DESIGN AND IMPLEMENTATION OF THE HETEROGENEOUS COMPUTING DEVICE MANAGEMENT ARCHITECTURE

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

Brian Robert Schultek

UNIVERSITY OF DAYTON

Dayton, Ohio

December, 2014
DESIGN AND IMPLEMENTATION OF THE HETEROGENEOUS COMPUTING DEVICE MANAGEMENT ARCHITECTURE

Name: Schultek, Brian Robert

APPROVED BY:

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

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

John G. Weber, 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

DESIGN AND IMPLEMENTATION OF THE HETEROGENEOUS COMPUTING DEVICE MANAGEMENT ARCHITECTURE

Name: Schultek, Brian Robert
University of Dayton
Advisor: Dr. Eric Balster

In this thesis, a novel software architecture called the Heterogeneous Computing Device Management Architecture (HCDMA) is introduced. The HCDMA is designed to address the growing problem of PCIe based acceleration device management. This type of architecture is ideal for computational acceleration in environments where size, weight and power need to be balanced for high performance computing solutions. The HCDMA, when coupled with an external PCIe expansion chassis, fills the need for a flexible and scalable solution to this problem. By utilizing the HCDMA with external FPGA acceleration modules, there is an observed 4.16 times improvement over the industry standard software solution for JPEG2000 image compression as well as a 2.94 times improvement over software based Image Pre-Processing algorithm tool chain.
To my entire family, specifically Jessica, and my friends for all of the encouragement, support and love.
ACKNOWLEDGMENTS

Thank you to everyone that helped in making this opportunity possible.

• Thank you to Dr. Eric Balster for putting up with my weird schedule and working with me despite my laughable commute to work. Without you pushing me continually to get this done, I would not be here today. You can finally relax a little, as your work here is done. Probably.

• Thank you to Bill Turri for giving me a shot years ago. I shudder to think where I would be today had it not been for us having the same car at lunch the day we met. Thanks for continuing to put your faith in me. I hope working together has been as enjoyable for you as it has been for me.

• Thank you to Dr. Frank Scarpino for serving on my thesis committee and for sharing your wisdom, experience and vast catalog of stories while working together. It is truly an honor and privilege to say I work with you.

• Thank you to Kerry Hill, Al Scarpelli, and the Air Force Research Laboratory for all of your lab space, resources, and financial support. Thank you for the opportunity to work with AFRL and the support you have given me and my coworkers in the ADDA Lab.

• Thank you to Andrew Kordik, David Walker and Daniel Shaffer for your patience in answering
all my questions about all things hardware and having patience with me while I toiled on this project.

- Thank you to the rest of my coworkers in the ADDA Lab and UDRI for the faith that you placed in me by allowing me to tackle some of these problems on my own and bring this project to life.

- Thank you to Ben Fortener for helping to bring me to UDRI, and helping me learn JPEG2000 when I started at UDRI. I truly appreciate your grace in allowing me to close our office door (or instant message you) and ask you questions when I was sure other people would just laugh at me for asking them.

- Thank you to Dr. Weber for serving on my thesis committee.

- Thank you to my friends for having such patience and grace with me over the past few years while going to school at nights. I appreciate you sticking by me even though I rarely had the time or energy to devote much time to our relationships.

- Lastly, thank you to my family (especially my parents and my incredible wife, Jessica) for supporting and helping me throughout my life. Your constant love, grace, support and wisdom has gotten me here and I am forever in your debts.
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................ iii
DEDICATION ........................................................................................................ iv
ACKNOWLEDGMENTS ......................................................................................... v
LIST OF FIGURES .............................................................................................. ix
LIST OF TABLES ................................................................................................. xi
I. INTRODUCTION .............................................................................................. 1

II. TECHNOLOGY BACKGROUND ...................................................................... 4
   2.1 Field-Programmable Gate Array (FPGA) .................................................. 4
   2.2 Peripheral Component Interconnect Express (PCIe) ................................. 5
   2.3 External PCIe Enclosure ........................................................................... 5
   2.4 Algorithms Currently Targeted for Acceleration ...................................... 7
      2.4.1 JPEG2000 ....................................................................................... 7
      2.4.2 RAW Image Pre-Processing ............................................................... 9

III. ARCHITECTURE DESIGN ........................................................................... 14
   3.1 Streamlining Hardware Initialization ....................................................... 14
      3.1.1 GiDEL Driver Background ............................................................... 15
      3.1.2 GidelFPGABoard (GFB) Object .................................................... 16
      3.1.3 GidelFPGABoard Factory (GFBF) Object ...................................... 18
   3.2 Run-Time Configuration .......................................................................... 19
   3.3 Leveraging Multi-Threaded Design Principals .......................................... 22
      3.3.1 Architecture Multi-Threading ........................................................... 22
      3.3.2 Algorithm Multi-Threading ............................................................... 24
   3.4 Execution Example Walkthrough .............................................................. 26
IV. RESULTS ................................................................. 30
  4.1 UDRI JPEG2000 Encoder ........................................ 30
  4.2 RAW Image Pre-Processing ...................................... 35

V. FUTURE WORK ...................................................... 39
  5.1 Plugin Expansion Capability ..................................... 39
  5.2 Service Oriented Architecture ................................... 39
  5.3 Distributed System .............................................. 40
  5.4 Code Refactoring ................................................ 40

BIBLIOGRAPHY ........................................................... 42

APPENDIX:

A. SUPPORTING INFORMATION ...................................... 44
  1.1 HCDMA Debug Configuration Example .......................... 44
  1.2 JPEG 2000 Software Encoder Comparisons .................... 47
  1.3 Addition Infrared RAW Image Pre-Processing Test Results . 50
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>OneStop Systems 4U External PCIe Enclosure. [1]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>From a single source image encoded with JPEG2000, multiple images may be produced at varying sizes and quality. [2]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Block diagram of the stages implemented in the UDRI Image Pre Processing Suite</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>An example pixel magnitude response to increasing black body radiation. The left shows the response prior to an offset and gain, and the right show the response after an offset and gain. It is clear why the more linear pixel response after applying an offset and gain is desired.</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>A subsection of sample image data before and after the pre-processing toolchain</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>A GiDEL ProcSTAR III and GiDEL ProcE IV, two examples of PCIe based FPGA cards displaying the array of possible hardware configurations</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>A representation of how the 3 layers of the GiDEL hardware interface operate. The Operating System Interface handles hardware communication and is installed in the operating system kernel. The Low-Level Proc Board Interface is a library that the HCDMA links with during compilation and the Generated Proc Board Interface is code included with the HCDMA and compiled directly into the HCDMA binary.</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Two examples of how the FPGA on a ProcE IV from GiDEL can be partitioned with algorithms</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>GiDEL FPGA C++ Object Factory Relationships</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>HCDMA Architecture design layout incorporating multiple threads for hardware monitoring and command and control in addition to separate threaded algorithm processors</td>
<td>23</td>
</tr>
</tbody>
</table>
3.6 Ideal design approach for a multi-threaded algorithm processor ........................................... 24
3.7 The flow of a client request through the objects and threads comprising an algorithm processing unit in the HCDMA ..................................................................................................................... 27
4.1 "Pentagon", the image data making up a single tile in the test imagery ........................................ 31
4.2 Sample 20x20 Tile Testing Image .................................................................................................... 32
4.3 Comparison of results when compressing 200, 400 and 600 tile sample images in both software and hardware with an increasing amount of available hardware ......................................................... 33
4.4 Comparison of the number of tiles dispatched to each available JPEG2000 core at various numbers of available cores .............................................................................................................................. 35
4.5 Sample input data used when testing the impacts of the HCDMA on infrared RAW image pre-processing ........................................................................................................................................... 36
4.6 Throughput performance comparison of the UDIPPS from Software only through 8 FPGA Cores running in parallel ........................................................................................................................................ 37
1.1 Comparison of JPEG2000 Software Encoders for 200 Tile Sample Frame..................................... 47
1.2 Comparison of JPEG2000 Software Encoders for 400 Tile Sample Frame..................................... 48
1.3 Comparison of JPEG2000 Software Encoders for 600 Tile Sample Frame..................................... 49
1.4 Impact of HCDMA on the Total Time Elapsed When Processing 2000 Frames in UDIPPS .................................................................................................................................................. 50
LIST OF TABLES

4.1 Comparison of Maximum Throughput in MB/s for Various Data Volumes . . . . 34
CHAPTER I

INTRODUCTION

For upwards of 30 years, engineers have been looking to additional devices to boost computational capabilities and offload processing requirements from the central processing units in computers, a process referred to as hardware acceleration (HA). HA often came in the form of additional application-specific integrated circuits (ASIC’s) which filled the role nicely, but are expensive and can take years to fully develop. If the customer’s needs change or there a defect is found in the ASIC during operation, getting an updated version can take up to several months before a replacement is available. In order to solve this problem, in the mid 1980’s Ross Freeman and Bernard Vonder-schmitt developed a computer chip called a field-programmable gate array (FPGA) that allows for the customer to reprogram the device to perform a desired function. The development of the FPGA led to a drastic drop in the amount of down time a customer would face due to defects or changing requirements ([3]).

ASIC’s and FPGA’s remained the primary hardware acceleration methods until the mid 2000’s when application programming interfaces (API’s) were released to enable the execution of custom algorithms on graphics processing units (GPU’s). Until the advent of these API’s, GPU devices were strictly limited to the rendering of graphics and driving computer displays, however API’s like NVIDIA’s CUDA development framework [4] allows customers to harness the computational power of a GPU pipeline for a wide variety of tasks. Additionally, as the market embraced CUDA
and it’s competitor OpenCL [5], devices like the NVIDIA Titan [6] started to hit the market that were targeted for high-performance computing that lacked any of the display driving abilities of the original GPU’s [7].

Alternatively, if a proper accelerator card based solution is unavailable, the other option for increasing performance has been to increase the number of available CPU’s in the system. This is an extremely common approach for servers, which now commonly house multiple high performance CPU’s on the motherboard. If still more performance is required such as is often the case in applications such as weather or ballistics simulation modeling, supercomputers can be employed to deliver incredible computational performance. With one example called the Blue Gene/L supercomputer from IBM housing 131,072 microprocessors and delivering a massive 135.3 trillion floating point operations per second, supercomputers are certainly a major player in the world of high-performance computing ([8]).

Unfortunately, some problems of today’s world don’t always allow for a large room and virtually limitless power and cooling to fuel a state of the art supercomputer, but still might need more of a boost than a single FPGA or GPU can deliver on their own. FPGA’s and GPU’s are traditionally integrated into a server, referred to as the host machine, via the use of the peripheral component interconnect express (PCIe) bus. Eventually, the host machine will run out of slots to enable the addition of supplementary resources or the machine will become overcrowded to the point of causing power consumption and cooling issues. This is often the case in many avionics applications where constraints are placed on the size, weight and power (SWaP) of onboard systems. There is an ever growing desire to expand the capabilities of such systems so the need for an intermediary solution to fit these constraints has been evident for some time now. In recent years, companies such as OneStop Systems [1] have taken up the challenge and developed a wide variety of expansion chassis aimed at extending a host system’s PCIe capabilities in such environments where adding
another server simply isn’t feasible. These solutions allow the offloading of compute device power and cooling from the host server while staying within operational constraints of low SWaP and high performance situations.

The major drawback of simply expanding the availability of PCIe slots to add resources is that, at present, there exists no such framework with which to easily adapt to and manage the additional resources. This thesis presents the Heterogeneous Computing Device Management Architecture (HCDMA) to address the problem of additional resource management of PCIe expansion devices for low SWaP, high performance computing solutions. This architecture allows for the needed flexibility and scalability offered by an expansion chassis to be leveraged efficiently by various applications. The use of the HCDMA has allowed for a highly flexible implementation of the JPEG2000 solution yielding up to an approximated 4.16 times throughput gain over the industry leading software as well as an approximated 2.94 times throughput gain over comparable software by utilizing an FPGA based implementation of the Image Pre Processing algorithms. Following the introduction, Chapter 2 provides background on the technologies leveraged to build the HCDMA, Chapter 3 provides details of how the HCDMA is designed, Chapter 4 looks at the testing of and results yielded by the HCDMA, and finally Chapter Y gives conclusions and future research work.
CHAPTER II

TECHNOLOGY BACKGROUND

When simply implementing a multi-threaded approach fails to yield desired throughput, the development and implementation of code to utilize an additional acceleration card can be quite beneficial. In order to dive a little deeper, it is important to have an understanding of all of the key components that are used to enabled the HCDMA to be so powerful.

2.1 Field-Programmable Gate Array (FPGA)

FPGA’s are semiconductor devices that offer the ability to be programmed after being manufactured, as opposed to application-specific integrated circuits (ASIC’s) which are produced to a specification for a particular purpose. The field-programmable aspect refers to the device’s inherit ability to be reconfigured after manufacturing to suit a variety of needs [9].

FPGA’s are made up of reconfigurable components called logic elements (LE’s) and a series of reconfigurable interconnects that allow the LE’s to be physically interconnected in different ways. The way in which these elements are connected is defined by a binary file that is flashed to the device. The binary files are the compiled results of code written in a hardware descriptive language such as Verilog or VHDL [9].

There are many benefits to FPGA’s over other computing devices such as offering a faster time to market over traditional ASIC solutions, reconfigurability to fit a desired purpose, and lower NRE
costs overall. In addition, FPGA’s typically run much lower power than GPGPU’s or CPU devices, some of which can consume up to 375 watts per card [10], whereas FPGA’s typically operate in the sub-50 watt range. Also, due to their size, FPGA devices typically can fit in a smaller chassis than most commercial GPGPU solutions, making them ideal solutions in size, weight and power (SWaP) constrained environments.

2.2 Peripheral Component Interconnect Express (PCIe)

The communications backplane that enables a majority of today’s external acceleration cards is called the Peripheral Component Interconnect Express bus. PCI Express is the third generation of I/O bus and it provides a high speed and high performance technology for interconnecting peripheral devices [11]. A PCIe link is a physical connection is made between two devices and can be either 1, 2, 4, 8, 12, 16 or 32 lanes wide and is denoted as the number of lanes with a leading x as in x1, x2, x4 and so on [11]. A single lane is composed of two differing signal pair with a pair of wires for sending data and a pair of wires for receiving data. So a x1 link (which has 1 lane) results in 4 total signals to and from the PCIe device [11]. Likewise, a x32 link consists of 32 lanes resulting in a total of 128 signals to and from the device. The links are created for a certain width and frequency automatically by the devices at each end of the link during hardware initialization [11].

For communication between devices, PCIe encodes messages via a packet based protocol [11]. This allows the messages to be transmitted and received in a serial manner and enabled byte striping across the available lanes in the link. There are various packets defined for memory access, I/O, configuration, messages and completion requests [11].

2.3 External PCIe Enclosure

Unfortunately, one of the limiting factors for PCIe device utilization for a highly scalable system is the fact that a majority of the servers that are on the market have only limited physical room for
the addition of PCIe resources. This limitation often means that the system to be designed ends up being strictly scoped to solving a single problem, rather than being able to be flexible. This is especially the case with GPGPU assets, which can sometimes be too large to fit into most server chassis.

Some implementations work around this problem by using a cluster of servers, such as the Maxwell supercomputer hosted at the University of Edinburgh, which sports 64 FPGA’s across 32 IBM Intel Xeon blade servers in 5 IBM BladeCentre chassis [12]. By building a cluster of machines the problem can be easily overcome, however in environments where size, weight and power do not afford the luxury of throwing more host machines at the problem, a more elegant solution must be used. It is this reason why external PCIe enclosures have been developed such as One Stop Systems, such as the one seen in Figure 2.1.

![One Stop Systems 4U External PCIe Enclosure](image)

Figure 2.1: OneStop Systems 4U External PCIe Enclosure. [1]

The external enclosure, shown in Figure 2.1, utilizes a expansion card installed in the host machine and a PCIe cable to provide the additional PCIe real estate needed to form a more complete flexible and extendable system. The enclosure supports 4 cannister, with each cannister capable of holding 4 long-length PCIe devices, 8 short-length PCIe devices or several combinations of varying length assets. The ability to provide additional power for the cards is also accessible for the assets which is not always provided in server chassis. Additionally, since the chassis is built to utilize
the latest generation of PCIe capabilities (generation 3), the cannisters are able to be removed and inserted if they need to be for any reason during operation without cycling the power to the entire system or even taking down the host machine (a feature known as hot-swapping). The enclosure is also capable of being managed remotely via Intelligent Platform Management Interface (IPMI) and can be associated with multiple servers at once if need be. Each cannister provides dedicated power rails as well as 1 dedicated 12 millimeter fan in order to provide optimal airflow and keep the PCIe resources properly cooled.

2.4 Algorithms Currently Targeted for Acceleration

2.4.1 JPEG2000

JPEG2000 is the most recent image compression standard from the Joint Pictures Expert Group [13]. Established as an ISO standard in 2000 and revised in 2004, the JPEG2000 standard is known for offering greater compression performance over JPEG (its predecessor). At higher bit rates, JPEG2000 is capable of producing imagery with an additional 20% compression over similar JPEG imagery while performing much better at lower bit rates [2].

A primary difference between the JPEG2000 and JPEG standards is the encoding mechanism. JPEG employs a discrete cosine transform in order to compress data passed through its encoder, while JPEG2000 utilizes a more advanced wavelet transform in order to produce better results. Using the wavelet transformation, images broken into tile components are decomposed into different levels called wavelet levels. These levels contain sub bands that describe the horizontal and vertical spacial frequency characteristics of the original data. Each of these levels is related to each successive level by a factor of two meaning that each successive level has about half of the horizontal and vertical resolution as the previous level, making it possible to generate various resolution images from the same compressed source image by decoding only a subset of the sub-bands [13].
Figure 2.2: From a single source image encoded with JPEG2000, multiple images may be produced at varying sizes and quality. [2]

Sub-bands of the tiles can then be further divided into rectangular arrays of coefficients called code-blocks and then each individual bit-plane is coded with three coding passes. The coding passes collect information about the context of the bit-plane compressed image data which is subsequently used by an arithmetic coder in order to decode the compressed bit-stream [13]. The end result is a compressed image bit-stream that supports random code-stream access and processing, resiliency against data errors, and the ability to decompress the image in a number of ways to support a wide variety of use-cases as can be seen in Figure 2.2 [2].

UDRI has developed and fielded an implementation of a FPGA core for JPEG2000 encoding [14] [15] [16] [17]. This implementation takes an input image and divides it into tiles of size 1024 pixels wide by 1024 pixels high. These tiles are then grouped and dispatched to the available FPGA cores for compression, also known as the Tier I encoding process. A data stream containing this Tier I data and associated metedata is then sent back to the software in order to complete processing to produce the final encoded file stream, which is known as the Tier II encoding process.
2.4.2 RAW Image Pre-Processing

Some imagery, such as infrared (IR) imagery, typically requires more processing than electro-optic imagery in order to adjust the data so that it is visibly more meaningful to a viewer. UDRI has developed an image pre-processing pipeline similar to the pipeline seen in Figure 2.3 to handle some of this processing called the UDRI Image Pre Processing Suite (UDIPPS). IR focal planes have less uniform pixel response across the field of view as well as have significantly higher probability of bad pixels over electro-optic or visible sensors, mostly due to limitations of the materials used in the IR focal plane array [18]. The bad pixel replacement (BPR) removes possible distracting false positives in projected IR imagery, while the non-uniformity correction (NUC) helps maintain uniform contrast and intensity relationship across the field of view. Non-uniformity can be a distraction and cause true negatives as the data becomes less distinct. These bad pixels and visible variations across an image must be corrected as much as possible to provide good image content to an image analyst. An unsharp mask (USM) is a filtering method that produces a sharpening effect. It typically employs a low-pass and/or high-pass filter to apply a gain to the high pass coefficients and/or, conversely, to attenuate the low pass coefficients. Histogram specification (HS) is the process of transforming the raw image data such that its histogram more closely resembles a desired (specified) histogram. This typically allows for a higher dynamic range to better visually show the important content in the image. Finally, since pixel values from an IR sensor can typically be captured at greater than 8-bits of precision the dynamic range of the data must be compressed to within 8-bits in order to visually analyze the data, since most monitors, however, are only capable of displaying 8-bits of precision. Dynamic range compression (DRC) compresses high precision data into a smaller range in order to achieve this.

Non-uniformity correction is a correction process mainly utilized in infrared imagery. Detector nonuniformity is a phenomenon adversely affecting many imaging systems, particularly infrared
In a focal plane array, the responsivity of the individual photodetectors will vary from detector to detector. This creates a fixed pattern noise that degrades the acquired imagery [20]. Furthermore, the nonuniformity tends to drift with time and environmental conditions, making a one-time factory calibration insufficient [19]. In order to correct issues in pixel response, offsets and gains are applied to the raw imagery on a per-pixel basis. These offsets and gains are calculated by radiating the sensor at the spectrum desired and then measuring the response of each pixel as the intensity at the sensor increases. Every pixel, due to deficiencies in the manufacturing process, sensor operating temperature, and other effects, will respond at a different rate to the increasing radiation [21]. Figure 2.4 shows an example pixel response prior to NUC and after applying NUC. The UDRI preprocessing pipeline utilizes Scene-based NUC (SBNUC), which is the real-time calculation of the offsets and gains applied in the NUC algorithm, updated for every new image in the system. When a sensor is recording images of the same scene for many frames at a time, it allows for time-averaging to help an algorithm decide what response a pixel is showing in general to the scenery. This localized time-average can be a simple mean of the difference of a localized moving average filter. By having a scene average, this response can then be normalized by the NUC algorithm to correct for non-uniformities across the sensor array.
Figure 2.4: An example pixel magnitude response to increasing black body radiation. The left shows the response prior to an offset and gain, and the right show the response after an offset and gain. It is clear why the more linear pixel response after applying an offset and gain is desired.

The bad pixel replacement (BPR) algorithm takes in a matrix of bits the same size as the image dimensions called a bad pixel map (BPM), where each bit represents a pixel in the corresponding image. For each bad pixel (marked as a 1 in the BPM), BPR substitutes the corresponding pixel in the source imagery with an average of its neighboring pixels in the resultant imagery. This results in a more accurate pixel value in the output imagery than the dead or stuck pixel data that was present in the source.

Following the BPR process, an unsharp mask (USM) filter is applied to sharpen the imagery. The reasoning behind sharpening the image is that, especially with large-area, infrared sensors, any small targets on the scale of 1 to 2 pixels are quickly lost in the degradation of high spatial-frequency content of an image, especially after image compression. Sharpening the image prior to compression using the unsharp mask algorithm described here has been shown to be the best solution to preserving small targets throughout compression [22]. The implementation of the algorithm here uses a 7x7 Gaussian filter kernel to achieve the low-pass coefficients of an image. The low pass coefficients are then subtracted from the original image to yield “high pass” coefficients. The low
pass coefficients are weighted by 0.2 in order to attenuate them before adding them back to the high pass coefficients.

After USM, the image data is then routed through a histogram specification (HS) and dynamic range compression module (DRC). Typically, the majority of the dynamic range in the raw data size (e.g. 14-bits per pixel) is unused. This could be due to a number of things, including poor pixel response due to how susceptible the sensor is to environmental noise. A histogram normalization or specification is one way to modify the pixel values to spread the histogram across the entire dynamic range, producing a much more useful (at least visually useful) image. In histogram specification, the cumulative histogram can be thought of as a cumulative distribution function of the image. Some sort of mapping can be specified (a piecewise linear approximation of a sigmoid or a true sigmoid function have been shown to provide very good results [23]) for transforming the cumulative histogram of the image to be closer to that of the specified function. This results in stretching the dynamic range of the values of the image that occur most frequently. Overall, a much larger portion of the dynamic range is used and thus provides more contrast and information to a viewer.

With most infrared imagery, its dynamic range must be reduced to 8-bits per pixel before being able to view it. This can be done simply by mapping the minimum and maximum values of the image at its raw bit-depth to the smaller 8-bits per pixel minimum and maximum values, 0 and 255, respectively. This mapping can be linear or shaped to be more like the cumulative histogram, but selecting the same mapping used for the histogram specification would allow the combining of histogram specification along with dynamic range compression in one lookup table. UDRI implemented this approach in order to reduce complexity, size of hardware, and latency in the hardware chain. At the end of this process, the adjusted image data is sent back into the host system from the hardware for whatever processing is to be done next. This could be anything from stitching and...
projecting the imagery onto terrain maps to compressing the imagery for storage depending on the
requirements of the processing chain. Figure 2.5 below highlights the dramatic effect that the image
preprocessing chain of algorithms has on imagery.

Figure 2.5: A subsection of sample image data before and after the pre-processing toolchain
CHAPTER III

ARCHITECTURE DESIGN

The HCDMA is designed to achieve maximum flexibility and scalability by leveraging any available hardware acceleration assets in a run-time reconfigurable way. Abstracting specific hardware software interfaces allows algorithm implementations to interface with different hardware platforms without necessitating any code changes via a uniform class specification. Additionally, algorithm optimizations such as multi-threading and request dispatching enable the HCDMA to optimally utilize all available hardware assets in the most effective available manner. Configuration at run time is enabled via a XML specification allowing the end user to either let the HCDMA choose the hardware each algorithm runs with or target specific hardware devices for operation.

3.1 Streamlining Hardware Initialization

One of the main ways in which the HCDMA is able to achieve maximum scalability and flexibility is via its control and exposure of available hardware platforms. Typically each algorithm implementation has had to be tightly coupled with the targeted hardware in order to properly utilize all available resources. For example, some PCIe based FPGA platforms feature more than 1 FPGA asset on a single board. By abstracting specific hardware object definitions away into a common interface, algorithms are able to simply interface with an array of hardware without necessarily having to know any details about the hardware itself.
3.1.1 GiDEL Driver Background

GiDEL’s FPGA boards come in a variety of shapes, sizes, and FPGA fabrics such as the two seen in Figure 3.2. When working with one of their boards, developers start by laying out the necessary components in their ProcWizard tool. This tool allows for the developer to create bi-directional registers, bi-directional first in, first out (FIFO) data structures and off-chip DRAM storage. Because each FPGA fabric is different, the ProcWizard tool requires the developer to target a specific board when building their design. Once the layout is complete, the tool can generate C++ code, HDL code and documentation to match the specified design which can then be used to further develop the other aspects of the software and hardware system.

Figure 3.1: A GiDEL ProcSTAR III and GiDEL ProcE IV, two examples of PCIe based FPGA cards displaying the array of possible hardware configurations

From a software perspective, the GiDEL hardware interface consists of three layers called the Operating System Interface (OSI), the Low-Level Proc Board Interface and the Generated Proc Board Interface [24]. The OSI is implemented in the form of a kernel level driver called the Target Driver. This driver is automatically installed with the installation of the GiDEL libraries and is not actually directly connected to any user code [24]. The Low-Level Proc Board Interface, also called the Board Driver, is a class that is implemented in the GiDEL libraries which provides the tools to communicate with the board [24]. The third layer (the Generated Proc Board Interface) is the C++ code that ProcWizard generates which is derived from the Board Driver class [24]. The
Application Driver directly represents the hardware that was designed via ProcWizard and is used to communicate with the corresponding generated HDL on the board via the PCI bus [24].

![Diagram of GiDEL hardware interface layers](image)

Figure 3.2: A representation of how the 3 layers of the GiDEL hardware interface operate. The Operating System Interface handles hardware communication and is installed in the operating system kernel. The Low-Level Proc Board Interface is a library that the HCDMA links with during compilation and the Generated Proc Board Interface is code included with the HCDMA and compiled directly into the HCDMA binary.

This driver is responsible for allocating the memory that will be used for communicating with the FPGA board in the software and also initializes the hardware with its associated raw binary file (RBF) that was generated with the HDL produced in ProcWizard [24]. Together these two drivers enable software developers to interface with hardware in an intuitive way and form what GiDEL refers to as the Proc API [24].

### 3.1.2 GidelFPGABoard (GFB) Object

Due to the way that the Proc API tightly couples algorithm interface points with the targeted hardware, it became necessary to wrap the classes that are generated by the ProcWizard tool for describe each hardware configuration option within a custom class for use in the HCDMA. Each
of these Generated Proc Board Interface objects generated by GiDEL is contained within a custom object that inherits from a software interface called a GidelFPGABoard (GFB) object. Since the board driver itself takes care of the initialization of all assets the wrapper class is in charge of creating and managing an instance of the board driver for use in the HCDMA. Once the board driver has been created and initialized successfully, each object of type GFB then proceed to gather any interface points into arrays for easy access by the algorithms.

Figure 3.3: Two examples of how the FPGA on a ProcE IV from GiDEL can be partitioned with algorithms

For example, consider one design that contains 1 core for JPEG2000 compression like sample configuration A in Figure 3.3. For this design there is 1 offset specifying where the software should create a handle for sending data to the board via DMA transfer, 1 offset specifying where the software can read data from the board via DMA transfer and a handful of read and write registers designed to facilitate control of the system. Another design may contain 3 cores of JPEG2000 like configuration B in Figure 3.3, thereby tripling all available offsets and registers that need to be managed and utilized by the algorithm during processing.

In order to simplify providing access to these items to the algorithms that need them, the GidelFPGABoard object adds each offset or register to a specific collection. This way, when an algorithm
is setting up it’s hardware resources, all it needs to do is to ask each board for the various collections it has made. By wrapping the board driver in a custom class and leveraging the fact that all of the generated objects that communicate with the hardware are Proc objects, the HCDMA is able to provide algorithms with access to FPGA acceleration hardware in a way that they can be utilized seamlessly, regardless of their actual physical specifications.

### 3.1.3 GidelFPGABoard Factory (GFBF) Object

Upon initialization of the HCDMA, a configuration file is processed which dictates how the hardware resources in the host machine are to be utilized. If the desired configuration does not match the physical configuration, the architecture exits and notifies the user to ensure that the desired configuration is valid to ensure stability of the system. Once the requested configuration is validated by the HCDMA, the system uses the factory pattern in order to produce GFB objects. Requests are made to an object called the GidelFPGABoardFactory (GFBF) that produces GFB objects of the requested type as is shown in Figure 3.4. Each of these objects are specifically targeted to PCIe hardware slots in order to allow for maximum reconfigurability and control over how the hardware is utilized.

Once the creation of each of the GFB objects is complete, the pointers for each available memory controller are collected and the boards are initialized for use by each algorithm. This allows the algorithm interfacing with the hardware to simply iterate through all of the memory addresses available in order to allocate the necessary DMA handles against them for data transfer to the hardware. With these memory pointers already collected, the algorithms do not need to know what hardware those pointers actually reference, just so long as they can be utilized correctly.
3.2 Run-Time Configuration

In order to ensure the HCDMA's is as flexible as possible, the first thing that is done when the architecture is starting up is that a XML file defining the desired operating parameters is processed and used to initialize the system. Board elements like those in Listing III.1 at the top of the configuration file contain specify how hardware elements should be initialized by the HCDMA. These elements can be set up to allow the HCDMA to select the appropriate physical board in the host system to utilize for an algorithm or alternatively allow the user to configure anything from specifying which banks of memory to utilize during processing to which exact board to target by specifying PCIe slot number. Additional elements in the configuration file are grouped by the algorithm that they are relevant to.

Listing III.1: Example Board Settings Subsection of the Run-Time Configuration File for the HCDMA

```xml
1  <boards>
2    <numCardsToLoad title="Number of FPGA Cards to be used for compression" type="integer">2</numCardsToLoad>
3  </boards>
4  <board0>
5    <boardType list="PROC_STARIII_150_4, ProceIV_530" title="FPGA Board Type" type="list">ProceIV_530</boardType>
```
There is a section for settings, seen in Listing III.2, relating to the HCDMA itself which contains entries for setting the level of logging to be produced by the system, the destination file system path where the logs are to be created in addition to the thresholds to be used by the HCDMA for hardware asset temperature monitoring.

Listing III.2: Example System Settings Subsection of the Run-Time Configuration File for the HCDMA

```xml
<settings>
  <desiredLogLvl title="Desired logging level to produce. 0 is least, 3 is as much as possible. Default is 0." type="integer">3</desiredLogLvl>
  <desiredLogPath title="Desired server log path. (Note: needs a trailing slash. Ex. /home/user/logs/)" type="string">./</desiredLogPath>
  <temperature>
    <pollingInterval title="Desired number of milliseconds between temperature polling." type="integer">10</pollingInterval>
    <swToHwThreshold title="The temperature at which the server will resume processing data on the hardware" type="integer">75</swToHwThreshold>
    <hwToSwThreshold title="The temperature at which the server will switch processing to software to allow the hardware to cool down" type="integer">80</hwToSwThreshold>
    <unloadHwThreshold title="The temperature at which the hardware will be unloaded in order to prevent overheating damages" type="integer">83</unloadHwThreshold>
    <reloadHwThreshold title="The temperature at which the hardware is safe to be reloaded without immediate risk of overheating damage" type="integer">73</reloadHwThreshold>
  </temperature>
</settings>
```

For both of the JPEG2000 and Image Pre-Processing modules, settings are available which enable debugging parameters since any image-specific configurations, such as quantization levels in JPEG2000, are passed in with the client’s processing request. One of the configurable parameters for the JPEG2000 algorithm allows the specification of the number of request processing groups...
to initialize, which has a default value of 10. Thus there will be up to 10 requests going through the processing chain for the algorithm in question. This value allows the HCDMA to be scaled for the amount of available memory in the system, as each request processing group that is created will allocate enough memory to transfer the maximum amount of data to and from each available hardware core. Listing III.2 shows an example of some of these settings, while Listing A.1 available in Appendix 1.1 provides a comprehensive list of all available parameters.

Listing III.3: Example Algorithm Processor Settings Subsection of the Run-Time Configuration File for the HCDMA

```
  <j2k>
  <dmaHandleCount title="Specify the number of DMA handles to create for the J2K process to use. More handles means more memory, but more requests can be queued up for hardware." type="integer">10</dmaHandleCount>
  </j2k>
  <preprocessing>
  <histogram>
  <histogramType title="Desired histogram type to run. Default is Linear." list="Linear, AOG, Sigmoid" type="list">Linear</histogramType>
  </histogram>
  <dmaHandleCount title="Specify the number of DMA handles to create for the PreProcessing process to use. More handles means more memory, but more requests can be queued up for hardware." type="integer">10</dmaHandleCount>
  </preprocessing>
```

The other configurable value of note corresponding to both the JPEG2000 and Image Pre-Processing modules tells the HCDMA to save the DMA buffers it transfers to and from the targeted hardware for each request. This is specifically marked as being intended for debugging purposes only as it has a severe impact on the latency of the system, especially with running with a high core count or large imagery. The ability to dump out exactly what the hardware itself is sent and what is returned has proven to be an invaluable asset in debugging when adding new features to the hardware implementation. Listing A.1 available in Appendix 1.1 provides a sample XML configuration file containing this flag.

For Image Pre-Processing, the HCDMA also allows for the configuration of the desired histogram specification calculation method and the values used by the desired method to calculate a
new histogram table to be applied to the imagery. While these values can also be changed by sending a control signal to the HCDMA directly during run-time operation, the option to set these values up front and not configure them manually allows for quick and easy integration with a number of platforms that might not care to optimize these calculations.

3.3 Leveraging Multi-Threaded Design Principals

In order to ensure the HCDMA is maximizing its utilization of available hardware assets, it is necessary to implement algorithms and board interactions in such a way as to be independent from one another. While one request is being processed by the hardware, the architecture should be spending time preparing subsequent requests or returning previously completed requests.

3.3.1 Architecture Multi-.Threading

In order to minimize any potential impacts of running a particular algorithm in an architecture meant to encompass many types of algorithms, each algorithm is to start and maintain its own assortment of threads separate from the main thread. The main thread itself is responsible for processing the run-time configuration, initializing and managing any assets as requested as well as operating as the main communications relay for the system and is represented in Figure 3.5. Additionally, the HCDMA spawns an additional thread charged with periodically polling any available hardware devices to ensure they are operating within ideal temperature thresholds.

On startup main command and control thread processes the run-time configuration from the provided XML file, and initializes the main log file based on the specification found in the configuration. Then, the hardware assets requested in the configuration are mapped to physical boards and validated. The validation process checks to ensure that the host system has access to the requested number of boards, if not more. Once that passes each board configuration is processed and matched with a physical asset from the host system. This amounts to verifying that the specified board type
has an implementation of the specified algorithm, as well as attempting to target a specific board in the system if the configuration specifies to do so. If the configuration requests a board be utilized from a PCIe slot that does not actually register as having an asset, then the HCDMA exits and alerts the user to the issue.

Once the hardware configuration is successfully mapped to the physical boards in the host system, the desired algorithm processors are started up and provided with the requested hardware assets. After initialization, these algorithm processors actually spawn their own threads to be able to run separately from one another and the main system while the HCDMA’s main thread drops into a request processing loop.

The request processing loop polls the named pipe used by clients to submit requests or control the operation of the HCDMA for as long as the architecture is online. If a request is read from this pipe that is intended for an algorithm processor that has successfully been previously spawned, then it is passed to the target algorithm for processing. However if a request is received that is intended to be dispatched to an algorithm that is not running, an error is logged and the system continues on.
The request pipe may also contain control commands for the HCDMA, such as shutting down of the architecture which is the mechanism that is used for a graceful shutdown or restart.

### 3.3.2 Algorithm Multi-Threading

During the initial analysis of the algorithms to accelerate, it became apparent that much of the time a request spent going through the server could actually be overlapped with another request. For example, suppose that it takes 100 milliseconds to copy data to the appropriate DMA buffers for transferring the data to the hardware and then that section of the code is idle for another 300 milliseconds while the data is processed by the hardware before taking 50 milliseconds to be copied back into the response buffer for the client. One of the main reasons to move towards a multi-threaded approach was to remove this overlap.

![Image of algorithm multi-threading](image.png)

**Figure 3.6: Ideal design approach for a multi-threaded algorithm processor**

In the HCDMA, algorithm processors ideally incorporate approach portrayed in Figure 3.6. Each algorithm processor consists of a controller object that spawns a dispatcher thread, ensuring
that the main command and control thread of the HCDMA does not interfere with the algorithm’s processing abilities. This algorithm processor object then creates and manages what are called the Client Communication (or Processor) object and Hardware Communication object. The Processor objects handle any client communication aspects such as parsing the incoming request, setting up the DMA buffers, parsing hardware responses and sending the data back to the client for any post-processing needs. Hardware Communication objects are responsible for handling all things that have to do with the hardware such as creating DMA handles, sending data to and from the hardware and setting values for any registers that the hardware might need to operate.

Both of these communication objects also contain multiple threads to allow for a total of four independent stages: processing client requests, sending data to hardware, receiving data from hardware and creating client responses. This way, up to four requests can be processed at any given time if needed, enabling as much overlapping of processing as possible to ensure optimum performance. Requests are passed from stage to stage via a thread-safe queue which allows for fault tolerance if any on piece experiences a failure as well as enabling all stages to run at their optimum speed without being held up by any other stage. In a failure scenario, any request can simply be re-enqueued in the stage that failed for a second attempt, preventing any loss of data.

One of the benefits of this division of labor is that it gains the ability to enable a pipelined mode of operation should the hardware support it. The send and receive data threads must be coupled through use of a mutex; once a request has been sent to the target hardware for processing, a mutex is locked until that same request has finished processing and the processed data has been received by the HCDMA. Some algorithms require a reset after processing before any additional data can be processed which would cause problems if triggered in the midst of processing. If this reset is not required, however, the threads can operate independently resulting in an optimal utilization scenario
where the data transfer thread can constantly stream data to the hardware while the data receiver
thread constantly reads processed data and passes it out to be returned to the proper client.

3.4 Execution Example Walkthrough

Prior to starting up the HCDMA, the user must edit the run-time configuration file to represent
the desired system configuration to reflect their desired hardware utilization, in addition to adjusting
any of the general settings such as the logging output level and location. For pre-processing tests,
configuring the thresholds used by the histogram calculation process is an optional step that should
be considered as well. With the configuration properly set, the HCDMA can be launched. With the
initial thread launched, the HCDMA proceeds to parse the XML in the provided configuration file
and begins to verify its contents. The initial logging object is created and begins dumping output
to the screen and corresponding log file in the desired location. The HCDMA then scans the host
machine to determine which hardware assets are available and attempts to match what resources it
finds to any that might have been specified in the configuration file.

One available hardware has been assigned for use by the desired algorithms, the HCDMA's
main processing thread instantiates the necessary algorithm controller objects. The number of ob-
jects started at this point is entirely dependent on the utilized hardware and the data parallelization
abilities of the algorithm. For JPEG2000, there is simply one algorithm controller dispatching data
to all available hardware cores while the UDIPPS will have a separate algorithm controller object
allocated with a single FPGA core for each requested instance of the algorithm.

Upon being created, each algorithm controller will spawn any necessary threads needed for
optimal throughput. In the current design, this consists of 2 threads contained in an object tasked
with board interaction and 2 threads in an object aimed at client interactions. Once all objects and
threads have been initialized, the various threads fall into an idle state in their processing loops that
allow them to periodically check for incoming data to be processed and be asleep the rest of the time. Figure 3.7 shows how these objects are separated and related.

Once a client submits a request, the HCDMA’s main processing loop determines what type of processing the request is for and passes along the identifier of the named pipe in which the request resides to the appropriate algorithm processor’s dispatcher thread. The dispatcher thread passes the request over to the client communications object (arrow 1 in Figure 3.7) in order for the Client Request Receiver Thread to read the request information out of the client’s request pipe and prepare the associated image data for transfer to the hardware. In Pre Processing, this image data is simply copied directly into the memory allocated for DMA transfer. For JPEG2000, this means determining the most appropriate way to distribute tiles amongst the available compression cores.
based on the number of tiles in the input image and the optimal number of tiles to be sent to each core and copying the image data into the appropriate memory space to be transferred to the boards. In addition to the image data, JPEG2000 also requires that each buffer of tiles to be compressed contain a header containing pertinent metadata for compression such as an array of quantization values, the count of the number of tiles in the buffer and the DWT configuration to use, as the UDRI developed core offers both the 5/3 (reversible) and 9/7 (irreversible) DWT transforms.

A message containing the DMA handles populated with image data and some accompanying request metadata is then constructed and passed over to the board communications object (arrow 2 in 3.7) to be enqueued for processing by the Hardware Transmission Thread; the thread responsible for sending data to the hardware. The Hardware Transmission Thread does nothing more than initiate and complete DMA transfers from the host server to the PCIe based FPGA board(s) and initiate any command and control sequences required by the hardware design. For the JPEG2000 implementation, this requires initiating and completing DMA transfers to as many cores as is required by the imagery while UDIPPS only has to manage one DMA transfer to the hardware due to its design. The Hardware Transmission Thread then passes the message it has processed over to the Hardware Receiver Thread which is responsible for receiving information off of the hardware where it will be populated with the results from hardware (denoted by arrow 3 in 3.7).

Initially, the data may not be ready for retrieval so the Hardware Receiver Thread might have to loop while checking the value of a status feedback register on the board. Checking the register prevents the software from executing a DMA transfer on the board before the data has fully completed it’s processing and prevents any DMA timeout warnings or errors. Once processing is completed, the Hardware Receiver Thread initiates a DMA transfer to retrieve the data from the hardware and passes the message it was working on (now containing the processed data from the board) back to
the client communications object to be handled by the Client Response Transmission Thread as is indicated by arrow 4 in 3.7.

In this final stage the message’s contents are copied back out of the HCDMA and a response is placed in a named pipe that signals that the client can proceed with any post-processing steps that might be needed. Thus for JPEG2000, the client can run the optimal truncation process and produce the final image buffer and for UDIPPS this means simply writing out the resultant image data to be saved.

That is how one request from one client flows through the HCDMA’s processing structures, however the architecture is designed to be processing many requests from many different clients at once so its easy to see why the separation of algorithms and responsibilities provided in the HCDMA is such a powerful tool.
CHAPTER IV

RESULTS

4.1 UDRI JPEG2000 Encoder

To test the scalability of the HCDMA, 8 FPGA cards are loaded into the external PCIe enclosure attached to a host server containing 4 Intel Xeon E7-8870 processors and 256 gigabytes of DDR3 RAM.

The image used for all testing is given in Figure 4.1 and is called "Pentagon". "Pentagon" is a desirable image selection due to the high contrast differences between sections of the pentagon, which provide an ample challenge for the encoder and tend to be more difficult than typical aerial imagery is to compress. Therefore, the numbers produced by compressing an aerial image of pentagon should be on the lower end of what the encoder is capable of producing for more realistic data.

"Pentagon" is 1024 pixels high by 1024 pixels wide, as the UDRI J2K encoder is designed to operate primarily on tiles of this size. In order to generate test images suitable for testing the capabilities of the system, the "Pentagon" image is repeated in order to create various test imagery to exercise the capabilities of the system at different data amounts. For example, the image given in Figure 4.2 shows an example one such test image containing 400 total tiles in a 20x20 tile formation (20480x20480 pixels). Together with this generated imagery as the test data, we are able to vary
the levels of hardware available to the HCDMA and produce an idea of how the scalability and flexibility offered by the HCDMA impact the data throughput of compression requests.

We test the system by having a client submit requests for JPEG2000 compression to the HCDMA. Without varying how the client operates, we are able to change the HCDMA’s parameters between each set of compression requests in order to get a good look at how increasing the number of resources available to the HCDMA can impact the throughput.

Once each available hardware configuration has been tested, results are compiled and charted in order to visualize the impact of hardware resources on the throughput as calculated in megabytes per second. The time being used for calculating this throughput number is a combination of the time it takes to send the data via DMA transfer to each core, how long each core takes to encode the supplied data and the amount of time it takes to return the data via DMA transfer. As can be seen from Figure 4.3, the throughput for the 400 tile sample image starts out very low at roughly 48 MB/s and increases nicely until around seven or eight cores are utilized at which the throughput plateaus. While the throughput numbers continue to rise slightly at this point, this shows there is a threshold at which the overhead for core management starts to interfere with throughput performance. The same observation can be made about the 200 tile and 600 tile compression tests.

Figure 4.1: "Pentagon", the image data making up a single tile in the test imagery
The results in Figure 4.3 highlight the results of these 3 sets of configuration sweeps. These tests are also compared with the results of compressing the same sample imagery with the Kakadu JPEG2000 software encoder utilizing 64 threads (see Appendix 1.2 for additional results), which is portrayed as a flat line since the software doesn’t scale the same way hardware does.

The results comparison clearly highlights the benefits provided by utilizing hardware acceleration, as the software results for all data levels are similar to the result seen by the 2 core test result which is somewhere around 75 megabytes per second of throughput. Additionally, it is clear that the three throughput curves are similar in shape with the variation being where the knee of the throughput curve occurs. This is expected, as smaller imagery will need less hardware than larger images for optimal throughput lest the overhead of managing superfluous cores become the majority of the
Figure 4.3: Comparison of results when compressing 200, 400 and 600 tile sample images in both software and hardware with an increasing amount of available hardware

As such, the knee for the 200 tile count occurs at a lower core count than it does in the 400 or 600 tile tests, as is expected because the data sizes will hit their optimal hardware utilization to achieve maximum throughput at different hardware utilization levels. Table 4.1 lays out what these maximum throughput levels are and how they compare with Kakadu at the data volumes tested.

Taking a look at the average number of tiles per hardware request allows a determination to be made as to whether or not there is an ideal hardware resource utilization level. This determination allows is important in order to tweak the way data is dispatched to the available hardware optimally.
Table 4.1: Comparison of Maximum Throughput in MB/s for Various Data Volumes

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Kakadu</th>
<th>HCDMA</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 Tiles</td>
<td>72.7</td>
<td>194.3</td>
<td>2.67x</td>
</tr>
<tr>
<td>400 Tiles</td>
<td>77.5</td>
<td>259.8</td>
<td>3.35x</td>
</tr>
<tr>
<td>600 Tiles</td>
<td>75.7</td>
<td>315.2</td>
<td>4.16x</td>
</tr>
</tbody>
</table>

The comparison can be seen in Figure 4.4 and allows an initial correlation to be drawn based on the matching throughput at that request from 4.3. Testing with the 400 tile image from Figure 4.2 results in the HCDMA making seven individual requests to the hardware in order to compress the full frame when only 1 core is available. The first request to hardware in this case sends 58 tiles to be compressed while the other 6 requests to the hardware contained 57 tiles. With all 24 cores available, only 1 request to the hardware is needed to compress all of the tiles, with 16 of the available J2K cores receiving 17 tiles each and the remaining 8 cores receiving 16 tiles each.

It should be noted that while examining these results, the large spikes in the tiles per core count correlate to the number of DMA requests needed to process the data for the amount of available hardware. As more hardware becomes available, more tiles can be dispatched to each core resulting in less consecutive requests needed. Starting by looking at the 200 tile data set, it is easy to see that the system creates a hard-cap at 15 tiles per core by default as a first-pass optimization when implementing the algorithm. A lower boundary is needed due to the GiDEL hardware API’s having restrictions on the minimum data size that can be in a DMA transfer. After analyzing the curves in Figure 4.4 this lower bound could clearly be improved to be closer to 20 tiles minimum. This is displayed by the 200 tile test as the knee in the throughput chart occurs around 8 cores (which is 25 tiles per core) and basically plateaus from there as the tiles per core levels out.

Looking at the larger 600 tile data set shows that somewhere between 50-55 tiles per core as an upper bound provides optimal throughput with over 300 MB/s. This level of throughput is relatively
Figure 4.4: Comparison of the number of tiles dispatched to each available JPEG2000 core at various numbers of available cores

sustained until the tiles per core starts dipping below the 30 tiles mark. Similarly the 400 tile data set proves to produce optimal throughput from between 25-55 tiles per core range.

4.2 RAW Image Pre-Processing

Using the same server configuration, it is easy to display how the HCDMA yields scalability and flexibility to algorithms that cannot be distributed in the same way JPEG2000 can, such as the UDRI Image Pre Processing Suite (UDIPPS). One core of this suite achieves roughly 150 megabytes per
second of throughput so in order to boost the data rates for the UDIPPS, more cores can again be added. Each core operates completely independently with the HCDMA dispatching incoming processing requests from clients in a round-robin fashion. This technique allows for data parallelization despite each core operating completely independently.

Figure 4.5: Sample input data used when testing the impacts of the HCDMA on infrared RAW image pre-processing

The test data to be used for testing the HCDMA’s impact on the UDIPPS is the same infrared imagery previously published along with the unsharp masking algorithm [22]. This imagery is smaller than what is expected by the UDIPPS and, due to the nature of the algorithms, the imagery must all be the same uniform and expected size. In order to achieve this, the sample image repeated several times in order to produce a sample image that meets the size expected by our hardware. Figure 4.5 depicts this generated test data. We build a configuration for the HCDMA to start up with increasing amounts of available hardware and then we start up 10 client processes each with the intent of processing 200 images for each configuration.
Due to the nature of the available core binaries and the available hardware in our test environment, the test can only sweep impacts of available hardware from 1 core to 8 cores. Figure 4.6 displays the results of testing how additional hardware impacts the throughput performance of the UDIPPS. Comparing with the single-threaded software implementation, a sizable increase of roughly 271% is observed as a result of utilizing the FPGA implementation instead of the software implementation. From there it's easy to observe the impact of adding additional available hardware. It is expected that the throughput performance would only scale so far as there are more clients.
than available FPGA cores, so in this scenario we expect a plateau after 10 available FPGA cores.

Additional test results from this sweep can be found under Appendix 1.3.
CHAPTER V

FUTURE WORK

5.1 Plugin Expansion Capability

While the HCDMA originally is designed to simplify the interaction between hardware and software, it is obvious that in order to fully utilize the advantages provided by the system, easy integration of new algorithms is an important aspect. In the interest of increased flexibility it is ideal that additional algorithms be able to be built external to the main architecture and integrated via a plugin system. This way, anyone could take the integration API and expose their algorithms to the HCDMA via their own plugin. There are obvious hurdles to this approach such as exposing key data constructs for message passing and object interactions between the core HCDMA structure and any algorithm processors that a user might wish to add. Steps have already been taken down this path, with one algorithm having been successfully added to the HCDMA in this manner, however the current process is not as streamlined as would be ideal and there are still some caveats to operation that would need to be eliminated before distribution.

5.2 Service Oriented Architecture

In addition to being able to expand the system via plugins, one of the future goals of the system is for it to operate as more of a service oriented architecture. In this way, the core HCDMA could be started up with additional algorithms and resources brought online for use as needed rather than
being statically set and initialized at run time. This way the HCDMA could operate in a mode not
designed for optimal performance and throughput, but rather in a mode of operation focused on
flexibility. Ideally, it could eventually become the arbiter for all hardware accelerated interactions
needed by a system by seamlessly allocating and releasing hardware assets as the situation demands.

5.3 Distributed System

While the HCDMA in its current form stresses flexibility as much as it can, the reality of the
state of the heterogeneous computing industry today is that vendors generally do not aim to be
flexible. Some libraries only run against certain *NIX kernels while other systems might not have
an expansion chassis readily available, but rather use multiple physical servers (such as clusters
or supercomputers) for the purpose of adding additional PCIe based devices. In order to handle
such systems it would be desired to be able to distribute requests over a network. One of the
future goals of the HCDMA is to be able to field requests from remote systems by utilizing socket
based communications over a local area network in order to broaden the available resources for
environments with less strict size, weight and power restraints. Eventually this capability could be
further developed by creating a network of coordinated HCDMA instances, each with their own
available hardware assets, for the purpose of routing and scheduling request processing in a way
that would process a request in the most optimal way possible.

5.4 Code Refactoring

In its current state, the HCDMA fulfills its responsibilities quite well (as is evident by Chapter
IV), however in order to keep the design clean and concise as the system grows in capabilities it will
be necessary to refactor some aspects of the design. GidelFPGABoard (GFB) objects, for example,
probably should not all inherit from the same type but rather use multiple-inheritance to segregate
algorithm responsibilities in the interface layer into algorithm board types (GidelBoard for GiDEL
specific functions, J2KBoard for Jpeg2000 specific functions and so on). This would allow for algorithm-specific functionality to stay compartmentalized, and limits the scope of files that need to be updated when something in the algorithm interface needs to be updated. Additionally it would allow for a much simpler process when adding interfaces to the system.
BIBLIOGRAPHY


APPENDIX A

SUPPORTING INFORMATION

1.1 HCDMA Debug Configuration Example

Listing A.1: Example Debug Run-Time Configuration File

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<hcdma>
  <boards>
    <numCardsToLoad title="Number of FPGA Cards to be used for compression" type="integer">2</numCardsToLoad>
    <board0>
      <boardType list="PROC_STARIII_150_4, ProceIV_530" title="FPGA Board Type" type="list">ProceIV_530</boardType>
      <boardConfiguration list="IRPP, J2K, EOPP" title="FPGA Board Configuration" type="list">J2K</boardConfiguration>
      <targetedBus title="Debug Only: For targeting a specific card in a system, provide the integer form of the bus number here (Hint: Our apps print the hex form)" type="integer">8</targetedBus>
      <targetedSlot title="Debug Only: For targeting a specific card in a system, provide the integer form of the slot number here (Hint: Our apps print the hex form)" type="integer">0</targetedSlot>
      <targetedBanks title="Debug Only: For targeting a specific banks in card, letter representation of the banks in a comma-separated format here (Ex. ProcE’s default to A,B,C)" type="string">A,B,C</targetedBanks>
    </board0>
    <board1>
      <boardType list="PROC_STARIII_150_4, ProceIV_530" title="FPGA Board Type" type="list">ProceIV_530</boardType>
      <boardConfiguration list="IRPP, J2K, EOPP" title="FPGA Board Configuration" type="list">IR</boardConfiguration>
      <targetedBus title="Debug Only: For targeting a specific card in a system, provide the integer form of the bus number here (Hint: Our apps print the hex form)" type="integer">12</targetedBus>
      <targetedSlot title="Debug Only: For targeting a specific card in a system, provide the integer form of the slot number here (Hint: Our apps print the hex form)" type="integer">0</targetedSlot>
    </board1>
  </boards>
</hcdma>
```
<targetedBanks title="Debug Only: For targeting a specific banks in card, letter representation of the banks in a comma-separated format here (Ex. ProcE’s default to A,B,C)" type="string">C</targetedBanks>

</board1>
</boards>
<j2k>
  <dmaHandleCount title="Specify the number of DMA handles to create for the J2K process to use. More handles means more memory, but more requests can be queued up for hardware." type="integer">10</dmaHandleCount>
  <enablePipelining title="Toggles the software pipelining code to be on or off. Default is false (off)." type="boolean">false</enablePipelining>
  <dumpDMABuffers title="Debug Only: For telling the j2k board com to save the buffers it sends/receives to/from the board. Default is false." type="boolean">false</dumpDMABuffers>
</j2k>
<preprocessing>
  <histogram>
    <histogramType title="Desired histogram type to run. Default is Linear." list="Linear, AOG, Sigmoid" type="list">Linear</histogramType>
    <bottomClip title="In Linear and AOG histograms, this is used to determine the bottom % of the histogram data to clip" type="double">0.02</bottomClip>
    <topClip title="In Linear and AOG histograms, this is used to determine the top % of the histogram data to clip. Provide the percent from 1 desired (ex. if you want 96%, provide .04 as this will yield 1.00 -.04 in the code)" type="double">0.04</topClip>
    <map1var1 title="Used in Sigmoid as the percentage of the original histogram data to use" type="double">0.85</map1var1>
    <map2var1 title="Used in Sigmoid as the percentage of the resultant histogram to map to" type="double">0.70</map2var1>
    <hi_out_pc title="Used in AOG to determine the gain to be applied" type="float">64</hi_out_pc>
    <lo_out_pc title="Used in AOG to determine the gain and offset to be applied" type="float">10</lo_out_pc>
  </histogram>
  <dmaHandleCount title="Specify the number of DMA handles to create for the PreProcessing process to use. More handles means more memory, but more requests can be queued up for hardware." type="integer">10</dmaHandleCount>
  <enablePipelining title="Toggles the software pipelining code to be on or off. Default is false (off)." type="boolean">false</enablePipelining>
  <dumpDMABuffers title="Debug Only: For telling the pp board com to save the buffers it sends/receives to/from the board. Default is false." type="boolean">false</dumpDMABuffers>
  <dumpHistTables title="Debug Only: For telling the hist board com to save the tables it creates. Default is false." type="boolean">true</dumpHistTables>
</preprocessing>
<settings>
<desiredLogLvl title="Desired logging level to produce. 0 is least, 3 is as much as possible. Default is 0." type="integer">3</desiredLogLvl>

<desiredLogPath title="Desired server log path. (Note: needs a trailing slash. Ex. /home/user/logs/)" type="string">./</desiredLogPath>

<maxClientCount title="The maximum number of clients any one processing system is set up to handle at this time." type="integer">100</maxClientCount>

<temperature>
  <pollingInterval title="Desired number of milliseconds between temperature polling." type="integer">10</pollingInterval>
  <swToHwThreshold title="The temperature at which the server will resume processing data on the hardware" type="integer">75</swToHwThreshold>
  <hwToSwThreshold title="The temperature at which the server will switch processing to software to allow the hardware to cool down" type="integer">80</hwToSwThreshold>
  <unloadHwThreshold title="The temperature at which the hardware will be unloaded in order to prevent overheating damages" type="integer">83</unloadHwThreshold>
  <reloadHwThreshold title="The temperature at which the hardware is safe to be reloaded without immediate risk of overheating damage" type="integer">73</reloadHwThreshold>
</temperature>

</settings>
</hcdma>
1.2 JPEG 2000 Software Encoder Comparisons

![JPEG2000 Software Encoder Comparisons](Figure 1.1: Comparison of JPEG2000 Software Encoders for 200 Tile Sample Frame)
Figure 1.2: Comparison of JPEG2000 Software Encoders for 400 Tile Sample Frame
Figure 1.3: Comparison of JPEG2000 Software Encoders for 600 Tile Sample Frame
1.3 Addition Infrared RAW Image Pre-Processing Test Results

![Graph showing total time elapsed for each configuration of UDIPPS to process 2000 frames based on available FPGA core count.](image)

Figure 1.4: Impact of HCDMA on the Total Time Elapsed When Processing 2000 Frames in UDIPPS