OPENCL BASED DIGITAL IMAGE PROJECTION ACCELERATION

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

Bryan Michael Badalamenti

UNIVERSITY OF DAYTON

Dayton, Ohio

August, 2015
OPENCL BASED DIGITAL IMAGE PROJECTION ACCELERATION

Name: Badalamenti, Bryan Michael

APPROVED BY:

Eric Balster, Ph.D.
Advisor Committee Chairman
Associate Professor, Department of
Electrical and Computer Engineering

Frank Scarpino, Ph.D.
Committee Member
Professor Emeritus, Department of
Electrical and Computer Engineering

Vijayan Asari, Ph.D.
Committee Member
Professor, Department of Electrical and
Computer Engineering

John G. Weber, Ph.D.
Associate Dean
School of Engineering

Eddy M. Rojas, Ph.D., M.A., P.E.
Dean
School of Engineering
ABSTRACT

OPENCL BASED DIGITAL IMAGE PROJECTION ACCELERATION

Name: Badalamenti, Bryan Michael
University of Dayton

Advisor: Dr. Eric Balster

In this thesis, several implementations of an image back projection algorithm using Open Computing Language (OpenCL) for different types of processors are developed. Image back projection is a method to take aerial imagery and create a map-like image that contains real-world dimensions and to remove the perspective angle from the camera. The processors that ran the back projection algorithm include a Central Processing Unit (CPU), a Many Integrated Core (MIC), two Graphic Processing Units (GPUs), and two Field-Programmable Gate Array (FPGA) devices all have different architectures are require different programming styles. OpenCL is a new programming standard that provides a common programming language between the different types of devices using a widely used programming environment. OpenCL follows the C99 Standard and provides support for devices with parallel computing capabilities to help create an optimized solution. Different versions of the back projection algorithm are presented here to examine the flexibility of the OpenCL standard and to create the most optimal solution among the different devices. Timing measurements are compiled for each version of the algorithm for each device and compared against one another and a C++ single threaded host CPU implementation used as a baseline. The fastest implementation for each device is then compared in terms of execution time, throughput, and maximum
total dissipated energy (TDE). The presented OpenCL back projection solution provides a 90 times speedup (62.2 ms) on the CPU when compared to the baseline implementation (5680.7 ms) of the algorithm. The throughput for this CPU implementation is 1302.25 megabytes per second (MB/s) and the maximum TDE is 7.2 J per frame.
To my entire family, especially my parents, for all of their love, support, and encouragement.
ACKNOWLEDGMENTS

I would like to thank the following people for their support in making all of this possible and getting me to this point.

- **Dr. Eric Balster**: For giving me this opportunity in graduate school and for providing the means and the push needed for me to complete this in a professional manner.

- **Kerry Hill**: For funding and supporting my research.

- **Al Scarpelli**: For funding and supporting my research.

- **Bill Turri**: For supporting my research.

- **Andrew Kordik**: For contributing his knowledge and insight in image projection and providing me guidance along the way.

- **Jonathan Skeans**: For providing help and knowledge in OpenCL and providing me guidance along the way.

- **David Mundy**: It was rewarding to work alongside you in the lab.

- **Daniel Shaffer**: For providing help with image projection.

- **David Fan**: For his support in all things programming that allowed me to more quickly use the lab equipment and create testing scripts.
• **Dr. Frank Scarpino:** For his knowledge in back projection and being on my defense committee.

• **Dr. Vijayan Asari:** For bearing through my thesis defense.

• **AFRL/UDRI/ADDA:** For providing a great work environment and creating the framework and systems I used to finish my research and work.

• **Frank & Kathleen Badalamenti:** I would like to thank my parents for all the love and support they have provided me throughout my life. You are the best parents a son can ask for.

• **Adam Badalamenti:** I would like to thank my brother for always being there for me.

• **Helena Badalamenti:** I would like to thank my grandma for her support and reminders that I am getting old.

• **Buddy, Kiba, & Lotus:** My dogs who are always there when I get home providing me with emotional support and helping to clear my mind of all the stress.
# TABLE OF CONTENTS

ABSTRACT ................................................................................................................................... iii

DEDICATION ................................................................................................................................. v

ACKNOWLEDGMENTS ................................................................................................................ vi

LIST OF FIGURES ........................................................................................................................ x

LIST OF TABLES .......................................................................................................................... xii

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

  1.1 Objectives in Research ........................................................................................................... 2

  1.2 Significance of Research .......................................................................................................... 2

     1.2.1 Previous Optimization Attempts .................................................................................. 3

  1.3 Thesis Organization ............................................................................................................... 3

II. OPENCL OVERVIEW .................................................................................................................. 5

  2.1 History of OpenCL ................................................................................................................ 5

  2.2 Programming in OpenCL ..................................................................................................... 7

     2.2.1 Host Programming ...................................................................................................... 7

     2.2.2 Kernel Programming .................................................................................................. 12

III. BACK PROJECTION OVERVIEW ............................................................................................ 14

  3.1 Projection Background .......................................................................................................... 14

     3.1.1 Image Projection .......................................................................................................... 15

     3.1.2 Back Projection ........................................................................................................... 16

     3.1.2.1 Collinearity Equations .......................................................................................... 19

     3.1.2.2 Digital Elevation Maps (DEMs) ........................................................................... 20
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Flow chart of OpenCL programming and execution</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>The OpenCL framework</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Model of an OpenCL device regarding processing distribution</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Back projection processing chain</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Back projection diagram</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>The input aerial image provides little context of the image to the region</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Back projected aerial image provides additional information to a viewer</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Information about the region presented in the input image</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Sample 3x3 DEM grid</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Intel Xeon E5-2670 v2 CPU</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>Intel Xeon Phi 5110P</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>Nvidia K10 GPU</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Nvidia K40 GPU</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>BittWare board with an Altera GSD5 FPGA.</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>Nallatech board with an Altera GSD8 FPGA.</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Single Work Item Diagram</td>
<td>33</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5.2</td>
<td>Single Threaded Work Group Diagram</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Row Wise Threaded Diagram</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Column Wise Threaded Diagram</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Single Work Item Run Times</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>Single Work Item FPGA Optimization Run Times</td>
<td>43</td>
</tr>
<tr>
<td>6.3</td>
<td>Single Work Item FPGA Optimization Logic Utilization</td>
<td>44</td>
</tr>
<tr>
<td>6.4</td>
<td>Single Threaded Work Group Item Run Times</td>
<td>46</td>
</tr>
<tr>
<td>6.5</td>
<td>Single Threaded Work Group FPGA Optimization Run Times</td>
<td>47</td>
</tr>
<tr>
<td>6.6</td>
<td>Single Threaded Work Group FPGA Logic Utilization</td>
<td>48</td>
</tr>
<tr>
<td>6.7</td>
<td>Row Wise Threaded Run Times</td>
<td>49</td>
</tr>
<tr>
<td>6.8</td>
<td>Row Wise Threaded FPGA Optimization Run Times</td>
<td>50</td>
</tr>
<tr>
<td>6.9</td>
<td>Row Wise Threaded FPGA Optimization Logic Utilization</td>
<td>51</td>
</tr>
<tr>
<td>6.10</td>
<td>Column Wise Threaded Run Times</td>
<td>52</td>
</tr>
<tr>
<td>6.11</td>
<td>Column Wise Threaded FPGA Optimization Run Times</td>
<td>53</td>
</tr>
<tr>
<td>6.12</td>
<td>Column Wise Threaded FPGA Optimization Logic Utilization</td>
<td>55</td>
</tr>
<tr>
<td>6.13</td>
<td>Best run times for each device</td>
<td>56</td>
</tr>
<tr>
<td>6.14</td>
<td>A more detailed comparison of the fastest run times on each device</td>
<td>57</td>
</tr>
<tr>
<td>6.15</td>
<td>Comparison of the throughput for each device based on the fastest run time on the device</td>
<td>58</td>
</tr>
<tr>
<td>6.16</td>
<td>Comparison of the maximum TDE for each device based on the fastest run time on the device</td>
<td>59</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

4.1 Intel Xeon E5-2670 v2 CPU Specifications. ....................... 23
4.2 Intel Xeon Phi 5110P Specifications. ............................ 24
4.3 Nvidia K10 GPU Specifications. ................................. 25
4.4 Nvidia K40 GPU Specifications. ................................. 26
4.5 BittWare board with an Altera GSD5 FPGA Specifications. ......... 27
4.6 Nallatech board with an Altera GSD8 FPGA Specifications. ......... 28
6.1 Single Work Item O3 Optimizations. ............................. 44
6.2 Single Threaded Work Groups O3 Optimizations. ................. 47
6.3 Row Wise Threaded O3 Optimizations. .......................... 51
6.4 Column Wise Threaded O3 Optimizations. ........................ 54
In computing, parallel programming is a necessity when creating high performance software. One form of parallel programming is called data-parallelism is the execution of the function on different elements of a datasets at the same time. A second form is call task-parallelism. Task-parallelism is the execution of different functions on the same or different datasets at the same time. Parallelization can take place on a single or across several devices. Many parallel programming toolsets only include the first method [1]. This thesis topic focuses on the new programming standard called Open Computing Language (OpenCL) based on the C99 programming language which allows developers to implement both forms of parallel programming [2]. This standard was developed by Apple in conjunction with the Khronos Group and major processor manufacturers such as Altera, Intel, and Nvidia as a heterogeneous device programming language [1]. This language was inspired by Apple’s call for a common language for developers across the broad array of different types of processors. Prior to OpenCL, each manufacturer had its own method of programming their devices. With OpenCL, a single language could be used to program one or multiple devices and for those devices to more easily interface with one another. However, due to the nature of each device an implementation of an algorithm may need to be implemented differently on each type of
processor to get an optimized throughput. This thesis examines this idea by implementing different versions of an intensive processing algorithm across a range of processors to determine which implementation yields the best results.

1.1 Objectives in Research

The objective of this thesis is to optimize an image back projection algorithm using OpenCL on four types of processors. The algorithm is constrained to be a low Size, Weight, and Power (SWaP) imaging system with the highest possible throughput. This is accomplished by examining the capabilities of the OpenCL framework and potential in optimizing the process intensive algorithm. In an attempt to conform to the requirements four types of processors are examined. They include a Central Processing Unit (CPU), a Many Integrated Core (MIC), two Graphic Processing Units (GPUs), and two Field-Programmable Gate Array (FPGA) processors.

1.2 Significance of Research

The focus of this thesis is to optimize the image projection algorithm using OpenCL while examining the pros and cons on the new standard. Optimizing the image projection algorithm allows for quicker processing of aerial imagery for use in several arenas, from government agencies to private industry. Improving the processing speeds allows for near real-time processing and the low SWaP constraint means that the systems that this is implemented on will be able to be lighter and more efficient or will give these systems the opportunity to improve in other ways. The examination of OpenCL is done in the hopes of examining the feasibility and to gathering the pros and cons of using the new standard.
1.2.1 Previous Optimization Attempts

Back projection of an aerial image is not a new concept. As such, there have been many different attempts and methods used to improve the efficiency of the algorithm.

One such optimization was completed Kuo et al. where computation was reduced per pixel by using the results of neighboring pixels and introducing a special pipelined FPGA-based hardware architecture [3]. Using their system, they were able to accelerate orthorectification to 16 megapixels per second, a 16x improvement.

A second optimization also looked to reduce computation time per pixel. Chen et al. devised a method they called "Patch Backprojection" to minimize the computation required [4]. In this method, the area of interest was divided into multiple tiles. The corners of the tile are projected on the image to form a set of anchor points for both the lowest and highest elevations. An element from the tile is then projected into the image space based on the anchor points.

A third optimization was made by French & Balster. using integer arithmetic to accelerate the algorithm [5]. Two changes were made to the algorithm. First the floating point operations were removed and replaced with fixed-point arithmetic operations. Second, the division within the algorithm was replaced with a multiplication of the inverse. These changes reduced the processing time by a factor of 1.2x for 128-bit integer processing and more than a 2x reduction for 64-bit integer processing. Experimentation was completed only on a CPU, but this can be implemented on other types of processors for faster computations on those platforms.

1.3 Thesis Organization

The following two chapters are intended to give the reader an overall idea of tool and algorithm used in this document. The Chapter II begins with an introduction to concepts and practices in OpenCL development. Chapter III gives a brief introduction to the image projection algorithm.
The thesis then proceeds to Chapter IV, which discusses the process of testing that is done and reviews the processors utilized in testing. Chapter V discusses the different versions of the image projection algorithm devised and implemented in OpenCL for testing. Next, Chapter VI presents the performance results of the different algorithm versions on each device used. Finally, Chapter VII provides some conclusions based on the work presented in this thesis and ideas for possible future work.
CHAPTER II

OPENCL OVERVIEW

OpenCL is a relatively new programming language that allows developers to program an array of different processors with the same code. This provides developers quicker development when programming the same code for multiple types of processors. Each device manufacturer needs to conform to the OpenCL standard to be listed as a conforming device, but each manufacturer can release their own additions and development tools.

2.1 History of OpenCL

Prior to the creation of the OpenCL graphical and high performance computing had no dominant architecture [1, 6]. The different devices shared a common purpose, processor manufacturers like Nvidia and Altera had to create their own coding architectures for their devices, Cuda and the hardware description languages (HDL) respectively. Apple began to urge these vendors to create and share a common interface among their devices to ease the burden on developers. So Apple and several processor manufacturers came together and formed the OpenCL Working Group within the Khronos Group [1]. In 2008 this group released the first version of the OpenCL standard and as of March 2015 the OpenCL standard is at version 2.1 [7]. When creating the language, the working group expanded on the C99 language to create OpenCL [2]. All of the functionality provided by C/C++ is available to use in OpenCL, along with additional OpenCL specific functionality. By
utilizing functionality of OpenCL, processors such as GPU, FPGA, CPU, and MIC processors are able to be programmed using a single language. Code programmed in OpenCL which executes on a GPU can be reused without any changes to run on a FPGA provided the user has the proper tools and software development kits (SDKs).

The Khronos OpenCL working group developed the OpenCL language as an open standard, thus vendors are required to create and release their own OpenCL SDK for use with their devices. While they have to follow the standard, each vendor is able to release additional packages and tools geared towards using OpenCL on the devices they create. For example, Altera created and implemented functionality to their existing development tools, Quartus II, to allow developers to more easily use their devices. They created the Altera Offline Compiler (AOC) to compile the OpenCL code for use on their FPGAs [8]. The AOC includes the necessary code for board communication and additional optimization options [8]. Additionally, the AOC included tools that aid in the development and optimizations completed for the Altera FPGAs. Altera’s profiler id one of these tools. It displays several statistics about the OpenCL kernel gathered from the memory and channel accesses [9]. Tools such as this allows for in-depth analysis of the kernels and information on how and what data is accessed and by which operations. In regards to computing parallelism, the CPUs, GPUs, and MICs devices achieve it by dividing the data into multiple sets to be operated on by the same instructions (data-parallelism). FPGA devices utilize task-parallelism where different operations on the same set of data concurrently [1]. Using OpenCL features and extensions created by the vendors, both types of parallelism are able to be implemented on every type of device.

The main purpose of OpenCL is to provide developers a common interface language for all of the different computing device. The working group adopted a "write once, run on anything" philosophy when developing it, so code does not need to be re-written to operate correctly on different architectures. This, however, is not to say that the code will run efficiently on every device. As the
devices, for example FPGAs and GPUs, operate differently and need to be coded in different ways to optimize for a specific type of device architecture.

2.2 Programming in OpenCL

Programming in OpenCL can be split into two steps; host code programming and kernel programming. The host code is used to create the structure required to execute the OpenCL code, but can also include pre-processing computations such as computing constant values. Kernel programming involves creating the function to be completed. The basic process of creating and executing using OpenCL is shown in Figure 2.1.

![Figure 2.1: Flow chart of OpenCL programming and execution.](image)

The first step is to program the host code to set up the necessary environment. The algorithm to implement is then created in the kernel file. Details of these steps are described in the respective sections. The third step is to send every necessary part to the device and execute. The final step is to send the output data back to the host system and read it into a C/C++ compatible format.

2.2.1 Host Programming

The host code is a C/C++ application used to assign tasks and communicate to the devices to set parameters or retrieve the processing results. Most implementations use a CPU as the host to setup
and transfer the data between devices. The setup process for the host system is illustrated by Figure 2.2 [2].

![Figure 2.2: The OpenCL framework.](image)

Each of these elements of the system completes a specific task and is setup using functions introduced by the standard. They are, as defined in the OpenCL Specification Guide [10, 1],

- **Host**: A C/C++ application used to assign the tasks and communicate to the devices.

- **Devices**: An OpenCL compliant device that is made up of a compute units. I.e. a GPU or a FPGA.

- **Kernels**: The function to execute on a device and is identified by the __kernel qualifier.
- **Buffers**: The memory objects where the data to be processed and the processed data is stored.

- **Contexts**: The environment in which the kernels execute along with the defining of the synchronization, and memory management. Includes a set of devices, accessible memory and its properties, and one or more command queues.

- **Platforms**: A group of devices managed by the OpenCL framework that allows an application to share resources and execute kernels. Includes connected compliant devices and the host.

- **Command Queues**: Object that holds the commands to be executed by a specific device. They are queued in-order, but can be executed in either in-order or out-of-order.

- **Programs**: A set of kernels.

The context includes everything in the OpenCL framework. The context contains the program, memory objects, command queue, and the platform.

The host code is also charged with handling the data transfer and partitioning the data to be sent to the device(s). Prior to sending the data, three pieces of information are required. The program to be executed, the data to be processed, and an allocation of memory to store the output data [1]. In OpenCL, the program to be executed is store in a kernel file. On the host, the data to be sent for each kernel must be initialized to a kernel argument. Each kernel argument must be indexed to keep track of what data is what and the size of the data that is being sent to or from the kernel. Additionally, a pointer to the data space must be set [10]. Data of interest is stored in buffers in OpenCL. When creating buffer object, the size of memory to allocate, the pointer to the data on the host, and the memory access patterns must be set. For example, output buffers require a memory access pattern that would allow it to be written to and allowing the input buffer to be read is necessary. After creating the buffers, the host will need to create the command queue and enqueue it.
Most OpenCL devices have contain multiple compute elements. After creating the buffer objects, the data can be partitioned to run however the user wishes. This is done through a function that enqueues the data in the form of a *NDRange*. NDRange is the description of the *N*-dimensional organization of work-items to be executed by the kernel code [1]. The function will enqueue the kernel to the command queue, but will allow for the specification of the data and how the kernel execution is distributed among the processing units on a device. Two of the key arguments when specifying the NDRange are the *global_work_size* and the *local_work_size*. They are, respectively, the number of work items to execute and the number of work items in a work group. Work-items and work-groups are defined below along with memory spaces and the work-item and work-group identification parameters [1, 11].

- **Work-Item**: The single implementation of a kernel on a specific set of data.

- **Work-Group**: The combination of multiple work-items that have access to the same memory space.

- **Compute Unit**: The processing resource that can support a work-group.

- **Private Memory**: Memory that can only be accessed by a specific Global ID.

- **Local Memory**: Memory that can be accessed by all elements in a Work-Group.

- **Global/Constant Memory**: Memory that can be accessed by all compute units and where the input and output data is stored.

- **Host Memory**: Memory on the host system from where input and output data is sent from.

- **Global ID**: Identifies the work-item and allows access to the correct data (The private memory)
• **Local ID**: Identifies the work-items from one another within a work-group (The local memory)

A visual representation of these definitions is shown in Figure 2.3 [2].

![Figure 2.3: Model of an OpenCL device regarding processing distribution.](image)

Most OpenCL devices contain multiple compute units that can be used to process data. The work-groups will operate one at a time on a single compute unit. If no compute units are available then the work-groups are forced to wait until there is an available compute unit. This is the case when the number of compute units is less than the total number of work-groups. So if a device has $M$ compute units and $N$ work-items per work-group, then $MN$ work-items will be executed at a time [1]. Within each work-group are the work-items. Each work-item operates independently of the
other work-items in the group the memory that it has the rights to access. Each type of memory has its own advantages and disadvantages to use. The global/constant memory is very large and is where the buffers are typically initially stored. Reading from it is relatively slow to access compared to the other types of memory. The local memory can only be accessed by work-items with the correct local ID, but has faster read and write times than the global/constant memory since it contains only the relevant data to the work-group. The work-items in the group will write to this memory space while the work-group is still operating. The private memory will be the fastest memory for a work-item to access. It contains only what is needed for that work-item, but is also the most limited in size.

2.2.2 Kernel Programming

The second component of OpenCL programming is Kernel programming. It is here that the function to execute is programmed. Kernel programming is the heart of OpenCL programming containing the code for the desired operation. They resemble C function calls, but with three major differences. First, the kernel function must return void. Second, each kernel declaration begins with __kernel. And finally, arguments need to be set, as some platforms will not compile otherwise [1]. Data from a kernel is returned through the kernel arguments in the form of buffer objects. When passing pointers to a kernel one of the following address space qualifiers must precede it [1]:

- __global: The argument data will be in global memory.
- __const: The argument data will be stored in global, read-only memory space.
- __local: The argument data will be in local memory.
- __private: The argument data will be in private memory. The default qualifier.

Programming within a kernel also requires the use of these qualifiers. Most variables will likely be of the default type, private, but casting and declaring variables of the other types can be leveraged
in kernel programming with some caveats. The __global argument can only be used with pointer variables. The __constant argument is made available to the entire program and is stored in constant memory space. This also means that it cannot be declared within a kernel. The __local argument will create variables that have their memory spaces deallocated when the work-group is completed. These variables will be stored in the local memory and shared among the work-items in the work-group. They cannot be directly initialized [1].

Another addition to OpenCL is the use of vector data types. C/C++ programming are scalar based languages, while OpenCL is able to handle both types. Vectors in OpenCL share a similar to arrays in that they contain multiple elements of the same data type. However, they can only contain a specific amount of elements, sizes are 2, 3, 4, 8, and 16 [1]. The vector data types are used similarly to the scalar counterparts and when used take the form similar to floatn where n is the vector size, i.e. float4 [10]. Using vectors can allow for faster processing by eliminating the need to process elements individually for the same operation. Additionally, the elements in a vector can easily be accessed if a single element is needed for an operation.
CHAPTER III

BACK PROJECTION OVERVIEW

One of the goals of this thesis is to accelerate an image projection algorithm. This chapter of the thesis will focus on and provide a basic understanding of what projection is to provide context for the complexity and need for the algorithm.

3.1 Projection Background

The purpose of an image projection algorithm is to take an aerial image and display it in a manner that is more intuitively to the user, such as a map. By displaying the data after being projected, the user is able to get a better grasp over the situation being presented by the image. This is done by presenting the data like a map and correcting the orientation of the image so that the top represents north. Projection also gives real world measurements to the pixel data. In projection elevation data is used to correct the distortion that comes from taking the aerial image and to make it so that the pixels each represent a real distance.

Projection is useful in several areas of interest. The military and law enforcement agencies can use it for explosive ordinance identification and border patrol [12]. Scientists can utilize it when monitoring glacial erosion and deforestation [12]. Relief and disaster management agencies can use it to identify the scope and effect areas of disasters like landslides and earthquakes [12]. Private
industry companies such as Google are able to apply it to their mapping services for services like Google Earth [13].

### 3.1.1 Image Projection

For the purpose of this thesis, the process of image is completed using an orthorectification algorithm. Orthorectification is the process of projecting an aerial image onto an Earth based plane and tying the image pixels to Earth coordinates while also removing the perspective angle of the input data [5, 3]. There are two types of orthorectification, forward and back projection. Forward projection can be defined as the process of projecting each pixel of the image from the image plane onto the projection plane. Back projection can be defined as the process of projecting the world coordinates into the image space [5, 3, 4]. This method is the more computationally efficient of the two and will be used within the scope of this thesis research. The processing chain for back projection is shown below in Figure 3.1.

![Figure 3.1: Back projection processing chain.](image)
The raw image is taken and processed with the back projection algorithm. To properly project the image additional information is required. This includes the digital elevation map (DEM) and input data related to the camera taking the image. The input data includes information about the camera model, the interpolation value \( I \), global positioning system (GPS) data, and Inertial measurement unit (IMU) information based on the position of the camera when the image is captured.

### 3.1.2 Back Projection

Back projection can be broken into two major processing steps, the calculation of the image plane pixel locations and the interpolation of the terrain data. Figure 3.2 illustrates the concept of back projection.

![Back projection diagram](image.png)

**Figure 3.2:** Back projection diagram.

A single aperture camera assumption is used with the focal point located at the point \((X_c, Y_c, Z_c)\) [5]. This point represents the longitude, latitude, and altitude of the camera that imaged the
data with respect to the Earth. In the projection surface (the world coordinates) are represented by
$X$, $Y$, and $Z$ respectively. The point $P(X, Y, Z)$ represents the longitude, latitude, and altitude $X$, $Y,$
and $Z$ respectively in the projection surface. They are the real world coordinates of a point in the image captured by the camera. Values from the projection surface are projected onto the image plane relative to the camera at specific points on the surface. The projected points are located at the point $p(i, j)$. The values for a point in the image plane, in-between the points projected points, are interpolated to get the estimated value. The image in Figure 3.3 shows an input image to project.

![Figure 3.3: The input aerial image provides little context of the image to the region.](image)

The input image gives a good representation of the area of land captured in the image, but the landmarks present provide little context to anyone unfamiliar with those structures. To perform projection, information about the region needs to be collected. This information can contains details
such as the real world distances, directions, landmark features, and elevations. Performing back projection on the input image will give a viewer this context as a projected image, shown in Figure 3.4.

Figure 3.4: Back projected aerial image provides additional information to a viewer.

The output image has been rotated from the original input. This rotation is, as previously mentioned, done so that the top of the image represents north. In addition to the rotation, the image has been stretched to better represent the terrain present in the image. Each pixel of the output image represents a real distance, e.g. 1 m, allowing the viewer to more accurately understand the scope of the image.
3.1.2.1 Collinearity Equations

The image plane pixel locations are calculated using the set of collinearity equations shown below in Equation (3.1) [5].

\[
i = -\frac{m_{11}(X - X_c) + m_{12}(Y - Y_c) + m_{13}(Z - Z_c)}{m_{31}(X - X_c) + m_{32}(Y - Y_c) + m_{33}(Z - Z_c)} \quad (3.1)
\]

\[
\begin{align*}
  j &= -\frac{m_{21}(X - X_c) + m_{22}(Y - Y_c) + m_{23}(Z - Z_c)}{m_{31}(X - X_c) + m_{32}(Y - Y_c) + m_{33}(Z - Z_c)} \\
\end{align*}
\]

The variables \(X\), \(Y\), and \(Z\) respectively represent the longitude, latitude, and altitude of the projection plane. The pixel locations column and row values are defined as \(i\) and \(j\) respectively. The transformation matrix, \(M\), is defined as

\[
M = \begin{bmatrix}
m_{11} & m_{12} & m_{13} \\
m_{21} & m_{22} & m_{23} \\
m_{31} & m_{32} & m_{33}
\end{bmatrix} = R_\omega R_\phi R_\kappa \quad (3.2)
\]

where the coefficients, \(m_{mn}\), make up the elements of the matrix [5]. In Equation 3.2, \(R_\omega\) is the rotation matrix of the X-axis with respect to the angle \(\omega\), \(R_\phi\) is about the Y-axis with respect to \(\phi\), and \(R_\kappa\) is the matrix about the Z-axis with respect to the angle \(\kappa\). The rotational matrices are defined as

\[
R_\omega = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \omega & \sin \omega \\ 0 & -\sin \omega & \cos \omega \end{bmatrix} \quad (3.3)
\]

\[
R_\phi = \begin{bmatrix} \cos \phi & 0 & -\sin \phi \\ 0 & 1 & 0 \\ \sin \phi & 0 & \cos \phi \end{bmatrix} \quad (3.4)
\]

\[
R_\kappa = \begin{bmatrix} \cos \kappa & -\sin \kappa & 0 \\ \sin \kappa & \cos \kappa & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3.5)
\]

To calculate the transformation matrix coefficients, 36 additions and 54 multiplication operations are required. Each of the collinearity equations, ignoring the calculations required for the matrix coefficients, require 10 addition, 6 multiplications, and 1 division operation. Adding the number of operations brings the total requirement to determine each pixel location to 56 addition operations, 66 multiplications, and 2 divisions.
3.1.2.2 Digital Elevation Maps (DEM)

The second step in orthorectification requires interpolating the terrain data. The terrain data is represented as a DEM, shown in Figure 3.5.

Figure 3.5: Information about the region presented in the input image.

The DEM is a representation of elevation values over a topographic surface [14]. The ground sampling distance (GSD) of the DEM is what determines the number of pixels that will be in a given region [3]. Typically the GSD is too coarse for the pixel projection image size. To compensate for this an interpolation factor, $I$, is used to more densely represent the DEM. The value of $I$ is chosen to critically sample the image focal plane. Doing this allows the DEMs to be used to obtain a more accurate result when creating the flat image projection. Due to system restraints, pre-computed DEM is not possible for SWaP constrained systems. On these systems the DEM needs to be interpolated during projection. This is accomplished using a bilinear interpolation [5, 12].

Due to the size of the DEM, it is divided into a DEM grid [5]. Each part of the DEM grid is enclosed within DEM posts that are set at $\Delta X$ and $\Delta Y$ distance apart, i.e. 1 m. Figure 3.6 shows a sample DEM grid.
Figure 3.6: Sample 3x3 DEM grid.

In this DEM, the grid is made up of nine sections with $X_{Grid}$ and $Y_{Grid}$ representing the number of DEM divisions in each respective direction. Each of these sections represent the interpolated DEM and contain longitude, latitude, and altitude values for every point within the interpolated DEM. These points are denoted by $X_i$, $Y_i$, $Z_i$ in Figure 3.6. The DEM grid can be superimposed onto the projection surface, as in Figure 3.2, where each of the DEM posts is then projected into the image plane.
CHAPTER IV

TEST PROCEDURE

This chapter of the thesis reviews the testing procedure that was used in the scope of this thesis. This includes details on the computer hardware used in development and testing along with a description of the tests that were done to collect the data that will be presented in Chapter VI.

4.1 Test Hardware

One of the main goals of this research is to examine the performance capabilities of OpenCL. As such, several different types of processors are used for testing the image projection algorithm described in Chapter V. There are six devices that were used in testing. They include a CPU, a MIC, two GPUs, and two FPGAs.

The most common hardware used today are CPUs, GPUs, and FPGAs. MICs are a newer type of device and are not as commonly used. Each has advantages and disadvantages. The CPUs offer several highly optimized, but generalized, computing cores. Of the devices used, the CPUs have the highest clock speed. The GPUs have hundreds of non-optimized computing cores useful for parallel computing. MIC devices are composed of, as the name suggests, many cores. The amount of cores exceeds that of CPUs, but is less than a GPU. These cores are more optimized than the GPU cores, but do not feature the full optimization of a CPU core [15]. FPGAs allow for the creation of one or several custom computing core, limited only by the available logic elements on the device [8], that
take on a pipeline structure. The FPGA requires that the core(s) be pre-compiled as a binary file prior to execution. This binary file is loaded onto the FPGA by the host system at run time.

The performance of the image projection code is compared against the timing of a C++ single threaded implemented on an Intel Xeon L5640 CPU. This CPU has 6 cores and the bandwidth is limited to 32 GB/s [16]. The base processor frequency is 2.26 GHz and a maximum operating frequency of 2.8 GHz.

4.1.1 Intel Xeon E5-2670 v2 CPU

The Intel Xeon E5-2670 v2 CPU shown in Figure 4.1 was used in this study [17].

![Intel Xeon E5-2670 v2 CPU](image.png)

Figure 4.1: Intel Xeon E5-2670 v2 CPU.

The specifications regarding the number of core processors, the base and maximum frequency, the maximum bandwidth, and the maximum total dissipated power (TDP) of the CPU can be seen in Table 4.1.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Base Frequency (GHz)</th>
<th>Max Frequency (GHz)</th>
<th>Max Bandwidth (GB/s)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2.5</td>
<td>3.3</td>
<td>59.7</td>
<td>115</td>
</tr>
</tbody>
</table>
The Intel CPU also allows for 20 threads to be created per processing core. Thus, the total number of allowable threads for this CPU is 200 threads.

4.1.2 Intel Xeon Phi 5110P

The Intel Xeon Phi 5110P, shown in Figure 4.2, is the MIC that is used [18].

![Figure 4.2: Intel Xeon Phi 5110P.](image)

The specifications regarding the number of core processors, the base and maximum frequency, the maximum bandwidth, and the maximum TDP of the Phi can be seen in Table 4.2.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Base Frequency (GHz)</th>
<th>Max Frequency (GHz)</th>
<th>Max Bandwidth (GB/s)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1.053</td>
<td>1.053</td>
<td>320</td>
<td>225</td>
</tr>
</tbody>
</table>

The Intel Phi has more cores compared to the CPU, but each of those cores has a clock speed that is two to three times slower. The Intel Phi has a much better bandwidth capabilities, but also has a higher possible power usage.

4.1.3 Nvidia K10 GPU

The Nvidia K10 GPU, shown in Figure 4.3, is contains two GK104 GPU chips [19].
The specifications regarding the number of core processors, the base and maximum frequency, the maximum bandwidth, and the maximum TDP of the K10 GPU can be seen in Table 4.3.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Base Frequency (GHz)</th>
<th>Max Frequency (GHz)</th>
<th>Max Bandwidth (GB/s)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>3072</td>
<td>745</td>
<td>745</td>
<td>320</td>
<td>225</td>
</tr>
</tbody>
</table>

Each of the GK104 chips have 1536 core processors and a base processor frequency of 745 MHz. The maximum bandwidth of each chip is 160 GB/s giving the total max bandwidth seen in Table 4.3.

4.1.4 Nvidia K40 GPU

The Nvidia K40 GPU, shown in Figure 4.4, contains a GK110B GPU chip [20].
The specifications regarding the number of core processors, the base and maximum frequency, the maximum bandwidth, and the maximum TDP of the K40 GPU can be seen in Table 4.4 [21].

<table>
<thead>
<tr>
<th>Cores</th>
<th>Base Frequency (GHz)</th>
<th>Max Frequency (GHz)</th>
<th>Max Bandwidth (GB/s)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2880</td>
<td>745</td>
<td>875</td>
<td>288</td>
<td>235</td>
</tr>
</tbody>
</table>

When compared to the Nvidia K10 GPU, the K40 GPU has fewer total processors, but each of those processors can boost their processing speeds to 810 MHz and 875 MHz. It has a lower bandwidth capabilities and the same maximum power consumption.

### 4.1.5 Altera GSD5 FPGA

The BittWare board, S5PH-Q, contains an Altera Stratix V GSD5 FPGA and is shown in Figure 4.5 [22].
Figure 4.5: BittWare board with an Altera GSD5 FPGA.

The specifications regarding the number of equivalent logic elements, the number of 18x18 variable-precision multipliers, the maximum operating frequency, and the maximum TDP of the BittWare GSD5 board can be seen in Table 4.5.

Table 4.5: BittWare board with an Altera GSD5 FPGA Specifications.

<table>
<thead>
<tr>
<th>Equivalent Logic Elements</th>
<th>Multipliers</th>
<th>Max Frequency (MHz)</th>
<th>SDRAM (GB)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>475000</td>
<td>3180</td>
<td>500</td>
<td>8</td>
<td>75</td>
</tr>
</tbody>
</table>

This FPGA contains 457K equivalent logic elements and 3180 18x18 variable-precision multipliers. The maximum operating frequency is 500 MHz. The device has a maximum TDP of 75 W.

4.1.6 Altera GSD8 FPGA

The Nallatech FPGA, 395-D8, contains an Altera Stratix V GSD8 FPGA and is shown in Figure 4.6 [23].
The specifications regarding the number of equivalent logic elements, the number of 18x18 variable-precision multipliers, the maximum operating frequency, and the maximum TDP of the Nallatech GSD8 FPGA can be seen in Table 4.6.

Table 4.6: Nallatech board with an Altera GSD8 FPGA Specifications.

<table>
<thead>
<tr>
<th>Equivalent Logic Elements</th>
<th>Multipliers</th>
<th>Max Frequency (MHz)</th>
<th>Max SDRAM (GB)</th>
<th>Max TDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>475000</td>
<td>3926</td>
<td>500</td>
<td>32</td>
<td>75</td>
</tr>
</tbody>
</table>

The one contained on this board is the GSD8 which contains more equivalent logic elements and variable-precision multipliers than the GSD5 at 695K and 3926 respectively. The maximum operating frequency and TDP the same at 500 MHz and 75 W.

### 4.2 Test Description

The test setup involves utilizing all of the OpenCL compliant devices listed in Section 4.1 as well as the four versions of the image projection algorithm that are listed in Chapter V. The four versions of the back projection algorithm are executed on each device for a single frame, shown in
Figure 3.3, and the execution time of each run is then measured using a monotonic timer. The test is repeated ten times and the average time is taken as the final result to be presented in Chapter VI. Execution times are then compared against the C++ single threaded CPU implementation derived from the source code. Estimated energy usage and throughput is then calculated on each device for the kernel that ran the quickest on it. For the boards containing Altera FPGAs, three of the compiler’s built-in optimizations are used along with a test using none of these options. The options used are the duplication of the compute units, utilization of SIMD (single instruction, multiple data) vector paths, and a compiler resource driven optimization (O3). When using OpenCL for the Altera FPGAs, the code needs to be compiled into an executable file prior to running. Creating the compiles is a resource intensive endeavor. As such, prior to running the compiles, a partial compile is completed using the AOC. This partial compile gives an estimate of the board logic utilization. The estimations, based on observations when completing the compiles for this thesis, were fairly accurate and considered when determining to do a full compile. For the scope of these tests, any estimation that exceeds 85% is deemed too high, due to a possibility of not fitting on the board, and not completed. This value was selected from the maximum logic utilization that the O3 optimization set when doing the resource-drive optimization [9].

As stated in Section 2.2.2, compute units are the processing resources that can support a workgroup. On devices such as the GPU, each of the core processors make up a compute unit. On the Altera FPGAs, the AOC only creates a single core when completing a compile to setup the FPGA hardware [9]. Using the AOC’s compute unit option allows the compiler to create multiple cores based on the user specification and the amount of usable logic elements remaining on the board. The creation of the component to correctly output the data from each of these cores is handled by the OpenCL compiler [9].
The SIMD operation, as the name suggests, allows the user to program a single input and receive multiple outputs though kernel vectorization. In kernel vectorization, scalar operations are translated to SIMD operations [9]. The compiler vectorizes the code and coalesces memory access accordingly. Kernel vectorization can be completed manually, but it is a much more time and work intensive conversion that requires some additional code manipulation. To compile using the SIMD option, the number of work-groups for the kernel needs to be specified [9]. Additionally, the SIMD factor chosen ($n=2, 4, 8,$ and 16) must evenly divide the number of work-groups. This is a compiler limitation that resulted in the Single Work Item and Single Threaded Work Groups version of the code not being able to utilize this option.

The final optimization used is the O3 option. O3 gives the compiler the control over what optimizations are used based on the resources available. As previously mentioned, the compiler sets a maximum logic utilization to 85%. The AOC "analyze automatically the effects of combining various values of kernel attributes" [9]. The AOC uses the analysis to determine multiple attributes of the code. They include:

- SIMD vectorization
- Number of compute units
- A sharing factor
- Loop unroll

The first two items are the optimizations previously described. The sharing factor is found based on the analysis the AOC runs. It finds the code blocks where work-items are infrequently executed and attempts to reuse the same hardware for multiple operations [9]. The loop unroll takes a loop, such as a `for` loop, and decreases the number of iterations the AOC executes. On the FPGA there is more logic used to unroll a loop. This is a result of the compiler creating more fabric to process the
loop rather than just repeating through it. The compiler will run through several different scenarios with different combinations of these optimizations, favoring the loop unroll and SIMD options [9], and will pick what it believes is the best optimization.

The use of these three built-in optimization techniques on the FPGAs in addition to the other devices gave a total of 64 possible tests to perform. However, not all 64 tests were performed due to the logic utilization for some of the FPGA optimizations. The tests that could not be completed are documented in Section 6.1.
CHAPTER V

BACK PROJECTION ALGORITHM VERSIONS

Four different versions of the image back projection algorithm are created using OpenCL. Each are coded in a different style. The difference in coding has two reasons. The first is to allow different memory access patterns and operations. The second reason is to create code more suited towards particular hardware. The four versions have been styled as Single Work Item, Single Threaded Work Group, Row Wise Threaded, and Column Wise Threaded.

5.1 Single Work Item

The Single Work Item version of the algorithm has only one thread to process the entire DEM grid. There is only one work-group and one work-item in, hence the name, as shown in Equations 5.1 and 5.2 where \( \text{Work Group Dimension} \) and \( \text{Work Item Dimension} \) are the number of work-groups and work-items used respectively.

\[
\text{Work Group Dimension} = [1, 1]
\]  \hspace{1cm} (5.1)

\[
\text{Work Item Dimension} = [1, 1]
\]  \hspace{1cm} (5.2)

This is the coding method that Altera has recommended for their FPGAs due to the pipeline structure that it creates where at each stage of the clock there should be something being processed until
completion [9]. The processing flow of the Single Work Item version is illustrated in Figure 5.1 where the $N^{th}$ instance represents $N$ pixels completed.

![Diagram showing processing flow](image)

*Figure 5.1: Single Work Item Diagram*

For this version, the first pixel in the DEM grid is processed, shown in Figure 5.1a, followed by each sequential pixel in the DEM, going to the next row after completing one, until that DEM is complete. After it finishes the last pixel in the DEM it begins to work on the next DEM in the row of DEMs, shown in Figure 5.1c. It will iterate this pattern until it moves onto the next row of DEMs in the DEM grid until it finishes the last pixel in the final DEM of the grid, shown in Figure 5.1f. The kernel code for the Single Work Item version of the projection code is programmed as a single loop. The pseudo code for the algorithm is shown below:
The Single Work Item version is implemented in three loops. The outermost loop goes through every DEM in the DEM grid. The second loop iterates through each column in the DEM. The final loop iterates through each row in the DEM. This results in every point in the DEM grid being operated on. For each of these points there are four generalized steps. The first is to determine the current pixel information from the DEM. Then the collinearity equations, Equations 3.1 and 3.1.2.1, are completed to determine the output pixel location. Next, the current pixel is interpolated using the DEM data to determine the output value. Finally, the output value is written to the output buffer. Any point that is beyond the boundaries of the DEM grid are set to a value of 0.

### 5.2 Single Threaded Work Group

The Single Threaded Work Group implementation has each work-group as a single thread to process all of the pixels in the DEM grid. What makes this version different from the Single Work Item version is that all of the DEMs run in parallel rather than sequentially. Due to their being only a single thread, the number of work-groups is, as shown in Equation (5.4), set to 1. The number of work-items in the work-group is determined by the size of the image and the spacing between the DEM posts, as shown in Equation (5.3) where the \( N \)th instance represents the sum of the components.
of the Work Group Dimensions.

\[
Work\text{ }Group\ Dimension = [X_{\text{Grid}}, Y_{\text{Grid}}]
\]

\[
Work\text{ }Item\ Dimension = [1, 1]
\]

This concept of the Single Threaded Work Group is illustrated in Figure 5.2.

![Figure 5.2: Single Threaded Work Group Diagram](image)

In this version, the first pixel in each sub-DEM is determined as shown in the Figure 5.2a. From there, the code runs through the first row of pixels and determining the correct values for each of those locations before moving onto the next row of pixels, as shown in the Figure 5.2c. The code will run until the final pixel in the final row of the DEM is calculated as seen in Figure 5.2f. The kernel code for the Single Threaded Work Group version of the projection code is programmed to calculate each pixel individually. The kernel is executed for each pixel to calculate the output image.

The pseudo code for the algorithm is shown below.
The Single Threaded Work Groups version is completed by \( N \) work-items in a single thread. This version of the algorithm has two loop steps. The first is to loop through each column in the DEM and the second is to loop through each row in the DEM. While the Single Work Item version contains a third loop, going through each DEM in a DEM grid, this version of the algorithm replaces that loop by utilizing multiple work-groups. By using multiple work-groups one of the loops is removed allowing for multiple DEMs to be operated on in parallel. The interior of the code is identical to the Single Work Item version where there are four main steps. First the current pixel location is determined, then the collinearity equations are used to determine the output pixel location. The third step is to calculate the output value and finally the output is written to the output buffer. Any point out of the boundaries of the DEM are set to a value of 0.

### 5.3 Row Wise Threaded

The Row Wise Threaded version of the back projection algorithm is a \( N \) threaded process. The number of threads, \( N \), depends on the iteration factor, \( I \), specified on the host code. The number of work-groups depends on the size of the image used, the spacing between the DEM posts, and the value of \( I \). The number of work groups and the number of work-items within each work group is determined on the host by Equations (5.5) and (5.6).

\[
Work\ Group\ Dimension = [X_{Grid} \cdot I, Y_{Grid}] \quad (5.5)
\]
In this version each thread in a work-group processes a row of pixels within each sub-DEM. A diagram of the process is depicted in Figure 5.3.

![Diagram of Row Wise Threaded Process]

Figure 5.3: Row Wise Threaded Diagram

In this version the DEMs in the DEM grid is completed in parallel. Within each DEM, an entire column of pixels is completed, shown in Figure 5.3a. After the values of the column of pixels are determined, the next column of pixels in the row are completed until the entire DEM is complete, shown in Figure 5.3c. This version is referred to as Row Wise Threaded, because each row of pixels in a DEM is a thread. The kernel code for the Row Wise Threaded version of the projection
code is programmed so that an \( I \) pixels are calculated for each kernel call. The pseudo code for the algorithm is shown below.

```
// Do projection over each row in a DEM
FOR each Row in DEM
    Obtain current pixel information from DEM based on Row
    Do collinearity equations
    Do DEM interpolation
    Write the output to the output buffer
END FOR // Row
```

The Row Wise Threaded version is implemented using only one loop. As with the Single Threaded Work Groups version, the loop iterating through the DEM grid has been replaced by multiple work-groups. The loop iterating through each column in the DEM has also been removed. Instead, the Row Wise Threaded version implements multiple threads. The number of threads is determined by the interpolation value \( I \) as specified in Equation 5.6. The only remaining loop iterates over each row in a DEM where the number of rows is based on the data set being used. As with both versions already mentioned the interior of the code remains the same; calculate the current pixel location, determine the output location with the collinearity equations, calculate the output value, and write to the output buffer. Any point outside the DEM boundaries are set to a value of 0.

### 5.4 Column Wise Threaded

The final version of the code is nearly identical to the Row Wise Threaded version. The Column Wise Threaded version is also \( N \) threaded, with the number of threads depending on \( I \). The number of work-groups also depends on the size of the image used, the space between DEM posts, and the value of \( I \). The number of work groups and the number of work-items within each work group is determined on the host by Equations (5.7) and (5.8).

\[
\text{Work Group Dimension} = [Y_{\text{Grid}} \cdot I, X_{\text{Grid}}] \tag{5.7}
\]

\[
\text{Work Item Dimension} = [I, 1] \tag{5.8}
\]
The number of work-items for this version can vary from the Row Wise Threaded based on the dimensions of the DEM grid. Column Wise Threaded is different due to the way each thread in a work-group processes the data. In this version, each thread processes a column of pixels within each DEM, as shown in Figure 5.4.

In this version the DEMs in the DEM grid is completed in parallel. Within each DEM, an entire row of pixels is completed, shown in Figure 5.4a. After the values of the column of pixels are determined, the next column of pixels in the row are completed until the entire DEM is complete, shown in Figure 5.4c. This version is referred to as Column Wise Threaded, because each column of pixels in a DEM is a thread. The kernel code for the Column Wise Threaded version of the projection code is nearly identical to the Row Wise Threaded. The only major difference is which
dimension the projection is completed across which is based on the flipped work-item dimension.

The pseudo code for the algorithm is shown below.

```plaintext
// Do projection over each row in a DEM
FOR each Column in DEM
    Obtain current pixel information from DEM based on Column
    Do collinearity equations
    Do DEM interpolation
    Write the output to the output buffer
END FOR // Column
```

In the pseudo code, the difference from the Row Wise Threaded version is what the loop is iterating over. In the Row Wise Threaded version it was over the rows. For the Column Wise Threaded version it is the columns in the DEM where the number of columns in a DEM is dependent on the data set being used. Otherwise the code is identical to the Row Wise Threaded version. The loop iterating through each DEM in the DEM grid is replaced with multiple work-groups and the loop iterating through each row in a DEM is replaced by using multiple threads. The number of threads is determined by the interpolation value $I$ as specified in Equation 5.8. As with both versions already mentioned the interior of the code remains the same; calculate the current pixel location, determine the output location with the collinearity equations, calculate the output value, and write to the output buffer. Any point outside the DEM boundaries are set to a value of 0.
CHAPTER VI

RESULTS

The timing results gathered were the result of running each of the code versions of the back projection algorithm described in Chapter V on all of the devices and compiler optimizations discussed in Section 4.1. The run times of each test is determined, measuring only the kernel execution time and ignoring the data transfer time, with a monotonic timer and recorded. For the FPGA tests that are deemed as not possible, due to hardware or software limitations, no run times are recorded.

6.1 Timing Measurements

In this section of the thesis, the timing results gathered from tests are analyzed by version. The best run time for each of the FPGA devices is selected and be compared to the timing from each of the other devices. As we will see, some of the AOC options could not be used due to a lack of logic elements on the board, a limit was set to less than 85% of the board’s logic, and others could not be completed due to compiler restrictions. It should also be noted that for each code version, only two compute units are implemented in the test. Each run time is the result of averaging the run times after repeating the test ten times. Each version comparison also features a run time comparison to a single threaded software time that was completed on the CPU referenced in 4.1 [5]. This timing acts as a baseline comparison between the different code versions, the different devices, and the
code base that the code in this thesis was derived from. The baseline run time was determined to be 5680.7 ms for this image frame [5].

6.1.1 Single Work Item Results

The first version of the back projection algorithm to examine is the Single Work Item. The best run times for each of the six available devices are shown in Figure 6.1.

![Single Work Item Run Times](image)

**Figure 6.1: Single Work Item Run Times.**

The results show that the FPGAs performed the best when running this version of code, conforming to Altera’s recommendation. The Altera GSD5 FPGA ran the fastest at 570 ms, a near 10x improvement from the baseline time, followed by the Altera GSD8 FPGA at 592 ms. The GSD5 FPGA design used the O3 optimization, while the GSD8 FPGA used no AOC optimizations. The CPU, Phi, and GPUs all performed worse than the baseline test, with the Nvidia K40 performing the worst at 2.67 minutes. The likely reason for the non-FPGA devices performing so poorly is based on the structure of the code. Being a single threaded and single work-item code, the advantages
of those devices is not taken advantage of. Instead each pixel was processed individually, one at a time. Each pixel had to be completed before the next pixel could begin. On the FPGAs, pixels were able to start one after the other until there are no more pixels to process.

Figure 6.2 shows the results of utilizing the Altera compiler optimizations.

![Single Work Item: FPGA Optimization Run Time (ms)](image)

Figure 6.2: Single Work Item FPGA Optimization Run Times.

The first observation to note is that the SIMD optimization are not done. This is due to a limitation set by the compiler that requires multiple work-groups such that the specified SIMD can evenly divided the work-group size [24]. The work-group size for the Task implementation is 1, keeping the SIMD option from being used. Due to this only the compute unit and O3 optimizations could be completed. The second observation is that it was the versions without AOC options were the fastest for both FPGAs at 570 ms and 592 ms for the GSD5 and GSD8 FPGAs respectively. Duplicating the number of compute units did not yield the expected results. Instead of cutting the run time by approximately half for each additional compute unit, as it was expected [9], the run time on both board had a slight increase in run time. The O3 optimization on the other hand, performed
extremely poorly with both taking over a minute to complete. The four main components to the O3 optimization are SIMD vectorization, number of compute units, a sharing factor, and the loop unrolling. Table 6.1 shows the optimizations that the compiler chose for the Single Work Item version of back projection.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>SIMD Vectorization</th>
<th>Compute Units</th>
<th>Sharing Factor</th>
<th>Unroll Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSD5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>GSD8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

In the table, a value of 1 indicates no optimization applied, so only the loop unrolling optimization was completed for the Single Work Item version. Unrolling the loop had negative results as seen in Figure 6.2. This is likely due to the structure of the algorithm, a single large loop, not efficiently managing the elements. The logic utilization for both boards is shown in Figure 6.3.

**Single Work Item: FPGA Logic Utilization (%) of Chip**

![Figure 6.3: Single Work Item FPGA Optimization Logic Utilization.](image-url)
Since no SIMD option could be used there is no logic utilization data. As expected the no AOC optimization version of the code used the least amount of logic elements. The GSD5 build without any AOC optimizations, having less equivalent logic elements, utilizes 39% of the board while the GSD8 build without any AOC optimizations uses only 33%. The percentage of the board each compute unit uses is 24% for the GSD5 and 12% for the GSD8 FPGA. The rest of the board usage can be attributed to the control logic that is provided for each board by the vendor. Doubling the number of compute units results in increased logic utilization. The O3 option on the GSD5 FPGA uses less that the compute units option, but performed considerably worse. The GSD8 FPGA uses nearly twice the amount of logic with the O3 optimization compared to the no AOC optimization version and also performed considerably worse. From this we can conclude that the amount of logic used for an algorithm does not necessarily correspond to performance.

6.1.2 Single Threaded Work Group Results

The second version of the back projection algorithm to examine is the Single Threaded Work Group. The best run times for each of the six available devices are shown in Figure 6.4.
All of the run times for the Single Threaded Work Group implementation ran significantly faster than the baseline time. The fastest device is the Intel CPU at 339.9 ms, a near 17x improvement from the baseline. The slowest device is the Nvidia K10 GPU at 1746.4 ms, a 3.25x improvement over the baseline. This version of the code is able to utilize advantages for every device. It has a partially pipelined structure that allowed it to perform well on the FPGAs, but it was not restricted to a single work-item. This allowed it to utilize more of the processing elements of the CPU, GPUs, and the Phi.

Figure 6.5 shows the results of utilizing the Altera compiler optimizations.
For the Single Threaded Work Groups code, the SIMD optimization could not be used due to the same reason as the Task, single work-groups cannot use AOC’s SIMD option. The usable AOC options results in run times not much different than the baseline version, which had yielded the best times on both FPGAs. Again, the compute unit duplication did not yield the expected results. Both boards faced a slight increase in run time when duplicating the compute units. The O3 optimization provided much better results for Single Threaded Work Groups than it did for Task. The run times for both the GSD5 and GSD8 FPGAs were slower by 100 ms and 27 ms respectively. Table 6.2 shows the optimizations that the compiler chose for the Single Threaded Work Group version of back projection.

Table 6.2: Single Threaded Work Groups O3 Optimizations.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>SIMD Vectorization</th>
<th>Compute Units</th>
<th>Sharing Factor</th>
<th>Unroll Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSD5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>GSD8</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
The compute unit was duplicated twice to give a total of 3 compute units. The resulting O3 optimization behaved similarly to the duplication of compute units optimization and increased the execution time. The logic utilization for the GSD5 and GSD8 FPGAs can be seen in Figure 6.6.

![Single Threaded Work Groups: FPGA Logic Utilization (% of Chip)](image)

Figure 6.6: Single Threaded Work Group FPGA Logic Utilization.

Again, the versions with no AOC optimizations had the lowest utilization at 34% and 29%. Each compute unit uses 8% of the GSD5 FPGA and 9% of the GSD8 FPGA. The O3 options used a significantly greater amount of the FPGA logic elements reinforcing the notion that more logic does not yield faster results.

### 6.1.3 Row Wise Threaded Results

The third version of the back projection algorithm to examine is the Row Wise Threaded. The best run times for each of the six available devices are shown in Figure 6.7.
The run times of the Row Wise Threaded version were all much faster than the baseline run time. The fastest is on the Intel CPU at 82.2 ms, about 69x faster, and the slowest run time was on the GSD8 FPGA at 869.6 ms, about 6.5x faster than the baseline. This version of the code allows the CPU, GPUs, and the Phi to use multiple cores and/or threads resulting in much better run times than in the previous two implementations. The GSD5 FPGA’s Row Wise Threaded run time is better than the previous two implementations, while the GSD8 FPGA ran more than 300 ms slower than both of the previous code versions.

The run times for each of the AOC optimizations are shown in Figure 6.8.
Figure 6.8: Row Wise Threaded FPGA Optimization Run Times.

The Row Wise Threaded was able to implement the compute unit, O3, and SIMD optimizations. The fastest optimizations for the GSD5 and GSD8 FPGAs are correspondingly the SIMD with a vectorization of two, 491 ms, and the O3 optimization, 869.6 ms. The version with no AOC optimizations for the GSD8 FPGA ran three times slower than the GSD5 FPGA. The GSD5 FPGA was not able to meet the logic requirements set by the guidelines of the testing procedure in Section 4.2 for both the compute units and SIMD of 8 optimizations. The GSD8 FPGA was able to duplicate the compute units and experienced a slight improvement in execution speed. Of the code versions presented thus far, this is the first to allow the use of the SIMD optimization. When using a vectorization of a factor of two, there was only a slight improvement. Increasing the vectorization factor to four had a much greater effect on the run time of the Row Wise Threaded code. On the GSD5 there was 1.8x improvement and on the GSD8 there was a 1.87x improvement over the vectorization factor of two. The GSD8 FPGA was also able to fit a vectorization factor of eight. This resulted in a 1.34x improvement over the vectorization factor of four. The O3 optimization resulted
in a better performance for both boards. The GSD5 FPGA had an improvement of about 50% from the version without any AOC optimizations and the GSD8 had a 74% decrease in execution time. Table 6.3 shows the optimizations that the compiler chose for the Row Wise Threaded version of back projection.

Table 6.3: Row Wise Threaded O3 Optimizations.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>SIMD Vectorization</th>
<th>Compute Units</th>
<th>Sharing Factor</th>
<th>Unroll Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSD5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GSD8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

For the GSD8 FPGA, unrolling was done resulting in the 74% decrease in execution time. The GSD5 FPGA utilized both the SIMD and compute unit duplication options. The results were an optimization that had a lower run time than the SIMD of 2 optimization. The logic utilization for the AOC optimizations is shown in Figure 6.9.

Figure 6.9: Row Wise Threaded FPGA Optimization Logic Utilization.
The GSD5 and GSD8 FPGAs require 56% and 49% of the logic elements when compiled without any additional AOC optimizations for the Row Wise Threaded version of back projection. Each compute unit requires 33% for the GSD5 FPGA and 23% of the GSD8 FPGA. The GSD5 FPGA had an estimated logic usage of 89% and 98% for the compute units and SIMD of eight optimizations. These were not completed due to being higher than the 85% logic utilization cutoff point set by the test parameters. The optimizations described in Table 6.3 requires 68% and 71% of the logic elements for the GSD5 and GSD8 FPGAs. On the GSD5 FPGA each vectorization factor requires 3%-5.25% of the logic. The GSD8 FPGA needs 0.875%-2.375% of the logic for each vectorization factor.

6.1.4 Column Wise Threaded Results

The fourth version of the back projection algorithm to examine is the Column Wise Threaded. The best run times for each of the six available devices are shown in Figure 6.10.

![Column Wise Threaded: Best Kernel Run Time (ms)](image)

Figure 6.10: Column Wise Threaded Run Times.
The device with the fastest run time was the Intel CPU at 62.2 ms, approximately 91x faster than the baseline time. The slowest device was the GSD8 FPGA at 1215.7 ms, a 4.7x improvement from the baseline. As with the Row Wise Threaded, this version allowed the CPU, GPUs, and the Phi to utilize multiple cores and threads. This code version gave the best run time out of the four versions at 372.8 ms.

The run times for each of the AOC optimizations are shown in Figure 6.11.

![Figure 6.11: Column Wise Threaded FPGA Optimization Run Time (ms)](image)

For every test completed the GSD8 ran significantly slower than its GSD5 counterpart. The fastest optimization for both boards is utilizing the O3 optimization. Table 6.4 shows the optimizations that the compiler chose for the Column Wise Threaded version of back projection.
Table 6.4: Column Wise Threaded O3 Optimizations.

<table>
<thead>
<tr>
<th>FPGA</th>
<th>SIMD Vectorization</th>
<th>Compute Units</th>
<th>Sharing Factor</th>
<th>Unroll Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSD5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>GSD8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Both FPGAs used only the loop unrolling for optimization. Duplication of the compute units provided a minor run time improvement from the version with no AOC optimizations, while the SIMD did not behave as it did for the Row Wise Threaded. For the Column Wise Threaded implementation, increasing the amount of vectorization increased the time needed to complete the projection. A likely explanation is the different memory access pattern that the Column Wise Threaded takes. The data used is stored as an array of values. For the Row Wise Threaded, thread is an entire row of pixels in a sub-DEM as we can see in Figure 5.3. This means that the data for those pixels are all next to each other. For Column Wise Threaded we can refer back to Figure 5.4 to see that the pixels to process in each thread are not are not in connected memory addresses. Due to this the Column Wise Threaded implementation requires more reading and writing processes when processing multiple pieces of data concurrently. The logic utilization for the AOC optimizations is shown in Figure 6.12.
Both the GSD5 and GSD8 FPGAs could not fit the SIMD vectorization of eight, with an estimated logic estimation of 134% and 94% respectively. The compute units of each board is 15% and 12%. The optimizations described in Table 6.4 requires 42% and 51% of the logic elements for the GSD5 and GSD8 FPGAs. The SIMD logic utilization ranges from 4.5%-12.125% for the GSD5 FPGA and 47.625% for the GSD8 FPGA for every increase in the vectorization factor.

6.2 Comparison

For each of the six devices, the fastest run times are shown below in Figure 6.13.
Figure 6.13: Best run times for each device.

From this graphs we can see that each device has a run time of less than 600 ms. The improvements from the original baseline run time ranges from 10x to 90x faster. A more detail version is shown in Figure 6.14.
The fastest device was the Intel CPU at 62.2 ms running the Column Wise Threaded version of the projection algorithm. The next fastest devices were the Nvidia K40 and K10 at 153.2 ms and 251.8 ms respectively using the Row Wise Threaded version. The GSD5 FPGA was the next fastest device at 372.8 ms. The GSD5 FPGA was fastest using the Column Wise Threaded version with the O3 optimization. The Intel Phi was the next fastest device and also used the Row Wise Threaded. The slowest device was the GSD8 FPGA at 556 ms. This FPGA ran the Single Threaded Work Groups version without any AOC optimizations.

### 6.2.1 Throughput

The timing numbers are relative to the data set used in the algorithm. A more thorough comparison can be made by examining the raw throughput from the device. The throughput can be calculated, in megabytes per second, using from the following equation:

\[
\text{Throughput} = \frac{\text{Data Size}}{\text{Run Time}} \quad \text{[MB/s]} \tag{6.1}
\]
where the data size is the input image file size. The calculated throughput for each device is shown in Figure 6.15.

![Kernel Throughput (MB/s)](image.png)

Figure 6.15: Comparison of the throughput for each device based on the fastest run time on the device.

The values in this graph were found by applying Equation (6.1) on the execution times from Figure 6.14 and size of the input image. The Intel CPU has a throughput of 1302.25 MB/s and produces nearly nine times more bytes per second than the GSD8 FPGA, 145.68 MB/s. For a comparison, the baseline throughput is calculated to 14.26 MB/s.

### 6.2.2 Power Measurements

One of the design requirements of the image projection system was that it needed to be low SWaP. The power required is based upon the algorithm, data, and devices used. The six devices used in testing all had manufacturer max total dissipated power ratings as previously described.
Using Equation (6.2), the maximum total dissipated energies (TDE) can be determined.

\[ TDE = (TDP) \cdot (\text{Run Time}) \]  \hspace{1cm} [\text{J}] \hspace{1cm} (6.2)

The maximum TDE for each device based on the run times shown in Figure 6.14 are shown below.

Figure 6.16: Comparison of the maximum TDE for each device based on the fastest run time on the device.

These values represents the most energy that is to expect to be used by each device for a single implementation of the algorithm. The fastest running device, the CPU, also has the lowest maximum TDE at 7.153 J. The Intel Phi has the highest max TDE at 90.88 J.
CHAPTER VII

FINAL THOUGHTS

OpenCL is a new tool to more easily program across multiple types of devices. Based upon the research presented here, OpenCL has shown that it is capable of working well across different devices and optimizing algorithms if properly coded. The tools currently available are both useful and have a low learning curve.

7.1 Conclusion

The main goals addressed within this thesis are to analyze the usage of OpenCL while optimizing an image projection algorithm. The results presented in this document show that OpenCL can be used to provide optimization to the algorithm. Using OpenCL a 90x improvement was made from the C++ single-threaded CPU implementation being used as a baseline measurement. The fastest of the six devices utilized was the Intel Xeon E5-2670 CPU at 62.2 ms. The throughput is 1302.25 MB/s and the maximum TDE is 7.2 J per frame. The device ran on the Column Wise Threaded version of the image back projection algorithm. The slowest of the six devices, the Nallatech board with an Altera GSD8 FPGA, ran at 556 ms using the Single Threaded Work Groups version with a throughput of 145.68 MB/s. The slowest device was still ran over 10x faster than the baseline data point.
The tools available to use are fairly effective and have a low learning curve. One of the main tools used was the Altera Offline Compiler. This tool provided built-in optimizations and diagnostic tools to help improve performance and debugging. The AOC optimizations increased the usage of the equivalent logic elements, but, for the most part, did not operate as expected. Duplicating the number of compute units on the FPGAs did not yield the approximately 2x improvement that was expected. Instead it provided negligible improvements at best and longer run times in most cases for both of the FPGAs. Increasing the number of SIMD vector paths was only usable for two of the algorithm versions, Row and Column Wise Threaded. For the Row Wise Threaded version there was a decrease in run time for each FPGA for each additional vector path added. The BittWare board with an Altera GSD5 FPGA decreased from 1076.7 ms to 491 ms using four SIMD vector paths. The GSD8 FPGA decreased from 3322.3 ms down to 1098.6 ms using eight SIMD vector paths. The Column Wise Threaded version saw increases in the run times when the SIMD optimization was used. The Row Wise, for example, had an increase in run time from 3435.4 ms to 6943.3 ms when four SIMD vector paths were used. The reason the Row Wise Threaded worked while the Column Wise Threaded did not is likely due to the memory access patterns of each version. For the Row Wise, the memory addresses for each element in the vector paths were adjacent to one another. The Column Wise version had elements that were separated from one another, by a row of pixels within each sub-DEM. The final AOC optimization, the O3, provided a mixed results. The Single Work Item version had a very significant increase in run time, to over a minute for each board, while the Single Threaded Work Groups version had a slight negative impact on the performance. The Row Wise and Column Wise Threaded versions both had improved performances using the O3 optimization with the Column Wise Threaded O3 optimization providing the fastest run time for the GSD5 FPGA.

Over the course of the research done for this thesis, Altera has made improvements and fixes to their tools. The O3 optimization has been depreciated, due to the inconsistency in its performance.
Vendors have also acknowledged and created fixes to some bugs on their devices. Nallatech has been working to provide fixes to issues when using their board. When creating the FPGA compiles, the Nallatech board requires the user to turn off memory interlacing to get correct results. When compared to the BittWare board, which could use the interlacing, the run times on the Nallatech board were much slower. That board, using the same model FPGA chip with more logic elements, should provide similar, if not better, results for each test and more flexibility when optimizing with the AOC.

7.2 Future Research

The improvements made to the image back projection algorithm using OpenCL are significant, but future work would likely provide additional improvements. Future research on the subject could include exploring additional memory access patterns and tricks, examining additional devices, implementing in AOC function called channels, and introducing integer arithmetic.

7.2.1 Examine Additional Memory Access Patterns

This can be accomplished two ways. The first is to examine vectorization. Testing done for this system utilized only the automatic vectorization, the SIMD AOC optimization. Algorithms can be manually programmed in OpenCL to utilize data vectorization by using variables such as \texttt{float4}. Only the Altera FPGAs used data path vectorization when using the SIMD option of the AOC. Additionally, on the Row Wise and Column Wise Threaded versions of the code could utilize it due to limitations imposed by the tool. Doing a manual vectorization would allow for each of the four versions of the back projection algorithm to be completed on every type of device. There are hurdles to taking this approach such as the process of converting the algorithm to use vector paths. Changing the code so that the variables are, for example, \texttt{float4} does not necessarily mean that the code will then run correctly. Additional changes would need to be completed elsewhere in the
code. The difficulty of converting each version of the code is likely to also be an issue. Each version has different memory access patterns and using multiple data points could have a negative impact on the performance.

The second is to examine different access patterns. OpenCL lends itself to the division of labor being the processing elements. Each element has access to its own individual memory and two layers of shared memory when sent to a device. Further exploration into exploiting the usage of these different layers of memory may yield additional benefits. The memory layers act as caches that are more quickly accessed when it is smaller and more dedicated to the task of the processing element. This implementation of the algorithm relied heavily on the global and private memory spaces. The private memory used stored the information on most of the input variables and all of the intermediary calculations done. The global memory space stored the buffers for the input image, terrain data, and the output image. One of the AOC tool allowed for a detailed breakdown of the completion time of different portions of the algorithm and for each version of the algorithm the biggest bottleneck occurred when reading and writing to the output buffer. The integration of the local memory could lead to better read and write speeds, and lower total run time, for the projection algorithm. It is worth noting that all four versions of the algorithm may not benefit from the use of local memory. For example, the Single Work Item version due to the how it operates will likely not receive any benefits from writing to an intermediary memory space while the Row Wise Threaded would benefit due to the division of the labor between work groups and items.

7.2.2 Examine Additional Devices

For future research, an examination of both the next generation models of all of the devices used in this thesis, additional board vendors, and additional processor manufacturers should be examined. Firstly, all of the boards used for this thesis have been on the market for a few years and are limited to an older version of the OpenCL standard, version 1.1. Newer devices will also have
better specifications and possibly additional processing resources. Additions have been made to the standard since, additions that may provide further ease of use and possibly improvements in code efficiency. Second, different board vendors for the Altera FPGAs should be examined. Based upon the research presented it is clear that the board vendor does have a significant role in how well a device runs. For example, based on the board specifications, we would expect that that Nallatech board with a GSD8 FPGA should at least perform on par with the BittWare board with a GSD5 FPGA, which has less logic elements, but to have more room for improvements. The same logic would dictate that additional processor manufacturers be examined. Altera was the first to create and release an OpenCL compatible FPGA, but recently Xilinx has created and begun releasing their implementation. Altera’s tools, while great and continuously improving, are not perfect, e.g. the compute unit optimization. Xilinx’s devices and tools could provide features that Altera’s do not.

7.2.3 Channels

AOC channels are a new OpenCL extension included in version 14.1 of Altera’s toolkit, the latest version as the writing of this document. They allow the user to utilize multiple different kernels on a single FPGA device and communicate with each other [25]. This allows the user to optimize one block of code differently than a second piece of code. This allows for more possibilities to optimize the code and could allow for additional improvements from what have been achieved in this thesis.

7.2.4 Integer Arithmetic

In Section 1.2.1, French & Balster utilized integer arithmetic to achieve a 2x improvement for the back projection algorithm. Final remarks left this implementation open to being introduced on
all types of platforms which would be much easier to accomplish using OpenCL rather than developing for each individual platform. Converting the OpenCL implementations to integer arithmetic could further decrease the processing time of the algorithm.
BIBLIOGRAPHY


[16] Intel Xeon Processor L5640.

[17] Intel Xeon Processor E5-2670 v2.

[18] Intel Xeon Phi Coprocessor 5110P.


[22] S5-PCle-HQ.

[23] Nallatech OpenCL FPGA Accelerator Cards.
