IMPLEMENTATION OF A POWER EFFICIENT
SYNTHETIC APERTURE RADAR BACK PROJECTION ALGORITHM
ON FPGAS USING OPENCL

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree of

Master of Science in Electrical Engineering

By

David Fan

UNIVERSITY OF DAYTON

Dayton, Ohio

August, 2015
ABSTRACT

IMPLEMENTATION OF A POWER EFFICIENT SYNTHETIC APERTURE RADAR BACK PROJECTION ALGORITHM ON FPGAS USING OPENCL

Name: Fan, David
University of Dayton
Advisor: Dr. Eric Balster

In this thesis, an implementation of a Synthetic Aperture Radar (SAR) back projection algorithm onto a Field-Programmable Gate Array (FPGA) device using Open Computing Language (OpenCL) is developed. SAR back projection is a method to form a high-resolution terrain image from radar data. SAR is used in many applications such as Geographic Information Systems (GIS), crop classification, oceanography, and much more. FPGA devices are normally written in a Hardware Description Language (HDL) to create highly parallel and optimized solutions. The FPGA device solution in this thesis is written in OpenCL, an open-source framework for heterogeneous computing devices including CPUs, GPUs, DSPs, and FPGAs. OpenCL provides a new method for programming FPGA devices using a familiar programming environment. OpenCL programming follows the C99 Standard and provides support for devices with parallel computing capabilities and leverages these capabilities in order to provide an optimized solution. A parallel implementation of a SAR back projection algorithm on FPGA is presented here and is profiled for timing, accuracy, and logic utilization to compare performance against CPU and GPU based design implementations. The presented OpenCL and FPGA back projection solution provides a throughput that is 18.7 and 2.2 times efficient in terms of power compared to CPU and GPU solutions.
To my parents, my brother, and Sarah with love.
ACKNOWLEDGMENTS

I would like to thank everyone for their support, especially following people:

- **Eric Balster**: For providing me this opportunity in my graduate career and for consistently bugging me about my thesis, allowing me to become a well-rounded professional.

- **Kerry Hill and Alfred Scarpelli**: For their support in investment and funding into UDRI, FPGAs, and OpenCL projects.

- **LeRoy A. Gorham and Linda J. Moore**: For their interest and support in my project for funding FPGAs and OpenCL SAR development and acceleration.

- **Jonathan Skeans**: For his extremely helpful experience, insight, humor, and support/aid in the OpenCL projects of which I would not be able to complete this thesis without his help.

- **Andrew Kordik**: For his knowledgeable contribution to my understanding of computer architecture, programming, SAR, and back projection.

- **David Mundy**: For his insightful viewpoints and experience in FPGAs, SAR, and back projection.

- **Daniel Shaffer**: For building a beast/Frankenstein of a server to compile all my FPGA OpenCL builds.

- **Bryan Badalamenti**: For his support in reminding me of things I needed do of which I would most likely entirely forget or leave until the last minute along my graduate career.

- **Frank Scarpino**: For his knowledge in SAR back projection processing and for being on my committee.
• **AFRL/UDRI/ADDA (Lab):** I would like to thank the Air Force Research Laboratory and everyone in UDRI/ADDA (Lab) for providing a fun and humorous environment to do research and work.

• **Cheng Fan and Qiu Ye:**

  For my parents. I would not be able to make it where I am now without them.

• **Phong Fan:** For my brother, for keeping me sane throughout this experience.

• **Mariana/Carla Aboujaoude, Howard Poston, Tom Sharp, John Connor, Seng Wu:**

  For my friends, for keeping me insane throughout this experience.

• **Sarah Struckman:**

  For always being there for me. This journey would not be worth it without you.
TABLE OF CONTENTS

ABSTRACT ................................................................. iii
DEDICATION ........................................................... iv
ACKNOWLEDGMENTS .................................................... v
LIST OF FIGURES ....................................................... ix
LIST OF TABLES ........................................................ xii

I. INTRODUCTION .................................................... 1

II. RADAR AND SAR BASICS ........................................ 4
  2.1 RAdio Detection and Ranging (Radar) ....................... 4
    2.1.1 Point Targets .............................................. 4
    2.1.2 Range ...................................................... 5
    2.1.3 Pulse Compression ........................................ 6
  2.2 SAR .............................................................. 9
    2.2.1 SAR Background .......................................... 9
    2.2.2 SAR Flight-path ......................................... 11
    2.2.3 SAR Phase History Data ................................. 11

III. BACK PROJECTION ............................................... 13
  3.1 Back Projection Processing Chain ........................... 13
    3.1.1 Differential Range Calculation ......................... 14
    3.1.2 Range Lookup ............................................ 15
    3.1.3 Phase Correction Coefficient Calculation .............. 17
    3.1.4 Image Integration ....................................... 19

IV. THE OPENCL STANDARD ......................................... 21
  4.1 OpenCL Background ........................................... 21
    4.1.1 OpenCL Host Programming ............................... 22
LIST OF FIGURES

2.1 Pulse-delay Ranging .............................................. 5
2.2 Pulse Width and Band Width Relationship .............................. 6
2.3 LFM (Chirp) Signal ................................................ 7
2.4 Returned Chirp Signal ............................................. 7
2.5 Pulse Compression Process ........................................... 8
2.6 SAR Frequency Bands ............................................... 9
2.7 SAR Processing Chain ............................................... 10
2.8 SAR Resolution ....................................................... 10
2.9 Strip-Map and Spotlight SAR ....................................... 12
2.10 SAR Range Bins .................................................... 12
3.1 Original and Color Enhanced Back Projected Image ................. 14
3.2 Back Projection Processing Chain .................................... 14
3.3 Differential Range Distance Plot ................................... 15
3.4 Original (Left) and Colored (Right) Non-Phase Corrected Image .... 18
3.5 Processing 1 Pulse [Pulse Before Integration (Top), Pulse After Integration (Bottom)] 19
3.6 Processing 50 Pulses [Pulse Before Integration (Top), Pulse After Integration (Bottom)] ........................................ 20
4.1 OpenCL 1 and 2 Dimensional Ranges ................................ 23
LIST OF TABLES

5.1 Bittware Specifications ............................................. 30

6.1 CPU and GPU Specifications ................................. 49

6.2 Baseline Back Projection Results ............................. 49
CHAPTER I

INTRODUCTION

The use of radar has been used in many areas of surveillance such as meteorology, geology, agriculture, hydrology, climatology, and others [1]. Radar uses electromagnetic waves to locate particular objects through echolocation, the transmit and receive delay of a particular signal to measure an object’s distance. The echo returns of an object are caused by its reflectivity and provides information about the intensity of the reflectors. Many radars are stationary and are focused in a particular direction to detect objects within a given distance. A particular application in radar known as Synthetic Aperture Radar (SAR) processing offers a methodology for mapping an image of the terrain through the processing of echo returns from ground objects. SAR was first conceptualized in 1951 and tested the following year [1]. The first space-borne SAR was launched in 1978 used for earth observation aboard the SEASAT satellite [1]. Since the first missions of SAR, SAR technology has grown into a highly developed and studied topic. To obtain the ability to receive echo returns from a given terrain, the SAR radar is placed on a moving airborne platform and focused at a point on the ground. The flight path of the moving platform can be manipulated multiple ways to benefit the image quality and/or the mapped scene. In this thesis, we examine Spotlight SAR for maximizing the terrain image resolution and alleviating some of the computational load by taking assumptions from the flight path pattern. In Spotlight SAR, the flight path of the airborne platform circulates a point on the ground to image one stationary area of the terrain. By exploiting the
positioning of the airborne platform, the location and reflected intensity of the returned object, an image can be formed. SAR processing, due to its nature, requires a large amount of computation of signal processing to generate an image of the terrain in high-resolution. In addition, as the image size grows, the computational complexity increases. The process within SAR processing to turn the SAR data into an image is called back projection which is computationally expensive. However, the heavy computational challenge provided by the back projection algorithm is accelerated through the emergence of parallel processing devices providing a better real-time solution for SAR image generation. Back projection is a widely written topic and much research has been done to implement a Fast-Back projection algorithm on highly parallel architectures, most notably on General Purpose Graphics Processing Units (GPGPUs/GPUs) and Field Programmable Gate Arrays (FPGAs) [2, 3, 4].

GPUs currently are widely dominated by NVIDIA’s CUDA programming language for parallel processing on NVIDIA architectures [4, 5]. FPGA technology is programmed primarily through a Hardware Description Language (HDL) such as Verilog or Very High Speed Integrated Circuit (VHSIC) HDL (VHDL) which allows for parallel programming. OpenCL is a newly released framework designed to program supported heterogeneous devices and largely bases many of its concepts from that of the CUDA framework. However, unlike CUDA, OpenCL support is available on a wide array of supported devices including CPUs, GPUs, FPGAs, and Many Integrated Core (MIC) processors. In this thesis, an implementation of a back projection algorithm is accelerated on an FPGA device using OpenCL for acceleration. In dealing with FPGA technology, an evaluation of logic utilization is analyzed as FPGA devices are designed to be programmable hardware logic. The presented OpenCL and FPGA back projection solution provides a throughput that is 18.7 and 2.2 times efficient compared to CPU and GPU solution.
The remainder of this thesis contains a total of seven chapters including the Introduction. Following the Introduction, background information is covered reviewing radar and SAR basics in Chapter 2, the back projection process in Chapter 3, The OpenCL Standard in Chapter 4, implementation of SAR back projection on FPGAs in Chapter 5, performance results of the FPGA implementation of SAR back projection in Chapter 6, future improvements and recommendations in Chapter 7, and lastly the final conclusion of this thesis in Chapter 8.
CHAPTER II

RADAR AND SAR BASICS

To serve as a background, this Chapter reviews radar and SAR basics, allowing for a common baseline to layout the groundwork for the remaining chapters. Basic concepts of SAR and the SAR processing chain are covered providing a background in SAR technology and terminology.

2.1 RAdio Detection and Ranging (Radar)

Radar is the use of radio waves to measure targets within a given range. Objects such as aircrafts, ships, vehicles, and buildings reflect radio waves similarly as they do within the visible light spectrum. Radar can be used to differentiate one target from another by the returns it receives back from a target. Radar can be used to determine the distance of an object, and determine and object’s speed. In the following sections, background information on the signal formation process and pre-processing pertaining to radar are described which are applicable to SAR processing.

2.1.1 Point Targets

A point target is considered as a detectable object on the terrain’s surface. An object that is separable from the background will have a radar cross section (RCS) in which an imaging radar can measure and reconstruct the object, in the case of SAR, onto a pixel or group of pixels on the ground. The RCS can be seen as a product of three factors, the geometric cross section, the reflectivity, and
the directivity. The three factors describe the object’s cross sectional area, re-radiated power from interception, and the direction of the back scatter of the return [6].

2.1.2 Range

The range of a target can be determined by measuring the time it takes the radio waves to reach the target and return. Radio waves travel at the speed of light, resulting in the target being half of the time it takes to transmit the signal for a round trip transmission. This method is called pulse-delay ranging which determines the distance of a target by the delay of a transmit and receive signal given by Equation 2.1 [7].

\[ R = \frac{ct}{2} \]  

(2.1)

Figure 2.1: Pulse-delay Ranging

Pulse-delay ranging can be shown by in Figure 2.1. In Equation 2.1, \( R \) is taken to be the range, \( c \) is the speed of light, and \( t \) is the round-trip transit time of the pulse (pulse width). Bandwidth, \( B \) is inversely proportional to the pulse width allowing the form to be rearranged. Equation 2.1 provides the range resolution.
2.1.3 Pulse Compression

When considering the pulse-delay range shown in Equation 2.1, if multiple targets are closely spaced, they will be overlapping and returned as a single target. The width of the pulse, \( t \) must be short so the pulses shown in Figure 2.1 will increase the resolution and allow for targets to be distinguishable. However, by shortening the pulse width, the bandwidth increases as shown in Figure 2.2. This result, shown in Figure 2.2, is due to the bandwidth being inversely proportional to the pulse width [8].

![Pulse Width and Band Width Relationship](image)

A problem presents itself here by the fact that a large amount of power is required to obtain a higher resolution. To overcome this problem, pulse compression is employed to improve the resolution without having to use a large amount of power. Pulse compression employs a Linear Frequency Modulated (LFM) Waveform, also known as a chirp, which increases or decreases the frequency with time [8]. The chirp signal can be shown in Figure 2.3 below.
Figure 2.3: LFM (Chirp) Signal

Figure 2.4: Returned Chirp Signal
By using a chirp, the received signals are easily distinguishable between targets by the target returns as shown in Figure 2.4. To deploy this process, a signal is generated and applied the carrier frequency to upscale the generated chirp. Once the carrier frequency has been applied, it is transmitted and received where the received signal is then mixed with a carrier frequency to downscale the signal[8]. The resulting sum and difference frequencies are then processed through a low-pass filter (LPF), deramped, digitized through an analog-to-digital (A/D) converter, and then processed through real-time processors. The LPF rejects the sum-frequencies and retains the difference frequencies to obtain the proper return signal [9]. The deramp process involves a mixing of the return signal with the reference signal and a deskew operation. The time domain is converted into frequency domain through the use of the Fast-Fourier Transform (FFT). The processed data is then sent to a computer processor to allow for exploitation into topics such as detection, estimation, and tracking. The pulse compression process is shown in Figure 2.5.
2.2 SAR

SAR processing is a special field of radar due to objective to create an image of the earth’s surface using radar. To generate an image, a radar is placed on an airborne platform which transmits and receives echo returns while the airborne platform is moving [7]. The movement of the airborne platform provides the second dimensional result for the image. In this section we cover the background of SAR and SAR data.

2.2.1 SAR Background

SAR is an terrain imaging system that generates high resolution images using radar. There are many benefits to using SAR, the main advantages are that SAR can operate at long ranges, and can create images in nearly any weather conditions. Using an RF system enables these capabilities which allow for the general use of SAR in many situations. SAR radar operate on various frequencies bands shown in Figure 2.6.

![SAR Frequency Bands](image)

Figure 2.6: SAR Frequency Bands

The SAR processing chain in its simplest form can be shown in Figure 2.7 consisting of obtaining the phase history, applying pulse compression, applying the FFT, and then image formation using back projection to generate a SAR image. The airborne platform contains a single radar and,
combined with the movement of the platform, creates a synthetic aperture. The movement of the plane introduces a resolution in the cross-range resolution. The cross-range resolution, $\delta_x$, within SAR becomes Equation 2.2 where $D$ is the size of the aperture, $R$ is the range, $\lambda$ is the wavelength, and $\beta$ is the beam width as shown in Figure 2.8.

$$\delta_x = \beta R = \frac{\lambda R}{D}$$  (2.2)

From Equation 2.2, having a large aperture $D$, will result in a loss in resolution. The on-board radar then must have a small aperture size to get a larger beam width $\beta$. 
2.2.2 SAR Flight-path

The two highly popular flight-paths of the airborne platform are known as Strip-map SAR and Spotlight SAR. In Strip-map SAR, the radar does not move its point of focus on the ground, however the airborne platform does. The Strip-map SAR method therefore generates strips of images on the ground. The second flight-path is Spotlight SAR where the radar remains focused on a single area on the ground and generally circles the area. To keep the focus of the radar on at the same point on the ground, four different methods can be used. The four methods include flight-path manipulation, mechanically steered array, electronically steered array, and digital spotlighting. Flight-path manipulation allows the airborne platform to control the radar beam such as flying in a circle. Mechanically and electronically steered arrays are methods to move the radar beam on-board the airborne platform such that it remains focused at the same spot. Digital spotlighting allows for multiple views through the segmentation of a much larger view. Spotlight SAR allows for a much higher resolution image by obtaining multiple views of the same area on the ground. To merge the images together, a method of back projection is used. This thesis focus of Spotlight SAR is for generating high resolution images. An example of these two SAR flight-paths are shown in Figure 2.9.

2.2.3 SAR Phase History Data

SAR phase history data is stored into a grid of range bins where the range is stored in row major form. The SAR phase history data obtains echo returns and stores them within the range bins such that they are stored within memory [7]. This is represented in Figure 2.10. In SAR, a pulse is a collection of echo returns to generate a single image. Once the bins have been stored within memory, they can be back projected to form an image.
Figure 2.9: Strip-Map and Spotlight SAR

Figure 2.10: SAR Range Bins
CHAPTER III

BACK PROJECTION

Back projection is an algorithm to convert SAR phase history data into an image. Back projection is able to build images pulse by pulse and integrate the new information as it is received [10]. The back projection algorithm is computationally expensive as it has a complexity of $O(N^3)$ for an $N \times N$ SAR image [10]. The algorithm itself does lend itself to parallel computing making it a potential area to optimize using parallel architectures [11, 12]. The algorithm uses phase history data which is generated from the pulse compression. In this Chapter, we cover the basic processing chain the of back projection used to generate an image. A sample centered point target image is shown in Figure 3.1.

3.1 Back Projection Processing Chain

The back projection algorithm builds an image within four steps. The four steps are a differential range calculation, range lookup, phase correction calculation, and image integration. This section covers the four steps of back projection to generate a SAR image. The back projection processing chain is formed using the four following steps shown in Figure 3.2.
3.1.1 Differential Range Calculation

The differential range calculation generates a canvas to which the image is to be plotted onto and calculates a three dimensional euclidean distance of the sensor and pixel minus the distance to scene center. The equation of the differential range calculation is given in Equation 3.1.

\[
\Delta R = \sqrt{(x_{\text{sensor}} - x_{\text{pixel}})^2 + (y_{\text{sensor}} - y_{\text{pixel}})^2 + (z_{\text{sensor}} - z_{\text{pixel}})^2} - R_K \tag{3.1}
\]

Equation 3.1 takes the position of the sensor \((x_{\text{sensor}}, y_{\text{sensor}}, z_{\text{sensor}})\) and calculates the distance to each pixel \((x_{\text{pixel}}, y_{\text{pixel}}, z_{\text{pixel}})\) for a given image. The calculated distance is then subtracted by the distance to scene center, \(R_K\), at a given pulse \(K\). The distance to scene center allows
the image to be centered at zero. Once the distance has been calculated, the distances for each pixel at every position of the sensor is known, allowing the energy at that the pixel to be plotted. Figure 3.3 shows the distances of each pixel at different sensor locations. Each pixel is computed independently from one another, this allows for all the calculations to be done in parallel.

![Figure 3.3: Differential Range Distance Plot](image)

### 3.1.2 Range Lookup

The range lookup plots the energy, given from the phase history, at a given pixel provided by the differential range. However, since the phase history data does not line up with the pixel locations provided by the differential range calculation, since they are discrete distances, a linear interpolation is used to fill in the gaps to generate the image [4]. The phase history data is complex and as a result, the resulting linear interpolation provides an image in complex form.

To implement a linear interpolation to generate points for the pixels provided by the differential range, two range bins from the phase history data are linearly interpolated to find the appropriate energy return at the location of the pixel. To compute the closest bin, the step size of the image scene in terms of the differential range can be calculated for each pulse $K$. Setting the frequency step size,
Δf, to match the size of the scene to be imaged, the bandwidth, B, given by Equation 3.2 where the range resolution, δr, is computed to be based on the frequency step size forming Equation 3.3. The maximum scene extent, Wr is then shown by Equation 3.4 [10].

\[ B = (K - 1) \Delta f \]  

\[ \delta_r = \frac{c}{2B} = \frac{c}{2(K - 1) \Delta f} \]  

\[ Wr = \frac{c}{2 \Delta f} \]  

The size of the range bins are equivalent to the number of FFT points, NFFT, which is used to calculate the range bin location. From the maximum scene extent, Wr, in Equation 3.4, the range bin index, indexi, for a given location from the differential range, dRi, can be calculated as shown in Equation 3.5.

\[ index_i = \frac{dR_i N_{FFT}}{Wr} + \frac{N_{FFT}}{2}; \]  

The result from Equation 3.5 is then interpolated using the range bins for a finer representation of the energy at the given distance. Taking the floor of indexi will provide the index for the first bin and the fractional part provides information on the slope needed to linearly interpolate the two bins, w0. The index to the range bins, rc, is accessed as the first bin, bini and the following bin is accessed for the second bin, bini+1. The interpolated point lies between these two bins. Once the two bins have been found, the linear interpolation can be computed using w0 to find the intermediate range, L, that lies between the two bins as shown in Equation 3.6 [13].
\[ w_0 = \text{index}_i - \lfloor \text{index}_i \rfloor \]
\[ \text{bin}_i = r \left( \lfloor \text{index}_i \rfloor \right) \]
\[ \text{bin}_{i+1} = r \left( \lfloor \text{index}_i \rfloor + 1 \right) \]
\[ L = (1 - w_0)\text{bin}_i + (w_0)\text{bin}_{i+1} \quad (3.6) \]

### 3.1.3 Phase Correction Coefficient Calculation

The phase correction coefficient calculation is used for obtaining the maximum energy for each pixel. The phase correction coefficient is applied after the range lookup which adjusts the energy to provide the largest magnitude on a target from a given pixel. The phase correction coefficient transforms the range lookup result through a rotation of the complex value in the real and imaginary plane such that each pixel represents the largest magnitude. The equation for calculating the phase correction coefficient, \( A \), is given in Equation 3.7. In Equation 3.7, \( f_k \) is the minimum frequency at pulse \( k \), \( \Delta R \) is the differential range, and \( c \) is the speed of light. By applying Euler's Equation shown in Equation 3.8, Equation 3.7 becomes Equation 3.9 which is needed to compute the real and imaginary parts on a processor in OpenCL.

\[ A = \exp \left( -j\pi f_k \Delta R \right) \quad (3.7) \]

\[ e^{-jx} = \cos (x) - j \sin (x) \quad (3.8) \]

\[ A = \cos \left( \frac{\pi f_k \Delta R}{c} \right) - j \sin \left( \frac{\pi f_k \Delta R}{c} \right) \quad (3.9) \]

\[ S = AL \quad (3.10) \]

The phase correction coefficient is applied by multiplying the phase correction result to the corresponding pixel obtained from the range lookup by Equation 3.10, to generate \( S \), the complex
image from a pulse. Applying the phase correction coefficient to the range lookup result requires a complex multiplication as shown in Equation 3.11.

\[
\begin{align*}
    u &= a + jb \\
    v &= c + jd \\
    uv &= (a + jb)(c + jd) \\
    &= ac + jbc + jad - bd \\
    &= (ac - bd) + j(bc + ad)
\end{align*}
\] (3.11)

Figure 3.4: Original (Left) and Colored (Right) Non-Phase Corrected Image

Figure 3.4 shows an image without phase correction applied. By applying phase correction, the image shown in Figure 3.1 is formed. The artifacts within the image can be further removed by applying windowing functions.
3.1.4 Image Integration

Image integration adds the corrected range lookup result to the previously plotted images. After the phase correction coefficient has been applied to the range lookup result, the new updated image for the given pulse data is created and can then be updated to the previous pulses. Updating the previous images is done by adding the newly created complex valued image to the previous images. By adding the new pulses to the previous pulse, the resolution of the image becomes clearer and continues to do so as more pulses are processed. To view the image stored as complex values, the absolute value of the image is taken and displayed. Integration can be seen as more pulses are added from the first pulse, shown in Figure 3.5, to the 50th pulse, shown in Figure 3.6, with the original image left and color added to the right for easier reference. Figure 3.5 and Figure 3.6 show that as more pulses are added, finer detail is achieved.

Figure 3.5: Processing 1 Pulse [Pulse Before Integration (Top), Pulse After Integration (Bottom)]
Figure 3.6: Processing 50 Pulses [Pulse Before Integration (Top), Pulse After Integration (Bottom)]
CHAPTER IV

THE OPENCL STANDARD

The OpenCL Standard provides a framework in which device vendors must implement specific capabilities defined by the standard to support OpenCL. The OpenCL framework is used to program across heterogeneous platforms providing support for task-based and data-based parallelism. The OpenCL Standard was originally developed by Apple Inc. in 2009 and later adopted by the Khronos Group. The OpenCL Standard follows the C99 programming standard and is heavily influenced by many concepts seen in CUDA programming for NVIDIA GPUs [14]. In this Chapter a review of OpenCL, parallel processing within OpenCL, and implementing OpenCL on different hardware is presented.

4.1 OpenCL Background

OpenCL is currently emerging as the first open-source languages used for parallel programming on multiple device platforms to increase the speed of deployed applications. The OpenCL Standard seeks to unify programming for multiple devices into a single language with a simplistic application program interface (API) to supply to the developer and enable their applications to easily migrate from one platform to another through its cross-platform support. The OpenCL Standard became a need for many developers that needed deploy their applications onto heterogeneous environments such as CPUs, GPUs, DSPs, and FPGAs. The idea of OpenCL is to "write once, run anywhere"
similar to the Java programming language but for supported OpenCL embedded devices. By allowing multiple platforms to run applications, the performance from using OpenCL allows a developer to accelerate code by distributing tasks to better suited architectures. Within The OpenCL Standard, many constructs from the CUDA programming language is used in structuring parallel executions on compute resources and memory space hierarchy [14]. OpenCL programming is structured into two parts, host programming and kernel programming. In host programming, a host device manages the distribution of tasks. In kernel programming, the actual tasks are computed and sent to other kernels or back to the host.

4.1.1 OpenCL Host Programming

OpenCL requires a host to manage and distribute tasks to different platforms and is primarily written in C. Many implementations use a CPU as a host to setup and transfer data to the various platforms. The host code manages and distributes tasks by initializing the number of work-items (threads) that are needed to be executed and setting up buffers to transfer data to on-board memory of the device. Scheduling of the kernels can be done within the host through the implementation of a wait_list when executing kernels. Thread groups are known as work-groups which are a collection of groups that have shared memory in the local memory space. An N-dimensional range can be defined when setting up work-items and work-groups similar to CUDA threads and CUDA blocks as shown in Figure 4.1 [15, 16].

To setup an OpenCL host to initialize an OpenCL kernel, the host needs to call specific functions to the OpenCL API using the OpenCL Source Development Kit (SDK) from the vendors of all the devices the application runs on. From within the host, six OpenCL structures are defined to setup sending commands to the device. The six structures are: platforms, devices, contexts, queues, programs, and kernels [16] Platforms contains devices which are separated by vendors such as Intel, AMD, NVIDIA, and Altera. Devices are the handles to the physical device by the supplied
platform. Context is a work-space of a collection of devices and programs that are used to process a task. Queues are methods of transferring commands and data to and from the host. Programs are a series of kernels to run on a device. The kernel is a single function that is to be executed on a device.

To communicate and send commands to the devices, a queue is used to issue tasks to each device from the host. Tasks include writing buffers, reading buffers, and launching kernels to any given device in a context [16]. The host model for setting up OpenCL structures are shown below in Figure 4.2.

4.1.2 OpenCL Kernel Programming

OpenCL kernels are the functions which are executed on the device. OpenCL follows the C99 programming standard when writing an OpenCL kernel in addition to more features to support parallel capabilities. Additional features include vector math, barriers, and different memory spaces [16]. OpenCL defines four different memory spaces known as global, constant, local, and private memory given in the order of slowest to fastest memory access [17]. Vector math allows for operations on arrays to be represented simpler, cleaner, and faster. Vectors sizes, however, must be a certain length to provide optimizations by operating on each element at the same time [16].
4.1.3 OpenCL Memory Model

Memory access is a huge bottleneck in many of today’s processors [18]. As a result, it is important to understand the memory model within OpenCL to achieve the best performance from parallel computing. These memory types are accessed from slowest to fastest in the given order and are shared differently among work-items. Global memory is the largest memory space and can be accessed anywhere, however it is the slowest memory space. Reducing the calls to global
memory will greatly optimize a kernel. Constant memory is placed into global memory, however it provides some benefits given that the memory is constant and does not change. Local memory is the memory shared between work-items within a work-group but not other work-groups. Local memory provides the next largest memory space following global and constant memory with the second fastest memory access capability. Private memory is the work-item’s memory space only accessible by the work-item itself. Private memory has the smallest amount of memory space and is the fastest memory space accessible by a work-item. The OpenCL memory model described can be shown in Figure 4.3 [16].

![OpenCL Memory Model](image)

Figure 4.3: OpenCL Memory Model

### 4.2 CPUs, GPUs, and FPGAs

The most commonly seen hardware used today include the use of CPUs, GPUs, and FPGAs to process data. CPUs offer a few number of highly optimized cores. GPUs offers many non-optimized cores for parallel computing. FPGAs provides customizable programmable logic to generate application specific cores, allowing for the capability to generate multiple optimized cores as long as the generated hardware logic can fit within the FPGA fabric [19].
CPUs and GPUs are comprised of many general purpose processing elements and OpenCL uses them to process given tasks. However, FPGAs require the host to load the hardware logic through a binary file which needs to be pre-compiled before execution. The logic built in FPGA hardware is representative of a core processor and pipeline. Programming OpenCL for GPU/CPU and FPGA devices provide fully functional implementations, however, they are optimized differently as CPU and GPU devices apply data parallelism and FPGAs provide task parallelism. In data parallelism, the same operations are executed on multiple sets of data simultaneously. In task parallelism, different operations are executed on the same or multiple sets of data simultaneously as it is streamed through a pipeline architecture. Since FPGAs operate on task parallelism, an optimized FPGA kernel require specialized programming and a careful attention to implementation when writing the OpenCL kernel [19]. Careful programming in OpenCL is needed in FPGA task parallelism to primarily avoid data and control hazards as they add additional logic to the pipeline and unnecessary clock cycles to the processing time.

OpenCL provides support for extensions. Extensions provide support for vendor specific capabilities that are not defined within the OpenCL standard [16]. The Khronos Group has specified some optional extensions which devices generally support including support for half, double, and quad precision specifications. Extensions allow for vendors to implement support for device specific capabilities such as OpenGL in which other vendors do not support for specific devices.

4.3 The IEEE-754 Floating Point Standard

OpenCL supported devices are required to support a modified IEEE-754 standard [20]. It is important to know these differences when implementing when dealing with floating point operations. As a result from these differences, the OpenCL Standard does not provide an exact match to that
of a CPU but requires certain floating point properties to ensure a degree of precision. The advantages of OpenCL implementing these differences allow for faster methods of computation which are more advantageous to parallel devices. The OpenCL standard requires support for 32-bit float type shown in Figure 4.4. Figure 4.4 represents a 32-bit floating point value storing 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa to generate a floating point value. Support for 64-bit double type and 16-bit half type are available through the use of extensions and are optional and device variant. The IEEE-754 standard defines a float within four categories, normal numbers, denormalized numbers, infinite numbers, and not a number. In OpenCL, devices are not required to support denormalized numbers. Denormalized numbers are numbers smaller in magnitude than the smallest possible normal number that can be represented. Denormalized numbers are useful when subtracting two numbers that will produce a denormalized number instead of zero which can be used for division rather than an invalid result from dividing by zero [16].

Rounding is treated differently in OpenCL than from the IEEE-754 standard. The IEEE-754 standard defines four different modes of rounding floats. The four rounding modes are rounding to the nearest even, towards positive infinity, towards negative infinity, and towards zero. OpenCL only requires support for the first mode, rounding to the nearest even where rounding between two
numbers; the one whose lowest order digit is even is selected. OpenCL requires only particular specifications defined within the IEEE-754 standard, however, vendors may provide support for other specifications defined by the standard. These specification requirements change if the half or double extension is used and is supported by the device.
CHAPTER V

FPGA IMPLEMENTATION OF BACK PROJECTION

In this Chapter, the back projection algorithm implemented on an Altera FPGA device using OpenCL is presented. The implemented algorithm is divided into three steps: the differential range calculation, the range lookup, and the phase correction calculation. A method to merge the three back projection steps is then presented with additional logic to reduce the access to global memory. The image integration step is appended to the end as the last step requiring only a multiplication and addition of two complex values.

Additionally, as OpenCL follows many concepts of CUDA, an initial test run converting the CUDA code directly into OpenCL is tested to view how well the CUDA implementation of the code runs on the FPGA board.

5.1 FPGA Back Projection Setup

The FPGA implementation of back projection divides back projection into three steps, differential range calculation, range lookup, and phase correction. Each step is optimized individually and then combined in the end to produce an optimized result. Altera is the first FPGA company to support OpenCL making it the valid candidate to test. Altera has started to fully support the OpenCL specification as of May 2013 for FPGA devices. The targeted FPGA device used for testing is the Bittware Inc S5PHQ_d5 board, supporting OpenCL version 1.0 on Altera SDK 14.1.1 and
Quartus 14.1.1. Figure 5.1 shows the FPGA card used for testing. The Bittware specifications are given in Table 5.1 [21].

<table>
<thead>
<tr>
<th>Type</th>
<th>Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>Straix V GX/GS</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>710 MHz</td>
</tr>
<tr>
<td>Logic Elements</td>
<td>952,000</td>
</tr>
<tr>
<td>Power</td>
<td>75 W</td>
</tr>
</tbody>
</table>

The FPGA implementation utilizes OpenCL to program the board using the *Altera Offline Compiler* (AOC) [19]. The AOC is used to generate a hardware configuration binary for the targeted
Altera FPGA device which is setup through the host program by The OpenCL Standard API. In designing the kernel for the FPGA, the Altera FPGA OpenCL design flow shown in Figure 5.2 is presented as documented in the *Altera OpenCL Programming Guide* [19].

The multi-step design flow in Figure 5.2 utilizes multiple Altera OpenCL tools to provide functionality, emulation, and profiling support. Support for functionality and emulation is needed because hardware configuration generation takes hours to complete, making development time scarce when testing. At this time of writing, the Altera tools are still under development but provide primary functional support. The emulation tools create a prototype of the kernel which simulates the computation of the FPGA board taking only a few minutes to build and test. The emulation ensures the kernel’s correctness before building the actual compile. The AOC tools provide an intermediate estimate of logic utilization to ensure the kernel is able to fit on the board. Altera provides a profiler tool, still under development, which provides information on the bottlenecks that are presented within the kernel. Building the kernel supplies a notices on task parallelism and ensures that an efficient pipeline is generated without read or write stalls. The remaining sections in this chapter cover the implementation and optimization the three-step back projection.
Figure 5.2: Altera FPGA Multi-Step AOCL Design Flow
5.2 Differential Range Calculation

From the differential range calculation provided by Equation 3.1, a block diagram of the FPGA logic design can be inferred as shown in Figure 5.3. The differential range calculation calculates the distance of each pixel \((x_{\text{pixel}}, y_{\text{pixel}}, z_{\text{pixel}})\) to the sensor \((x_{\text{sensor}}, y_{\text{sensor}}, z_{\text{sensor}})\) and stores it to the output \((\Delta R)\). The multiplication and square root functions consume more logic than the adders to compute the differential range. Additionally, the mathematical order of operations from the equation dictates the order of operations to provide a correct result.

Figure 5.3: Block Diagram of the Differential Range Calculation
To optimize the differential range calculation, multiple operations are tested. Firstly the multiplication function and the power function defined within OpenCL are measured in terms of logic utilization and timing to provide the best result. The next area of optimization is paralleling the operations among work-items. Each work-item is set to process a different pixel where the whole image is treated as a 1-dimensional array as concatenated rows. After applying parallel operations, the math operations are vectorized using the OpenCL math vectors such that multiple pixels can be operated at a time for each work-item. Processing multiple pixels at a time requires three different vectors for each dimension, holding elements in powers of two up to sixteen.

5.3 Range Lookup

From the equations of the range lookup provided by Equation 3.5 and Equations 3.6, a block diagram of the FPGA logic design can be inferred as shown in Figure 5.4. In the block diagram, a series of multiplications with an addition are done to find the index to lookup into the range profiles, $RC$. In addition, the fractional part of the bin index is used to calculate the value for the linear interpolation calculated as $w_0$ in Equation 3.6. The range lookup inputs are a differential range buffer ($\Delta R$), the complex range profile buffers as real (rc_real) and imaginary (rc_imag) components, the inverse of the scene extent (Wr_inv), and the size of the range profiles which is equivalent to the number of FFT points (N_FFT). The range lookup output (range) is the interpolated output of the range profiles provided as complex values.

To optimize the range lookup, the operations are paralleled among work-items. Moving the range profiles from global memory into local memory helps speed up the lookup into the range profiles by reducing the overall number of calls to global memory.
Figure 5.4: Block Diagram of the Range Lookup
5.4 Phase Correction Coefficient Calculation

From the equation of the phase correction coefficient calculation shown in Equation 3.9, a block diagram of the FPGA logic design can be inferred as shown in Figure 5.5. In the block diagram, trigonometric functions, sine and cosine, are used to calculate the coefficient. The inputs are a buffer for the differential range (ΔR) and the minimum frequency (f_k). The phase corrected output (A) is twice the size as the differential range buffer to hold complex values (real and imaginary parts) for each pixel in the scene.

To optimize the phase correction coefficient calculation, the trigonometric sine and cosine can be implemented using a Look-Up Table (LUT). The LUT defines a sufficient number of points on the unit circle such that an index to a pre-calculated trigonometric value can be easily accessed. A LUT can be used to replace the trigonometric functions because they are periodic and repeat every

Figure 5.5: Block Diagram of the Phase Correction Coefficient Calculation
Using a LUT for a set predefined number of points on the unit circle introduces an error into the calculation. In implementing a LUT, additional memory will be used to store the LUT values and additional logic will needed to calculate the conversion between any value given in radians to an index value for the LUT. The LUT will be compared against the OpenCL trigonometric functions for accuracy and logic utilization.

5.5 Merging the Back Projection Steps

To combine the back projection steps, the differential range result is supplied to the range lookup and phase correction coefficient calculation steps, removing the access to global memory. The outputs to the phase correction coefficient calculation and the range lookup is replaced by the image integration step which multiplies the result of the phase correction and range lookup followed by an addition to update the previous images with the generated image.

The initial access to global memory in the differential range calculation can be replaced with a coordinate generator. The coordinate calculator creates an evenly spaced grid used in the differential range calculation for the pixel coordinates \((x_{\text{pixel}}, y_{\text{pixel}}, z_{\text{pixel}})\). The inputs will be constant values stored in private memory as image_width, image_size, x_step, y_step, x_start, and y_start. The parameters image_size and image_width provides the size of the image, where image_width is the number of pixels along the x-axis and image_size is the number of pixels in the entire image, starting at the locations from x_start and y_start. Each coordinate generated is x_step apart along the rows and y_step along the columns. By adding additional logic to generate these values, access to global memory can be removed by adding five additional parameters in private memory.
Figure 5.6: Block Diagram of an Overall Back Projector
The merged setup in Figure 5.6 provides support process a single pulse at a time and processes pixels using vectors. To further improve the overall performance, Altera specific extensions can be utilized to provide subsequent sensor positions to the differential range equation and to provide image integration in a separate kernel. Particularly the use of Altera’s FPGA channels, which allow for the transfer of data to kernels and synchronizing kernels with high efficiency and low latency. Channels currently allow kernels to communicate to each other using first-in-first-out (FIFO) buffers and does not require the host to coordinate the movement of data between kernels [19]. By implementing channels, the FIFO buffers would queue up with sensor locations allowing for a continuous stream of pulses to be processed. The resulting pulse image can be sent into another FIFO buffer to perform the image integration and write back the combined image of pulses to global memory.
CHAPTER VI

PERFORMANCE RESULTS

In this Chapter, the results of the FPGA implementation of back projection using OpenCL is presented. The performance of the three step back projection setup presented in Chapter V is profiled based on its timing results and logic utilization. The individual back projection steps are tested over 1000 pulses and an image size of 1024x1024 pixels. After individual testing, the back projection steps are combined together for testing and are expanded to fill up the maximum logic utilization permitted by the FPGA to provide the maximum throughput from the FPGA device. To measure the efficiency of the throughput, the energy consumption for the CPU, GPU, and FPGA are calculated and compared. Lastly, different image sizes are tested and their corresponding performance and energy efficiency are measured.

Performance results are calculated in terms of the pulse repetition frequency (PRF). The PRF for a constant image size is calculated by Equation 6.1. A higher PRF indicates a faster throughput performance.

\[
PRF = \frac{Pulses\ Processed}{Total\ Processing\ Time} \quad (6.1)
\]
6.1 Differential Range Calculation Results

By implementing the setup and optimizations described by Section 5.2, the following results are measured. Firstly, multiplication and the power function are tested in OpenCL to compare which method is the most efficient in logic utilization and timing. The results of the comparison is shown in Figure 6.1 and Figure 6.2 which compare the logic utilization and timing results respectively.

From analyzing the results in Figure 6.1, use of the multiplication consumes 10% less logic utilization, 4% less dedicated logic registers, 5% less memory blocks, and 8% less DSP blocks. From Figure 6.2, multiplication has a PRF of 20 Hz greater than using the OpenCL power function. Overall, the use of the multiplication consumes less logic and requires less time to process, making it the preferred method to compute the differential range equation given in Equation 3.1.
The next optimization step considers the implementation processing multiple pixel locations at a time using work-items and vectors. The work-item implementation divides the task into multiple work-items that are non-vectorized. Vectors are tested at various sizes for storing and processing multiple locations in parallel for each work-item. Three different vectors are used to store each dimensional value for processing. The results are shown in Figure 6.3 and Figure 6.4.

From the results shown in Figure 6.3, the logic utilization gradually increases as the sizes of the vectors increase. In Figure 6.4, the timing improves as the vector sizes increases and evens out to around a PRF of 900 Hz when using a vector of size four or greater. Of the results, implementation using a vector of floats of size 4 provides the most efficiency in timing and hardware utilization as the timing results do not significantly improve when using a size greater than four while the hardware utilization continues to increase as the vector sizes increases.
Figure 6.3: Differential Range FPGA Fabric

Figure 6.4: FPGA Differential Range Multiplication vs Power Function PRF
6.2 Range Lookup Results

By implementing the setup and optimizations described by Section 5.3, the PRF and FPGA fabric is measured in Figure 6.5 and Figure 6.6.

![FPGA Range Lookup PRF](image)

Figure 6.5: FPGA Range Lookup PRF

From the results shown in Figure 6.5 and Figure 6.6, the timing remains to be consistent around a PRF of 650 Hz for each range lookup operation. The FPGA fabric increases as the vector sizes increase. The FPGA fabric when using a vector size of 16 uses 103% of the logic which is unable to fit and run on the bittware board, as a result, a PRF could not be obtained for a vector size of 16. The most inefficient process in this step is the number of reads to global memory for the required lookup to obtain an interpolated result. The PRF of range lookup begins to degrade as additional calls to global memory is made.
6.3 Phase Correction Coefficient Calculation Results

By implementing the setup and optimizations described by Section 5.4, the following results are measured. A comparison of the OpenCL trigonometric functions and LUT are tested for timing, logic utilization, and error at various vector sizes. Error is measured as signal-to-noise ratio (SNR) as provided by Equation 6.2 and further applied in decibels (dB) provided by Equation 6.3. The timing results for the phase correction are shown in Figure 6.7.

\[
SNR = \frac{\sigma_{signal}^2}{\sigma_{noise}^2}
\]  

(6.2)

\[
SNR_{dB} = 10 \log_{10} \left( \frac{\sigma_{signal}^2}{\sigma_{noise}^2} \right)^2
\]  

(6.3)
Figure 6.7: FPGA Phase Correction Math Functions vs LUT PRF

Figure 6.8: FPGA Phase Correction Math Functions vs LUT Signal-to-Noise Ratio (SNR)
Figure 6.7 represents the PRF to calculate the phase correction coefficient by two methods of calculation. The first method uses the OpenCL built-in math functions and the alternative method utilizes a sine LUT. The LUT requires processing time to calculate the correct index into the LUT. The implementation of the LUT provides a table of 1024 points from the unit circle defined within the sine function and a cosine function as an shifted offset of the sine function. From the timing results, the LUT provides competitive results when testing against the OpenCL trigonometric functions. Using a LUT creates and error in the calculation which is analyzed in Figure 6.8.

The measured error shown in Figure 6.8 indicates that the trigonometric functions have far more accuracy than the LUT implementation of 1024 points. Increasing the number of points on the LUT should increase the accuracy of the LUT if finer accuracy is needed. The FPGA fabric of the OpenCL trigonometric function and LUT are shown in Figure 6.9 and 6.10 respectively.

From the results shown by the logic utilization of Figure 6.9 and Figure 6.10, the trigonometric functions implementation consume a larger amount of resources as the vector size increases up to a maximum of 43% when using a vector size of 16. The LUT steadily increases at a slower rate than the trigonometric functions taking only a maximum of 30% when using a vector size of 16 which is 13% less logic than the trigonometric functions at the cost of accuracy.
Figure 6.9: Phase Correction Math Functions FPGA Fabric

Figure 6.10: Phase Correction LUT FPGA Fabric

48
6.4 Baseline CPU and GPU Spotlight SAR Back Projection Results

The implementation of back projection presented has been implemented on CPU and GPU processors written in C++ and CUDA [5, 13]. To present a baseline for throughput of the back projection algorithm, an Intel Xeon CPU E5-2670v2 and a NVIDIA Tesla K40m is tested. Table 6.1 shows the specifications of the tested hardware [22, 23].

Table 6.1: CPU and GPU Specifications

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed</th>
<th># Cores</th>
<th>Memory Bandwidth</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon CPU E5-2670v2</td>
<td>2.5 GHz</td>
<td>10</td>
<td>59.7 GB/s</td>
<td>115 W</td>
</tr>
<tr>
<td>NVIDIA Tesla K40m</td>
<td>745 MHz</td>
<td>2880</td>
<td>288 GB/s</td>
<td>235 W</td>
</tr>
</tbody>
</table>

To profile the code, each step of back projection is timed for throughput. To generate SAR data, a sample point target generator is used to idealize a point in space such that the back projection algorithm can process and create an image from. A 500x500 image is tested, profiling the time to process one pulse. The results of the baseline test is shown in Table 6.2.

Table 6.2: Baseline Back Projection Results

<table>
<thead>
<tr>
<th>Processor</th>
<th>PRF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon CPU E5-2670v2</td>
<td>26.9 Hz</td>
</tr>
<tr>
<td>NVIDIA Tesla K40m</td>
<td>4.7 kHz</td>
</tr>
</tbody>
</table>

The two solutions provided in Table 6.2 implement a single-threaded and multi-threaded solution for the CPU and GPU respectively. The multi-threaded solution is highly optimized within CUDA and operates at a PRF of 4.7 kHz [5, 13, 24, 25]. From the baseline results the CPU single threaded implementation provides the lowest PRF of 26.9 Hz. The resulting image from the back
projector is shown in Figure 6.11 where the sensor has completed a 360° flight-path over the scene. In Figure 6.11, color (right) has been added to the original image (left) to better view the image. From the resulting image shown in Figure 6.11, pulse artifacts remain in the resulting image as echos resonating from the single target in the center. Further signal processing can be applied to remove these artifacts by the use of windowing functions and expanding the FFT size before back projection.

![Baseline Original and Color Enhanced Back Projected Image](image)

**Figure 6.11: Baseline Original and Color Enhanced Back Projected Image**

### 6.5 OpenCL FPGA Spotlight SAR Back Projection Results

The individual back projection steps are combined together and are expanded to process one, two, four, and eight pixels in parallel. The merged implementation removes the calls to global memory, adds a coordinate calculator, and channel streaming the sensor location to the differential range calculation. The CUDA solution translated into OpenCL is also presented to evaluate performance of the GPU written code onto the FPGA. The developed back projector is used to process an 500x500 image over 1000 pulses from SAR data provided by a point target generator. The resulting image is shown in Figure 6.12 with color enhancement comparison in Figure 6.13. The difference
image from the reference to the OpenCL FPGA generated image is shown in Figure 6.14, containing an error of -140.079630 dB. The results of the FPGA back projection is shown in Figure 6.15 and Figure 6.16 for the kernel PRF and FPGA fabric consumption respectively. The PRF of the FPGA implemented solution is 124.6477 times faster than a CPU implementation and 1.4284 times slower than a highly optimized GPGPU implementation of the initial baseline results from Table 6.2. The logic utilization increases as more logic is used to process multiple pixels, increasing by 7% from 1 pixel to 2, 16% from 2 pixels to 4, and 37% from 4 pixels to 8. Respectively, the PRF increases by 726.3 Hz from 1 pixel to 2, 1 kHz from 2 pixels to 4, and 678.9 Hz from 4 pixels to 8. The results show that the PRF gain at the cost of FPGA fabric begins to level off after processing 4 pixels at a time. Moving from 4 pixels to 8 pixels adds 37% of additional logic for only a gain of 678.9 Hz PRF where 16% of additional logic provides a gain of 1054.9 Hz moving from 2 pixels to 4 pixels.
Figure 6.12: OpenCL FPGA Back Projection Original Image

Figure 6.13: Color Enhanced Back Projected Point Target Reference (Left) and FPGA (Right) Images
Figure 6.14: Back Projected Point Target Difference Image
Figure 6.15: FPGA Multi-Pixel Back Projection PRF

Figure 6.16: Multi-Pixel Back Projection FPGA Fabric
6.6 3D Volumetric SAR Data

The functional back projector is tested using the 3D Volumetric GOTCHA SAR Data [26]. The 3D Volumetric GOTCHA SAR Data is a set of collected SAR phase history data collected at X-band covering an urban environment. The original Volumetric SAR Data shown in Figure 6.17 is a 500x500 pixel image. The back projected reference image and FPGA image is shown in Figure 6.18 which have been color enhanced to better visualize the image magnitude. The difference between the reference and FPGA back projected image is shown in Figure 6.19 with an error of -76.365481 dB. The timing results of the Volumetric SAR Data is shown in Figure 6.20. From the timing results, the 8 Pixel Kernel provides the fastest result of 3 kHz, the 4 Pixel Kernel with 2.4 kHz, the 2 Pixel Kernel with 1.5 kHz, and the 1 Pixel Kernel with 900 Hz. The PRF increases 585.2443, 901.4726, and 577.8 Hz moving from 1 to 2 pixels, 2 to 4 pixels, and 4 to 8 pixels respectively where the largest increase is moving from processing 2 pixels to 4 pixels.
Figure 6.17: OpenCL GOTCHA Back Projection Original Image

Figure 6.18: Volumetric GOTCHA SAR Reference and FPGA Image
Figure 6.19: Back Projected Volumetric GOTCHA SAR Difference Image

Figure 6.20: FPGA Multi-Pixel Back Projection PRF on GOTCHA Volumetric SAR Data
6.7 Power

The power of which each device consumes are measured in joules using Equation 6.4 where $W$ is the device’s rated total dissipated power (TDP) specification and $t$ is the back projection kernel execution time.

\[ J = Wt \]  

(6.4)

The FPGA solution measuring the joules in addition to the CUDA GPU solution on the k40 presents results shown in Figure 6.21. From the power results given in Figure 6.21, the 8 Pixel Kernel provides the most efficient solution consuming 45.6% less joules (2.2 times more efficient) than the CUDA GPU solution. The next most efficient solution is provided by the 4 Pixel Kernel, consuming 57.5% less joules (1.7 times more efficient) than the CUDA GPU solution for each back projector. Additionally, the CPU, using a single core, consumes a large amount of energy using 0.43 joules for each back projector which is 18.7 and 14.8 times less efficient than the 8 Pixel Kernel and 4 Pixel Kernel respectively on the FPGA.

![CPU, GPU, and FPGA Multi-Pixel Back Projection Power Consumption](image)

*Figure 6.21: CPU, GPU, and FPGA Multi-Pixel Back Projection Power Consumption*
6.8 Processing Various Image Sizes

Various image sizes are tested using the CPU, GPU, and FPGA in terms of PRF and energy. The results can be seen in Figure 6.22 for the PRF as the image size increases. As the image size increases, the more pixels are needed to be processed thereby increasing the processing time and reducing the PRF. The image size scales will the processing time as image sizes of 500 x 500, 1k x 1k, 2k x 2k, and 4k x 4k images are processed. From the results the PRF is reduced by about a factor of 4 moving from one image size to the next. The reduction by a factor of 4 is expected as the previous image size is tiled four times to match the image size of the larger image, for example, 4 images of size 500 x 500 generates a 1k x 1k image. Similarly, the power consumption measured in joules increases by a factor of 4 as larger images are processed as shown in Figure 6.23.

![Figure 6.22: PRF (Hz) of Scaling Image Sizes](image)

Figure 6.22: PRF (Hz) of Scaling Image Sizes
Figure 6.23: Power Consumption (Joules) of Scaling Image Sizes
CHAPTER VII

FUTURE IMPROVEMENTS AND RECOMMENDATIONS

The presented OpenCL and FPGA back projection solution provides a throughput that is 18.7 and 2.2 times efficient compared to a CPU and GPU respectively. The provided implementation of back projection on FPGA devices with OpenCL development is still in an early stage of development and is capable of providing competitive performance and efficiency to that of a GPGPU.

There is still much room for improvement that can be done for the OpenCL FPGA solution of back projection as OpenCL on FPGAs is still an emerging technology. Investigating the range profile lookup in the interpolation can be improved to reduce calls to global memory provides an opportunity to greatly speed up throughput. Further improvements can be applied by processing multiple pulses in parallel and providing the resulting pulse image to the image integration kernel. Processing multiple pulses in parallel can be implemented by the use of multiple compute units of the back projector. To obtain more compute units, a larger amount of logic will be required to support the necessary back projection operations. An further analysis into bit precision can be investigated to provide a reduction in logic and an improved PRF. Altera channels provide areas for improvement with future updates provided by Altera to support the OpenCL specifications for implementing pipes as defined in a later specification of OpenCL.

Performance on later generations of FPGA boards can also be potentially utilized as they provide a larger set of programmable logic. Efforts made into reducing logic by bit reduction and expanding
the available compute units could be tested which would increase the throughput by a factor of the number of compute units. Reducing the bit sizes of the back projection calculations may introduce error which would need to be accounted for as the precision lowers.
CHAPTER VIII

CONCLUSION

The use of Synthetic Aperture Radar has been since the early 1950s and continue to expand to many applications today. However, due to the large amount of data needed to process the SAR data, new trends in technology can be used to address this issue. In the past few years, GPGPUs have been the main hardware device to process SAR data due to its ease in parallel programming. The OpenCL Standard released in 2009 provide a potential alternative to easily program other devices which can be used to process SAR data. The OpenCL Standard is a new technology that is capable of programming heterogeneous hardware devices which can be used for parallel processing. FPGAs is a highly parallel device which is capable of generating high throughput but requires a highly skilled engineer in a hardware description language and a significant amount of development time to program an FPGA device. This thesis presents the implementation of back projection, the process to generate an image from SAR data, using OpenCL to program an FPGA device. FPGA support for OpenCL have just recently been supported as of May 2013 and is still largely under development and improvement. Altera is the first group to support the OpenCL Standard on their newer FPGA devices.

The FPGA back projection solution in this thesis provides an optimized back projector that optimizes three computationally expensive operations on the FPGA and provides output that is 18.7 times more efficient than a CPU implementation and 2.2 times more efficient than a highly
optimized GPGPU implementation. The presented FPGA solution uses 99% of the FPGA fabric processing 8 simultaneous pixels at a time. The FPGA back projector is tested against a simulated point target generator and against the Volumetric GOTCHA SAR Data to simulate a practical implementation. As the OpenCL Standard specifications improve with Altera FPGA support, programming FPGAs using OpenCL can provide an alternative development solution for programming FPGA devices. Additionally, newer chips with larger FPGA fabric can be used to implement multiple instances of a back projector to offer multiple compute units providing a competitive solution to other heterogeneous solutions.


