</th>
<th>FPGA Implementation (mS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count non zeroes function</td>
<td>1.71</td>
<td>6.80</td>
</tr>
<tr>
<td>Threshold Stitch</td>
<td>1.32</td>
<td>9.98</td>
</tr>
<tr>
<td>Mat error propagation (originally comprised of count non zeroes and addition functions)</td>
<td>0.187</td>
<td>0.499</td>
</tr>
</tbody>
</table>

Table 4.3 Timing comparison of custom functions

The use of parallelization and faster access provided by the FPGA hardware allows for a speedup when implemented. All the functions listed above took advantage of the *Dataflow* and *Pipeline* features available when using the SDSOC compiler and supported FPGAs. As mentioned in section 3.2 these pragmas allow for the algorithm code to be executed in parallel and with pipelining between loop iterations.

The algorithms for bitwise or, bitwise xor, threshold stitch, and the mat error propagation algorithm all process input data iteratively so they are able to be moved to hardware with the least difficulty and with great results.
Processes such as the Gaussian blur and min and max location are more complex as they depend on the surrounding pixels. The process to translate to the FPGA hardware is more time consuming and complex but the performance boost is just as good as iterative programs when done correctly. Since these portions were already provided in the xfopencv codebase they did not require extra development time.

4.2. Hardware Usage

Due to the size of the Zedboard and limitations of hardware, not all functions are able to be put on hardware at the same time, leaving some to be processed on the ARM processor. In the final implementation used to generate the results in section 4.2, the bitwise or and bitwise xor sections were left on software. The amount of hardware slice logic used for the threshold stitch is demonstrated in Table 4.3. Table 4.4 shows the slice logic usage for the differential stitch.

<table>
<thead>
<tr>
<th>Site Type</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT Flip Flop Pairs</td>
<td>13187</td>
<td>53200</td>
<td>24.79%</td>
</tr>
<tr>
<td>Logic LUTS</td>
<td>22401</td>
<td>53200</td>
<td>12.10%</td>
</tr>
<tr>
<td>Memory LUTS</td>
<td>2106</td>
<td>17400</td>
<td>12.10%</td>
</tr>
<tr>
<td>Other (unique control sets)</td>
<td>1681</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice Logic</td>
<td>10724</td>
<td>13300</td>
<td>80.63%</td>
</tr>
</tbody>
</table>

Table 4.4 Zedboard slice logic usage for threshold stitch hardware functions
<table>
<thead>
<tr>
<th>Site Type</th>
<th>Used</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT Flip Flop Pairs</td>
<td>13187</td>
<td>53200</td>
<td>24.79%</td>
</tr>
<tr>
<td>Logic LUTS</td>
<td>36337</td>
<td>53200</td>
<td>68.30%</td>
</tr>
<tr>
<td>Memory LUTS</td>
<td>3753</td>
<td>17400</td>
<td>21.57%</td>
</tr>
<tr>
<td>Other (unique control sets)</td>
<td>2985</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Slice Logic (Total)</td>
<td>13285</td>
<td>13300</td>
<td>99.89%</td>
</tr>
</tbody>
</table>

Table 4.5 Zedboard slice logic usage for differential stitch hardware functions

For the differential stitch, the bitwise XOR function had to be removed as it caused the resulting hardware portion to use more slice logic than available. The bitwise XOR was used only once per iteration of the differential stitch so moving it back to software would not affect performance much.

4.3 Loss of Performance due to Overall Complexity

Because the AIRE program continually loops though the image manipulation algorithms, testing different size sub mats as it performs the image manipulation algorithms piece by piece, the onboard FPGA CPU still must handle many instructions. The ARM Cortex A-9 processor lacks power compared to a modern CPU and the onboard RAM cache is much smaller as well. The result is a severe loss in overall
performance due to the constant looping performed by the AIRE program as seen in Table 4.6.

<table>
<thead>
<tr>
<th>Stitching Algorithm</th>
<th>CPU Implementation (S)</th>
<th>FPGA Implementation (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Differential Stitch</td>
<td>0.03</td>
<td>9.5</td>
</tr>
<tr>
<td>Threshold Stitch</td>
<td>0.007</td>
<td>6.24</td>
</tr>
</tbody>
</table>

Table 4.6 Timing comparison of stitching algorithms for image sets of 1024x768

As one can see, the difference in time between the two methods is extremely bad for the FPGA side. The relatively small amount of RAM (512 Mb) available to the CPU leads to many page faults when running on software. Also, the constant movement of matrix data between hardware and software uses large amounts of clock cycles on the FPGA ARM processor. According to the SDS compiler the amount of clock cycles needed for the Datamover setup time is 2,117,715 cycles and movement of data is 3,659,437 cycles for a 1920 x 1080 image. This adds up to $5.78 \times 10^6$ cycles or ~8.67 mS per transference of data (based on ARM Cortex a-9 operating at a max rate of 667 MHz).

On a modern CPU, the available RAM is often several GB and the page faults are much lower. The matrix data can be passed between the calls to image manipulation functions much easier and because of this, a computer with a modern CPU can easily beat the FPGA in overall performance for the AIRE project.
It is possible that the AIRE program might be rewritten to run almost completely on hardware. This could result in much greater performance overall for the differential and threshold stitching algorithms.

5. Conclusion and Future Work

5.1. Conclusion

The results discussed in Chapter 4 highlight the advantages and disadvantages of converting the AIRE image manipulation algorithms to a hardware in the loop implementation. The moving of OpenCV functions to the Zedboard hardware did improve the time for image manipulation operations but the overall algorithm suffered due to expensive data movement handled by the on-board CPU.

Also, the limitations imposed by the available hardware area by Zedboard also affect performance of the overall algorithm. Functions bitwise XOR in the differential stitch method had to be moved off hardware further damaging the performance.

Although the overall program did experience a loss in speed when moved to a hardware in the loop implementation, the increased time for the individual methods show promise for future work. The estimated hardware time was over 100 times faster than the CPU implementation. Therefore, if data movement of the total image can be reduced, the hardware accelerated functions will suffer less time delay. This can be accomplished by
moving the entire algorithms for differential stitch and threshold completely to hardware. The resulting AIRE program will no doubt see an increase in timed performance.

5.2. Future Work: Moving the Majority of the Algorithm to Hardware

The moving of the entire differential stitch and threshold stitch algorithms to hardware would very likely increase the overall performance of the AIRE program. The major difficulties that would need to be accounted for if this were to be accomplished include the variable grid size of the sub-images being processed and the necessary changes to the parallelization of the hardware functions.

In the original AIRE program, one of the main features is that the original image is broken into variably sized sub image grids and each one is processed individually [2]. This cause a problem when moving this feature to hardware using the xfopencv and HLS libraries. The xfopencv data stores the array as a stream, and the only this can access is either iteratively or using a two-dimensional array of predetermined size [2]. This therefore requires that the grid sizes be known at the time of compilation, something that the AIRE program does not provide since they are calculated during runtime.

It is possible to keep the grid sizes dynamic and read different grid size out on hardware while passing them to the image manipulation functions. However, this method would require the developer to reexamine the parallelization of the hardware functions. Reading from the same image stream by parallelized functions could easily lead to a data deadlocks and corrupted streams. The best method to alleviate any problems that might
occur would be to examine the hardware functions and determine an effective scheduling and mutual exclusion alternative to the already provided xopenvcv pragmas.

Also, the movement of the entire algorithm to hardware might require a larger FPGA than the Zedboard used in this project. As seen in section 4.2, the limitations of the Zedboard required the two algorithms to be broken up and also the bitwise XOR to be taken off hardware in the final program. A FPGA with more available slice logic could aid in solving this problem (The Slice LUTS needed for all function to be moved onto the hardware was 61,454 LUTS while the Zedboard has only 53,200 available).

5.3. Transposing Images Passed to Hardware

The transposition of images for the FPGA might also prove to improve the overall image processing performance. In the AIRE program, not all portions of the passed images could be truly necessary to the stitching process. The overlap can be estimated beforehand and only the candidate portions are passed to hardware. This results in smaller matrices being passed from software to hardware and vice versa. Therefore, the total time moving data between hardware and software and the total runtime of the program can be reduced. After being moved to the FPGA hardware, the images could be transposed to match the edges to any other image it is compared with for stitching. Since only a portion of each image is necessary to conduct the stitching, removing the center of images and then transposing them when processing would have no effect on the data results while expediting the stitching algorithms.
5.4. Future Work: RIPL

If one might want to try a whole different method in future work, the use of a RIPL implementation on an FPGA might prove to be beneficial. According to an academic paper, the RIPL (Rathlin image processing language) is a domain specific language built with high level abstract image processing algorithms [22]. RIPL is developed with abstract skeleton image processing algorithms that can be used to create a custom image manipulation algorithm [22].

Unlike OpenCV, it can be changed and modified by the programmer to create existing or new algorithms. OpenCV is set to use the pre-existing algorithms and kernels. The RIPL algorithms are made from the basic RIPL skeleton [22]. The algorithm steps in the skeleton can be made to accompany any type of image processing algorithm [22]. Also, the data-flow has been changed with the goal of achieving better runtimes on an FPGA. The RIPL algorithm allows for data sharing between two image data pipelines while OpenCV requires the data to be duplicated before it is accessed by another pipeline [22]. OpenCV requires an image and pixel size to be set on compilation while RIPL can infer the dimensions of the input and intermediate images [22]. Also, RIPL erases the need for HLS pragma management for certain functions as it can automatically generate hardware pipelines. The results of these improvements lead to greater performance and less hardware requirements when processing images on an FPGA. Figure 2.5 highlights these improvements between the HLS OpenCV library vs RIPL.
Table 5.1 Comparison between RIPL and HLS OpenCV [22]

In this example, Sobel 2D edge detection still outperforms RIPL. The author claims this might be due to the multiple RIPL dataflow transitions needed to complete a Sobel edge detection [22].

The paper in which the use of RIPL algorithms to process images on an FPGA is new with a publication in March of 2018 [22]. There is possible room for improvement in the methods as time goes by and the algorithms become more widely known. The RIPL algorithms show promise and could be used in future image processing applications on FPGAs.
Appendix A: Porting an OpenCV Application to Zedboard

Part 1: Create the Zedboard Hardware Files

The steps for the creation of the hardware files are taken from Lab 1 in UG1236 [23]. Refer to the lab for more in-depth explanations for some of the steps if needed.

1. Start the Vivado Design Suite IDE and select the Create Project option found under the Quickstart menu.

2. The Project Wizard page should open, click next and you will reach the page asking to set your project name. Enter the name Zedboard. Also, select a location for project files and make sure Create Project Subdirectory is checked. Click Next.

3. On the Project Type page, select RTL Project.

4. On the add sources page, set the Target Language to C++ and Simulator language to Mixed.

5. Do not add any constraints on the Add Constraints page, just click Next.

6. On the Default Parts dialog, select the Boards tab and then click the Zedboard Zynq Evaluation and Development Kit.
7. Review the settings on the Summary page and then click Finish.

8. Now that the new blank project is open, find **Flow Navigator => Ip Integrator** and select **Create Block Design.**

9. On the Create Block Design dialog, set the name to **Zedboard** (the name of the block design must be the same as the name of the Vivado project).

10. In the new Block Design Canvas, right-click and **Add IP.** In the Dialog that pops up, select the **ZYNQ7** processing system IP.

11. Click the **Run Block Automation** option found in the IP Integrator window. In the window that appears, check the **All Automations** and **Apply Board Preset** boxes then **OK.**

12. Now add the Processor System Reset. Right click the IP integrator diagram and select **Add IP.** Search for **proc sys res** to find the Processor System Reset and add it to the design.
13. Repeat Step 12 three more times which will result in a total of four Processor System Resets in the design.

14. Using the Add IP option again, add a Clocking Wizard and Concat IP.

15. Now, the settings for the device IP must be set manually. Double click the ZYNQ7 IP to bring up the properties.


17. While still in the dialog, select Interrupts. Check the Fabric Interrupts box. Now expand the Fabric Interrupts drop-down menu and then the PL-PS Interrupt Ports drop-down menu and check the IRQ_F2P[15:0] box.

18. Click OK to save the apply the changes and close the dialog.

19. Next click on the clocking wizard to open the Re-Customize IP dialog. Open the Clocking Options tab and check the Input Frequency box. Set the value to 50.0.

20. Switch to the Output Clocks tab and check the boxes for clk_out2, clk_out3 and clk_out4. Change the output frequencies for the 4 activated clocks to 100, 142, 166, and 200 respectively. Finally, set the option at the bottom of the page Reset Type to Active Low. Click OK to apply the changes.

21. Double click the Concat IP to bring up the Re-customize IP dialog box and change the Number of Ports to 1. Click OK to apply the changes.
22. At this point, a green box should appear with the option to Run Connection Automation. Click this option and edit the dialog is opened to match the figure below.

23. Click OK to apply the automation. The resulting block design diagram should look like the following.
24. Validate and then save the block design.

25. Set the platform properties. This can be done by entering the following tcl commands into the IDE command prompt separately and pressing Enter after each one:

Set Platform Identification properties:

```tcl
set_property PFM_NAME "xilinx.com:zynq7_board:zynq7_board:1.0" [get_files [get_property FILE_NAME [get_bd_designs]]]
```

Set clock properties:

```tcl
set_property PFM.CLOCK { \ cli_out1 [id "2" is_default "true" proc_sys_reset proc_sys_reset_0] } \n```

Figure A.3 Final Block Design
clk_out2 {id "1" is_default "false" proc_sys_reset "proc_sys_reset_1" } \ 
clk_out3 {id "0" is_default "false" proc_sys_reset "proc_sys_reset_2" } \ 
clk_out4 {id "3" is_default "false" proc_sys_reset "proc_sys_reset_3" } \\
} [get_bd_cells /clk_wiz_0]

Set AXI Port properties

set_property PFM.AXI_PORT { \
M_AXI_GP0 {memport "M_AXI_GP"} \ 
M_AXI_GP1 {memport "M_AXI_GP"} \ 
S_AXI_ACP {memport "S_AXI_ACP" sptag "ACP" memory "processing_system7_0 ACP_DDR_LOWOCM"} \ 
S_AXI_HP0 {memport "S_AXI_HP" sptag "HP0" memory "processing_system7_0 HP0_DDR_LOWOCM"} \ 
S_AXI_HP1 {memport "S_AXI_HP" sptag "HP1" memory "processing_system7_0 HP1_DDR_LOWOCM"} \ 
S_AXI_HP2 {memport "S_AXI_HP" sptag "HP2" memory "processing_system7_0 HP2_DDR_LOWOCM"} \ 
S_AXI_HP3 {memport "S_AXI_HP" sptag "HP3" memory "processing_system7_0 HP3_DDR_LOWOCM"} \ 
}
Connect interrupts to xlconcat (Concat IP)

```bash
set intVar []
for {set i 0} {$i < 16} {incr i} {
    lappend intVar In$i {}
}

set_property PFM.IRQ $intVar [get_bd_cells /xlconcat_0]
```

26. Next, generate the HDL files. To do this right click on the block design diagram and select the **Generate Output Products** option from the menu.

27. When the dialog appears select **Synthesis => Global** and then **Generate**.

28. Once the synthesis is finished, right click the block design title in the Sources menu to the left of the block design and select **Create HDL Wrapper**. Select the option to allow Vivado to manage the wrapper and click OK.

29. Lastly, export the hardware files by selecting **File=>Export=>Export Hardware**.

---

Part 2: Create the Zedboard Libraries using Petalinux [24]

Petalinux runs on the following OS [24]:

---
- Red Hat Enterprise Workstation/Server 7.2, 7.3, 7.4 (64-bit)

- CentOS 7.2, 7.3, 7.4 (64-bit)

- Ubuntu Linux 16.04.3 (64-bit)

NOTE: If you choose to run the Petalinux on a virtual machine, make certain there is enough hard disk space, I would recommend at least 50 GB, Xilinx recommends 100 GB.

Petalinux requires third party libraries, check the list at Table 2-1 in the Petalinux guide:


NOTE: To get the 32 bit library of zlib on Ubuntu, use `sudo apt-get install zlib1g:i386`.

Petalinux requires the system to use bash, on systems like Ubuntu it is dash by default. To switch to bash enter the command: `sudo dpkg-reconfigure dash` when the option to use dash comes up, select no.
1. Download the Petalinux install script from Xilinx at
   
   
   Nav/embedded-design-tools.html

2. Run the Petalinux install script, run the script as a regular user and complete the installation answering all the prompts.

   NOTE: Petalinux requires the settings.sh file to be sourced when used, after installing add this line to the bashrc file and restart the terminal.

   # source '<petalinux install path>/settings.sh'

3. Once Petalinux is installed, create the folder for the Zedboard hardware platform using the following command: petalinux-create --type project --template zynq --name <folder_name>

4. In Part 1 you created the hardware files for the Zedboard. Locate the .SDK folder in these files.

5. CD into the newly created petalinux folder and run the following command to configure the platform.

   petalinux-config --get-hw-description=<PATH TO HW .SDK FOLDER GENERATED IN VIVADO>

6. In the configuration menu, make sure the OpenCV libraries are selected to be built.
7. Make sure the subsystem AUTO hardware settings line is checked (-*- leave all other settings default.

8. Exit the configuration, select save when prompted to finish configuring.

9. Once the configuring has been completed, run the following command to build the platform.

   petalinux-build

   The result should fill the petalinux folder with the results of the build. The ...
   ../images/linux/sdk/sysroots file will be used in the next part.

Part 3: Adding Zedboard libraries to existing platform

1. Install SDx 2017.4. This will not work on some newer versions of Sdx.

2. Download and extract the zcu102_es2_reVISION package from Xilinx.

3. Now the Zcu102 libraries need to be replaced with the Zedboard libraries generated in Petalinux. To accomplish this run the following commands:

   cd <zcu102_es2_reVISION path>/sw/aarch64-linux-gnu/sysroot/

   cp -r lib lib64 ; cd lib


   cd ../usr

   cp -r lib lib64 ; cd lib

4. Finally, add the SYSROOT environment variable to bashrc.

    export SYSROOT='<zcu102_es2_rv_ss path>/sw/sysroot'

5. Download the 2017.4 release of xfopencv from

    https://github.com/Xilinx/xfopencv . Use this include library to build any
    xfopencv programs.

**Part 4 Build an Example OpenCV Program [25].**

1. To build any xfopencv examples, simply edit the Makefile. Change the line

    `PLATFORM = <path-to-reVISION-platform>` to `PLATFORM = zed`. Save the
    file and run make. The program will compile and an sd_card folder will be
    generated to run on the Zedboard.

2. Create an SD card with 2 partitions. The primary partition should be bootable and
   over 1 GB with FAT filesystem type [26]. The second partition can be any size
   and of ext4 filesystem type [26].

3. Copy the contents of the sd_card folder onto the bootable partition of the SD card.


5. Now the SD card is ready. Make sure the Zedboard is set to boot from the SD
   card, load in the SD card and turn on the Zedboard.
6. In order to interact with the Zedboard on a computer, you can connect using the screen command. Connect to a PC using the USB UART on the Zedboard and run the screen command: `sudo screen /dev/ttyACM0 115400`

7. If the Zedboard kernel does not boot first, press the ps-rst button on the Zedboard to reboot.

8. Once in the Zedboard, mount the second partition of the SD card using mnt.

9. Copy the contents of the lib to the lib on the Zedboard.

   ```bash
cp -r <mount point of 2\(^{nd}\) partition>/lib/* /lib
```

10. Set the library paths as follows

    ```bash
    LD_LIBRARY_PATH=/lib:<mount point of 2\(^{nd}\) partition>/usr/lib
    export LD_LIBRARY_PATH
    ```

11. The compiled OpenCV program should now be able to be run from on the Zedboard using the terminal.
Bibliography


4. Xilinx SDSOC Development Environment
   https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html


9. Xilinx, *Zynq-7000 SoC Data Sheet: Overview*, Xilinx Inc, 2018


