FPGA-based Hyperspectral Covariance Coprocessor for Size, Weight, and Power Constrained Platforms

A Thesis Presented

by

David Alan Kusinsky

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

April 2013

This work is sponsored by the Department of the Air Force under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. Approved for public release; distribution is unlimited.

Abstract

Hyperspectral imaging (HSI) is a method of remote sensing that collects many two-dimensional images of the same physical scene. Each image corresponds to a single wavelength band in the electromagnetic spectrum. The number of bands imaged by an HSI sensor can be several hundred, and therefore a large amount of data is produced. This data must be handled by the platform on which the HSI sensor resides, either through onboard processing, or relaying elsewhere. Hence, the platform plays an important role in defining the capabilities of the entire remote sensing system.

Size, weight, and power (SWaP) are important factors in the design of any remote sensing platform. These remote sensing platforms, such as Unmanned Air Vehicles and microsatellites, are continually decreasing in size. This creates a need for remote sensing and image processing hardware that consumes less area, weight, and power, while delivering processing performance. The purpose of this research is to design and characterize an FPGA-based hardware coprocessor that parallelizes the calculation of covariance; a time-consuming step common in hyperspectral image processing. The goal is to deploy such a coprocessor on a remote sensing platform.

The coprocessor is implemented using a Xilinx ML605 evaluation board. The hardware used includes the Xilinx Virtex-6 FPGA, DDR3 memory, and PCIe interface. An implementation to accelerate the covariance calculation was created, and the OpenCPI open source framework was adopted to enable DDR3 memory and PCIe capabilities and ease coprocessor testing.

The coprocessor's performance is evaluated using several metrics: total power (Watts), processing energy (Joules), floating point operations per Watt (FLOPS/W), and floating point operations per Watt-kg (FLOPS/(W·kg)). The coprocessor is compared to a CPU-based processing platform and shown to have an overall SWaP advantage. Coprocessor FLOPS/W and FLOPS/(W·kg) performance is 2X and 2.75X that of the CPU-based platform, respectively. The coprocessor requires 45% less energy during processing.

This research shows that FPGA-based acceleration of HSI data covariance computations is promising from a size, weight, and power perspective. Significant unused FPGA resources in the coprocessor's FPGA can be used to add additional HSI data processing operations and direct HSI camera interfacing in the future.

Acknowledgements

I would first like to thank Prof. Miriam Leeser from Northeastern University for her support and guidance during my time at Northeastern, and for her suggestions after reading this thesis. I would also like to thank Shepard Siegel from Atomic Rules for his time answering my OpenCPI questions, introducing me to BSV, and the gratuitous amounts of sage-like advice.

There are several people at MIT Lincoln Laboratory that I'd like to thank for their support throughout my research: David Weitz, for reading this thesis and offering valuable advice. Rob DiPietro, Eric Truslow, Mike Pieper, and Joe Costa for answering all questions hyperspectral, and helping me focus my research. Dimitris Manolakis, for many side conversations and encouragement to "get done." Marc Burke, for many useful conversations and for providing test hardware. My officemate, Ed Leonard, for his support and sense of humour that has provided many laughs. I'd like to extend a special thanks to my Group 97 group leaders, Greg Berthiaume, Thom Opar, and Bill Blackwell, for their patience as I came back to work full-time while still completing my research.

I'd also like to thank my parents and family for their unwavering support for anything I've ever wanted to do.

Finally, I'd like to thank Meagan Lizarazo for her support and patience. Her support during this thesis helped to keep me on track, motivated, and thinking positively. She's helped me to try harder and be better.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Remote sensing can be defined as the overhead airborne observation of the earth [1]. Remote sensing platforms of this type are often constrained by size, weight, and power. This limits the amount of processing that can be accomplished by the platform, pushing these tasks to ground-based processing systems. At the same time, the movement of unprocessed data burdens the communications link between the platform and the ground systems.

One type of sensor that can be employed in remote sensing is known as a hyperspectral imaging (HSI) sensor. HSI sensor systems collect a series of two-dimensional images. Each image collected corresponds to one wavelength band in the electromagnetic spectrum [2]. The number of spectral bands imaged can be several hundred. This generates a large amount of image data that, if not compressed or processed, can exceed a communication system's maximum data rate [3]. Onboard processing of HSI data, within the SWaP constraints of the platform, can help alleviate this problem and enable real-time processing performance.

One common operation performed during the processing of HSI data is the computation of covariance. This research presents the design of an FPGA-based covariance coprocessor, suitable for SWaP-constrained HSI platforms. The coprocessor was targeted for the Xilinx ML605 evaluation board, which contains a Xilinx Virtex-6 FPGA and DDR3 memory. We discuss the architecture of the coprocessor and the details of the covariance computation, which is performed using single-precision arithmetic. We then cover the implementation of the coprocessor through the use of the Xilinx ISE design suite and OpenCPI middleware [4, 5]. Relative error analysis is presented to compare our results to a double-precision covariance computation. The performance of our coprocessor is compared to a CPU-based processing platform. This comparison includes processing throughput, power dissipation, and energy expended during processing.

This work has provided several contributions. To our knowledge, this is the first FPGA-based HSI data processing platform to perform the covariance calculation on HSI data having a large number of spectral bands; 50 in our case. Related work on FPGA-based covariance calculations has been on data having the equivalent of 6 spectral bands. Second, this is the first time that a user-generated processing algorithm has been successfully integrated at this low a level inside the OpenCPI framework on the ML605. Third, we demonstrate that FPGA-based HSI covariance calculation is an attractive option when compared to CPU-based approaches. Last, we show that our coprocessor has substantial unused resources that can be used to perform additional HSI data processing operations in future research.

1.1 Thesis Organization

Chapter 2 presents a background on hyperspectral imaging and HSI sensors, and discusses covariance and HSI data processing. Hardware typically used in HSI data processing is discussed, along with related work in the field.

Chapter 3 covers the methodology and design of the covariance coprocessor. The sample covariance algorithm implementation is discussed in detail, along with the datapath and control aspects of the design. Relevant information about the Xilinx ML605 platform is presented, and the open source middleware adopted for this research (OpenCPI) is introduced. Bluespec System Verilog (BSV), a hardware-synthesizable programming language used in this research, is also introduced.

Chapter 4 describes the hardware and software setup used in the evaluation of the covariance coprocessor. The evaluation of a CPU-based alternative platform for comparison is also presented here. A discussion of the benchmarks used for performance evaluation is included in this chapter, along with the introduction of a SWaP-conscious benchmark: FLoating-point OPerations Per Second (FLOPS) per Watt-kg (FLOPS/W·kg). Chapter 5 concludes and covers future work in this area.

Chapter 2

Background

2.1 Hyperspectral Imaging

Hyperspectral imaging (HSI) is a form of imaging where spectral information in selected regions (bands) across the electromagnetic spectrum, is collected. HSI is typically set apart from simple three-color and multispectral imagery in that the spectral bands being imaged are very closely-spaced, very high in number (a hundred or more), or both. Because processed HSI sensor data contains one spectral dimension and two spatial dimensions, this data is typically referred to as a hyperspectral "image cube".

Because of the large number of closely-spaced bands, HSI is particularly useful in remote sensing applications that will exploit this high spectral resolution and wide spectral range. Shaw and Burke define three major applications for HSI: anomaly detection, target recognition, and background characterization [2].

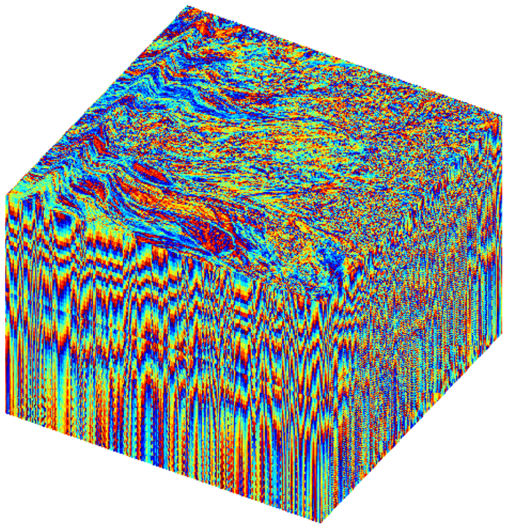

Figure 2.1: 50-Band Hyperspectral Image Cube

2.1.1 HSI Sensors

One example of an HSI platform is NASA's Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). AVIRIS is an airborne scientific instrument that images the ground in the visible, near-infrared, and short-wave infrared. AVIRIS produces image cubes having 224 spectral bands and 512 x 614 spatial pixels [6]. AVIRIS data is commonly used in remote sensing research [7, 8, 9], has a large number of bands, and much of the data collected by AVIRIS is available to the academic community. For these reasons, a 512 x 614 x N_b image cube of AVIRIS data was used in our research, where $N_b = 50$ bands. This image cube is shown in Figure 2.1.

A "whisk broom" sensor platform such as AVIRIS collects a hyperspecral image cube through a combination of scanning and platform motion. The spectral sensing is accomplished through a linear array of N spectral band-specific detectors. By

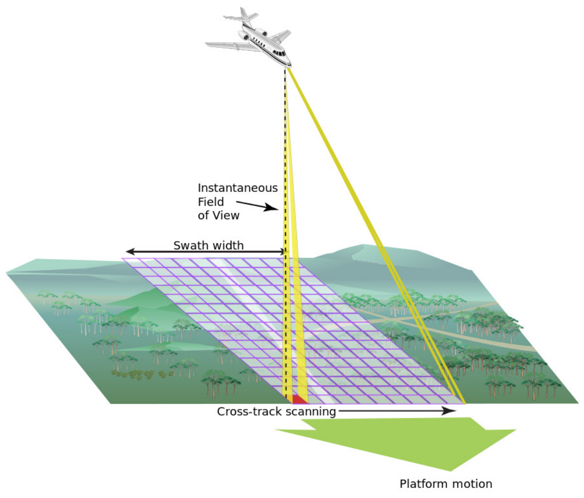

Figure 2.2: Whisk Broom HSI Platform Example

scanning the ground in the direction perpendicular to the platform's motion (cross-track direction), every spectral detector sees the entire ground swath width in this direction. This provides a single spatial line having N spectral bands. The forward motion of the platform adds the second spatial dimension, completing an image cube. This type of "whisk broom" collection method is illustrated in Figure 2.2. Other examples of HSI platforms are discussed in [1].

2.1.2 HSI Payload Design

A typical HSI payload design consists of several elements, as shown in Figure 2.3:

- A hyperspectral imager (camera) and its associated optics,

- Electronics to control the imager and capture its data,
- Electronics to provide accurate information about where the imager is pointing (GPS, inertial measurement unit (IMU), accelerometer), and
- Electronics to process the image cube (optional) and store the data, or provide it to the host platform (aircraft, satellite, etc.) for transmission.

It is advantageous to combine as many of the above elements as possible, while choosing hardware that can fit within the SWaP constraints of the system.

2.1.3 HSI Payload Size, Weight, and Power

Size, weight, and power (SWaP) is a common design trade on airborne, spaceborne, and even ground-based HSI payloads. Research has been done on band-selection techniques [10] and HSI data dimensionality reduction [3]. The former has the potential to reduce HSI sensor size and weight, and the latter can lead to reductions in data storage/downlink and processing requirements. Unfortunately, some of these techniques push additional processing requirements closer to the HSI sensor itself, wherever it may be operating.

With the shrinking size of highly-capable remote sensing platforms (UAVs, microsatellites, etc.) and the equally-shrinking power and payload weight capacity, the processing performance per Watt of power must improve to compensate, or the capabilities of the remote sensing system may be diminished.

2.1.4 Covariance

The calculation of covariance is a necessary step in many HSI data processing/target detection algorithms, such as matched filtering and anomaly detection [7]. In some applications, several covariance calculations may be performed per frame of imagery and can be a limit to performance. In this research, the calculation of the sample covariance was parallelized and performed on a Xilinx ML605 FPGA evaluation board with a Xilinx Virtex 6 FPGA. The performance of this covariance coprocessor was characterised, and the estimated improvement in SWaP was quantified by comparison with a CPU-based processing platform.

2.2 Covariance and HSI Data Processing Pipelines

Covariance is used in many HSI data processing algorithms. Because we are using pixel data from a three-dimensional data cube to compute covariance, we are calculating the *sample covariance matrix*, defined as:

$$\Sigma_{j,k} = \frac{1}{N-1}\sum_{i=1}^{N}(x_{ij} - \hat{x}_j)(x_{ik} - \hat{x}_k), \tag{2.1}$$

Here, for our HSI data, i represents the pixel number and N the number of pixels. j and k are the band numbers. $\hat{x}_j$ and $\hat{x}_k$ represent the means of bands j and k, respectively, over all image pixels.

HSI data processing algorithms are not typically implemented as a single step, but as several steps in a larger processing "pipeline". One example of a processing pipeline that involves multiple covariance matrix computations is presented in [11].

In that work, the sample covariance matrix is first computed in order to perform anomaly detection on an HSI data cube to detect all pixels that are anomalous (potential targets). These pixels are excluded from a second sample covariance matrix computation. This second covariance matrix is then used as the background covariance for the final target detection algorithm. This is known as Target Free Background Estimation (TFBE). Processing pipelines like these benefit from a covariance coprocessor with low size, weight, and power.

2.2.1 Processing Hardware

Several different kinds of hardware can be used to compute covariance, and in some cases the remaining HSI data processing steps in the pipeline. Although this research is focused on FPGA-based hardware, it is important that we understand the tradeoffs and relative strengths and limitations of other types of processing hardware. A short summary of this hardware is provided in this section.

Central Processing Units (CPUs)

CPUs are commonly used for all processing steps when processing HSI data offline on the ground, where SWaP is not as much a concern. Processing at this level requires that the sensor provide more raw, unprocessed data as well. The performance of CPU-based computing clusters have been compared to FPGAs [12]. The conclusion of this research was that clusters are better suited for cases where the data is already located on the ground, whereas FPGAs are better for obtaining a rapid response on the platform/payload.

For airborne systems, such as full-size UAVs, CPUs are used, but the scope and throughput of the processing may be reduced. For example, smaller single-board computers (SBCs) may be used instead of a standard-sized PC. These SBCs reduce SWaP, but typically do not have as powerful a CPU. Regardless, HSI processing at this level can enable a reduction in the amount of data that must be sent to the ground.

Graphics Processing Units (GPUs)

GPUs can provide a substantial improvement in processing performance. HSI data processing requires a large number of single or double-precision matrix multiplications, an operation at which GPUs excel. Recent research has shown that GPUs can provide HSI data processing performance that is equivalent to several nodes in a CPU-based computing cluster [8]. However, GPUs are typically higher in power consumption than DSPs and FPGAs, resulting in increased heat dissipation. This may be a problem for airborne and spaceborne platforms, where power and thermal management capacities are limited.

GPU manufacturers have become more focused recently in reducing power consumption, and the latest GPU families from NVIDIA have been designed to have reduced power consumption [13]. CPUs with integration GPU cores, and the potential for radiation hardened GPUs in the future will further the adoption of GPU-based processing on these platforms.

Digital Signal Processors (DSPs)

As the name implies, DSPs are commonly used in signal processing applications, such as audio/video processing, filtering, and compression, as well as fast Fourier transforms (FFTs). Many DSPs also have floating point capability, and like a microcontroller, can communicate with external peripherals. They often consume less power than CPUs, GPUs, and FPGAs. DSPs have been used to perform covariance calculations to accelerate algorithm performance [14].

DSP manufacturers are aware of the potential SWaP benefits of the DSP to the aerospace market, and are currently ahead of GPU manufacturers when it comes to enabling accelerated signal processing in space. High-reliability, radiation-tolerant DSPs for space applications are already on the market. Texas Instruments (TI), a major DSP manufacturer, recently announced a next-generation radiation-hardened DSP with 32 and 64-bit floating point support that is currently being qualified to the highest reliability standard for aerospace/military, “QML Class V.” At a recent trade show, TI mentioned that floating point operations typically performed on a higher-power FPGA can be offloaded onto their new product, reducing system power [15].

Field-Programmable Gate Arrays (FPGAs)

FPGAs can provide an improvement in processing performance, and additional benefits for SWaP-constrained platforms. FPGAs consist of a reprogrammable “fabric” that can be reconfigured as design and algorithm requirements change. One major strength of FPGAs is that they are very commonly used to interface with sensors

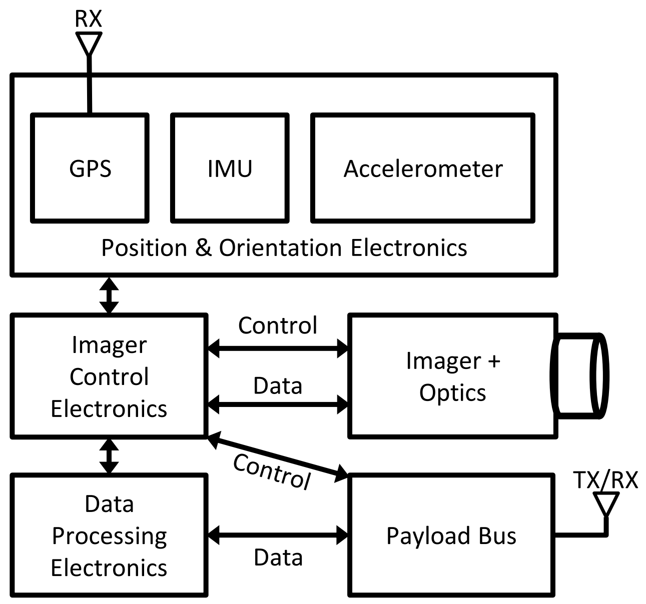
RX
GPS
IMU
Accelerometer
Position & Orientation Electronics
Imager Control Electronics
Control
Data
Imager + Optics
Control
TX/RX
Data Processing Electronics
Data
Payload Bus

Figure 2.3: Example HSI Payload

of all kinds, including HSI sensors. This places the FPGA in a unique position to both receive the HSI image cube and process it in one device. If the entire processing pipeline can be implemented in the FPGA, this removes the requirement for additional CPU/GPU/DSP-based hardware, and can offer an improvement in SWaP. FPGA manufacturers, such as Xilinx and Microsemi also have integrated DSP capabilities into their FPGA products. FPGAs have also been available as radiation-tolerant, high-reliability devices for several years.

Of these four types of processing hardware, FPGAs provide a unique advantage. They can perform a wide variety of high-performance processing, including DSP operations, as well as interface directly with high data rate camera hardware and navigation electronics.

2.3 Related Work

There is some related work in the field that discusses the computation of covariance in several types of hardware. In [14], four DSPs were chosen to perform the covariance calculations for 80 channels of data in a multi-beam echo sounding application. FPGAs were also used in their processing hardware to perform digital down-conversion (DDC) and other tasks.

Research by Martelli, et al. [16] recognized that covariance computation can be a bottleneck for some systems. They implemented covariance computation on an FPGA to compute a 6x6 covariance matrix used by a linear SVM classifier in a pedestrian detection application. Their research utilized fixed-point data representations and six features during the covariance computation. This would be the equivalent of processing six hyperspectral bands.

There has also been research into understanding the number of hyperspectral bands needed for a given detection application to achieve good detection algorithm performance. Costa [10] demonstrated that, depending on the detection algorithm and band selection technique, the number of bands used for detection could be reduced as much at 50% with only a 12% drop in the probability of detection. This research is relevant to the covariance coprocessor presented here, as the number of AVIRIS bands used (50) is a subset of the available bands.

2.4 Summary

This section provided a background on the sample covariance calculation and hyperspectal image processing. The concept of an HSI data processing pipeline was introduced, and several types of hardware suitable for the calculation of covariance were discussed. Elements of a typical hyperspectal imaging payload and the size, weight, and power tradeoff were also presented. The next chapter discusses the methodology and design of the covariance coprocessor.

Chapter 3

Methodology and Design

While the construction of a covariance coprocessor is the primary goal of this research, a holistic system design approach was taken around the coprocessor. The design allows for additional processing steps to be added in the future, and supports modifications to process HSI data from a variety of sources. This chapter describes the methodology and resulting design.

3.1 Sample Covariance Algorithm Overview

The computation of sample covariance requires many multiply-accumulate operations (MACs). These operations map well to FPGAs. Recall the sample covariance calculation in Equation (2.1). Every ith pixel intensity in spectral band j, x_{ij}, must be multiplied against its intensity in band k, x_{ik}. This occurs after the mean of bands j and k across all pixels are subtracted from these two intensity values, respectively. The resulting product is accumulated over all the pixels in the image cube.

In the covariance calculation for our AVIRIS image cube, there are two areas where parallelism can be exploited. First, some or all of the band-to-band inten-

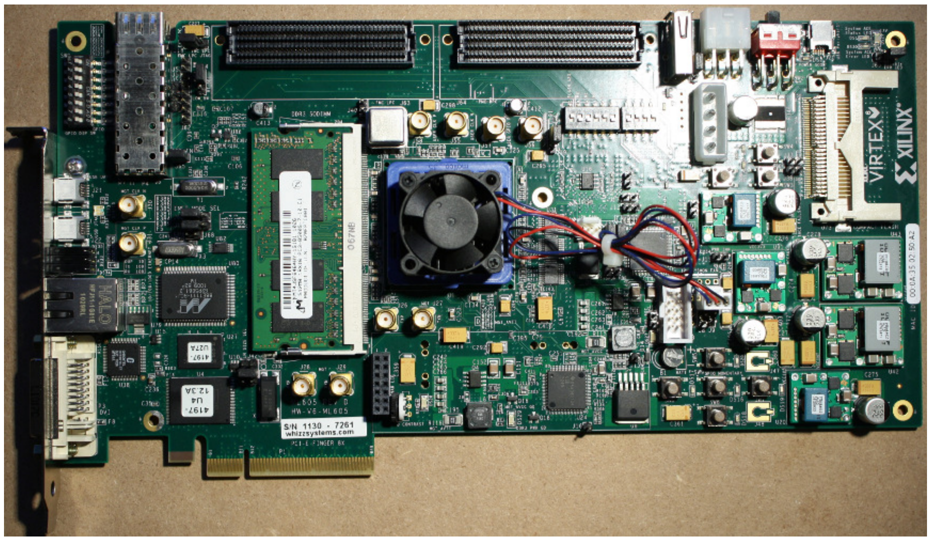

Figure 3.1: Xilinx ML605 Evaluation Board

sity products needed for a single pixel can be computed in parallel. Second, these band-to-band products can be computed for several pixels in parallel. Both types of parallelism were exploited in our research, and are covered later in this section.

3.2 Hardware Platform

The hardware platform used in this research is the Xilinx ML605 evaluation board shown in Figure 3.1, which contains a Xilinx Virtex-6 LX240T FPGA. This FPGA features 768 DSP48E1 slices, which are used by the adders and multipliers in our implementation, as well as 14,976 Kb of internal block RAM used for data storage.

Additional hardware available on the ML605 used in this research is 32 MB of BPI flash memory, 512 MB of DDR3 memory, and a PCI Express connector located on the board's edge. Also of interest for future work is the ML605's support for two

FPGA Mezzanine Cards (FMCs), which can be used for hardware interfacing.

The ML605 is well-suited for this research because of its highly capable FPGA, onboard DDR3 memory for image storage, and PCI Express capability for image cube loading and readback of processing results.

3.3 Algorithm Implementation

3.3.1 Data Interface and Middleware

Understanding the movement of data into and out of the covariance algorithm implementation is as important as the implementation itself. If the implementation is data-starved, then additional parallelization is not useful. PCI Express is used as the means to provide data to the coprocessor from a host PC. To satisfy the need for PCI Express data transfers, as well as control of the DDR3 memory on the ML605, the open source framework OpenCPI is utilized. OpenCPI [4, 5], is a middleware solution that is geared towards heterogeneous processing applications. OpenCPI enables communications with a host PC over PCI express, as well as reading and writing to the DDR3 memory on the ML605. The covariance implementation in this research interfaces with and executes within OpenCPI. Figure 3.2 shows a block diagram of the high-level design and relevant interfaces. The OCPI middleware, OCPI worker, and OCPI control application were modified as needed to communicate with our implementation.

The OCPI control application on the host PC is used for several tasks: writing the

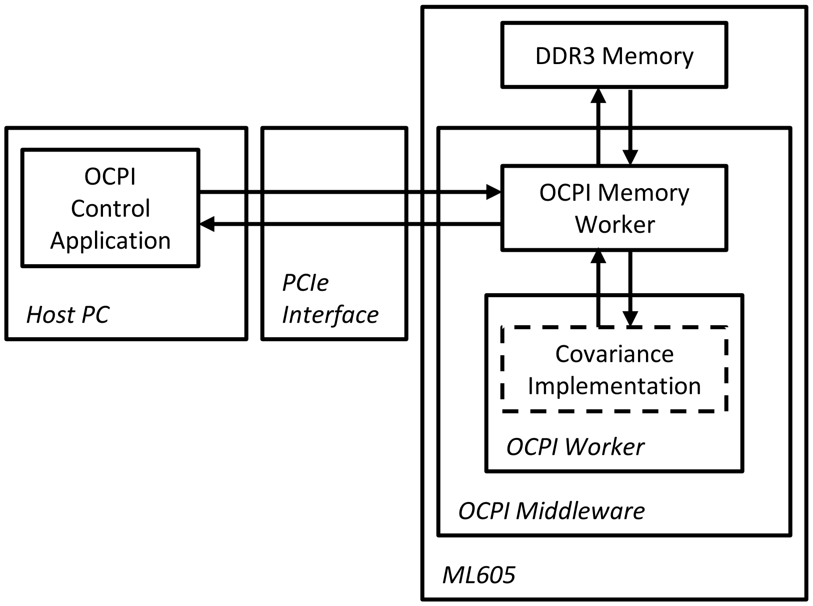

Figure 3.2: Design High Level Block Diagram

image cube to the DDR3 memory on the ML605, starting/stopping the coprocessor, monitoring status, and retrieving the covariance result. The host PC and control application were used as a control and diagnostics tool, and therefore are not actively needed by the coprocessor during processing.

It is important to note that the image cube transferred to the ML605 memory from the host PC is already mean-subtracted. Section 3.4.4 shows that significant FPGA resources are available after implementation of our coprocessor. These resources can be used to implement the subtraction of means in the future.

3.3.2 Datapath Interfaces

Although most HSI sensors output image data as either straight or two's-complement binary, single-precision floating point is used in our research to represent each of the

50 band values for each pixel. This is to provide an improved dynamic range during the covariance computation versus a fixed-point implementation. This means that $32 * 50 = 1600$ bits are needed to represent each pixel in our datapath.

OpenCPI provides a 128-bit wide data interface to the DDR3 memory that can provide data every clock cycle, provided that several read requests are always in flight. This information provides a lower bound of 13 clock cycles to transfer a pixel between the DDR3 memory and the covariance implementation that is used to guide the design of the final architecture.

Therefore, the coprocessor throughput is FPGA resource limited until it exceeds a pixel processing throughput of $\frac{\text{one pixel}}{\text{13 clock cycles}}$. This defines the interface and its capabilities.

Datapath Implementation

A block diagram of the high-level datapath, interfaces, and initial algorithm implementation is provided in Figure 3.3. From the figure, the data and computational requirements for the implementation can be seen. For this datapath:

- N pixels of single-precision HSI data are loaded from memory, each having 50 bands ($1600N$ bits). N represents the number of pixels that will be processed independently in parallel.
- 50 MACs are performed per clock cycle for each parallel pixel to compute the inter-band products. The 50 multiplier results for each parallel pixel are

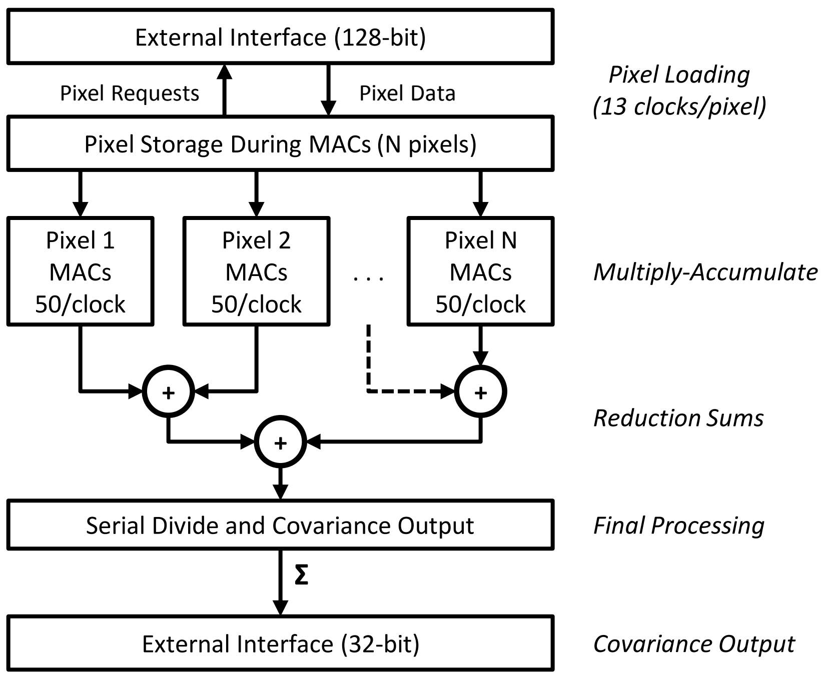

Figure 3.3: Datapath Block Diagram

accumulated into the appropriate row of a 50 x 50 accumulator matrix. Each parallel pixel has its own accumulator matrix.

- The next N pixels of HSI data must be pre-loaded during these 50 clock cycles in order to avoid stalling the pipeline.

- Once all pixels are processed through the MACs, if $N > 1$, reduction sums are performed serially to sum the N accumulator matrices needed to obtain the final sample covariance matrix.

This algorithm implementation takes 50 clock cycles per pixel to complete, and therefore its data throughput requirements can be satisfied by OpenCPI's DDR3 interface. Additionally, it is possible to process up to $\lfloor 50/13 \rfloor = 3$ pixels in parallel without stalling the implementation's pipeline. This implementation was therefore selected for our design, and was coded in VHDL using the Xilinx ISE 14.1 software suite. Initially, two pixels were chosen to be loaded and processed in parallel ($N = 2$).

Figure 3.4 shows a detailed diagram of the datapath of the design. N input pixels are passed through a FIFO of depth two constructed from FPGA block memory. The output of the FIFO is used by parallel multipliers, 50 per pixel, during which the next N pixels are enqueued into the FIFO. This buffering keeps the OCPI DDR3 memory interface occupied with read requests, which maximizes its performance.

The multipliers consist of single-precision Xilinx multiplier IP cores. These multipliers are provided new data every clock cycle to compute products to sum into the accumulator matrices for each parallel pixel. This requires $50N$ multipliers. The first clock cycle provides data for the $b_{1,1}, b_{1,2}, \cdots, b_{50,50}$ band-band products for each pixel. The second clock cycle provides the $b_{2,1}, b_{2,2}, \cdots, b_{2,50}$ products. Xilinx provides several options to control the speed and size of the multipliers in the FPGA fabric. To minimize the size of the multipliers in FPGA lookup tables (LUTs) and flip-flops (FFs), and maximize the utilization of DSP48E1 blocks, the maximum settings for latency (6) and DSP48E1 blocks (3) per multiplier are used.

FPGA block RAMs are used to implement storage of the accumulator matrices, and Xilinx single-precision adder cores are used for the additions. As with the parallel multipliers, $50N$ adders are used. The block RAMs are configured as simple dual-port RAMs. Dual-port RAMs are necessary because of the latency of the single-precision

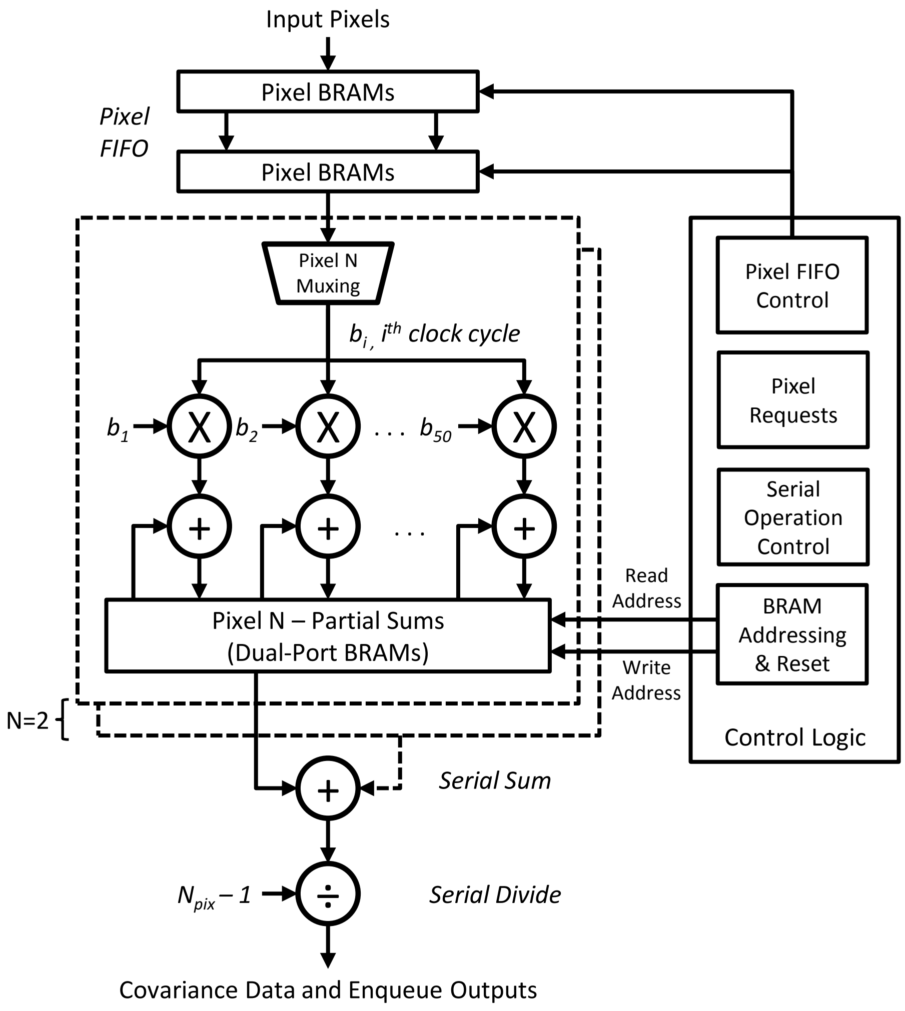

Figure 3.4: Detailed Datapath of Covariance Implementation

adders. The read address of the block RAMs are set to follow the current row of the band-to-band products being output by the multipliers. The write address is set to follow the row corresponding to the sum being output by the adders. Therefore, the write address lags behind the read address by the adder latency, and no read-write collisions occur. To minimize the LUT and FF size of the adders, the maximum settings for latency (11) and DSP48E1 blocks (2) are used.

Once all pixels in the image have been processed through the datapath, the N accumulator matrices must be added together and divided by $N_{pix} - 1$ to obtain the final covariance matrix. Because the number of pixels is large ($N_{pix} = 512 * 614 = 314,368$), a serial sum of $N_b^2 = 2500$ accumulator matrix values takes a small amount of time compared to the MACs. Therefore, this sum is performed serially and only requires a single adder.

The results of this serial sum are provided directly to a divider that divides by $N_{pix} - 1$ to produce one entry of the 50x50 covariance matrix each clock. This data is provided as an output of our datapath, along with an enqueue signal that is used by the OCPI worker shown in Figure 3.2 to schedule writes to the ML605 DDR3 memory. Once the sample covariance is fully enqueued to the OCPI worker, a request for a new image is made, and the process repeats.

Table 3.1 lists the ISE-estimated resource utilizations of the multiplier, adder, and divider cores used in the design, along with their latency and predicted maximum frequencies (f_{max}). In Section 3.4.4, we present the total resource utilization of our

Table 3.1: IP Core Summary From ISE

Single-precision Core	# LUTs	# FFs	# DSP48Es	Latency	f_{max} (MHz)
Multiplier	107	114	3	6	429
Adder	287	337	2	11	380
Divider	1,071	1,366	0	28	433

design.

Control Implementation

Internal control logic is necessary to ensure the proper operation of the design. The logic cores used in the design for multiplications, additions, and divisions, are equipped with “data ready” signals that indicate that the result is ready. These are connected to the “new data” input signals of the next mathematical logic core, along with the associated data whenever possible. An example of this type of control logic connection is between the output of the serial adder to the serial divider at the end of the sample covariance calculation.

This type of simple connection is not possible for every aspect of the design, and so additional control logic is necessary. VHDL code was written to control the request for a new image, the input pixel FIFOs, the data ready signal to the multipliers, and read and write addresses that are used by the partial covariance block RAMs. Because the partial covariance data held in the block RAMs will persist from image-to-image, a block RAM output reset signal was used in the control logic to ensure that the first input to the adders from the block RAMs is zero on each new image.

Code was also written to control the serial additions and divisions that occurs at the end of the sample covariance calculation, and the enqueue signals that are provided to the next-higher level of the coprocessor design.

3.4 Integration with OpenCPI

As mentioned in Section 3.3.1, OpenCPI (OCPI) was adopted as a middleware solution for the ML605 that allows the design to utilize the PCI Express interface, as well as provides a DDR3 interface that can meet the design data throughput requirements [4, 5]. OCPI also offers powerful control features that can be used to control the coprocessor from the host PC, which greatly improved the testability of the coprocessor. These control features will be covered in Chapter 4.

With the standalone covariance VHDL code completed and tested, it was then integrated with OCPI. Figure 3.5 shows a simplified block diagram of the OCPI hierarchy, as it exists on the ML605. The ML605 represents the overall *platform*. Within that platform, OCPI creates several *container* and *infrastructure* constructs. One of the containers holds the user's application(s), and another adapts these application(s) to the ML605 hardware platform, with the help of the infrastructure. For example, OCPI user applications can utilize the ML605 external PCIe interface through an abstracted interface at the application container level. Access to the ML605 DDR3 memory is provided by a separate DDR3 *device worker*, that can be included if needed.

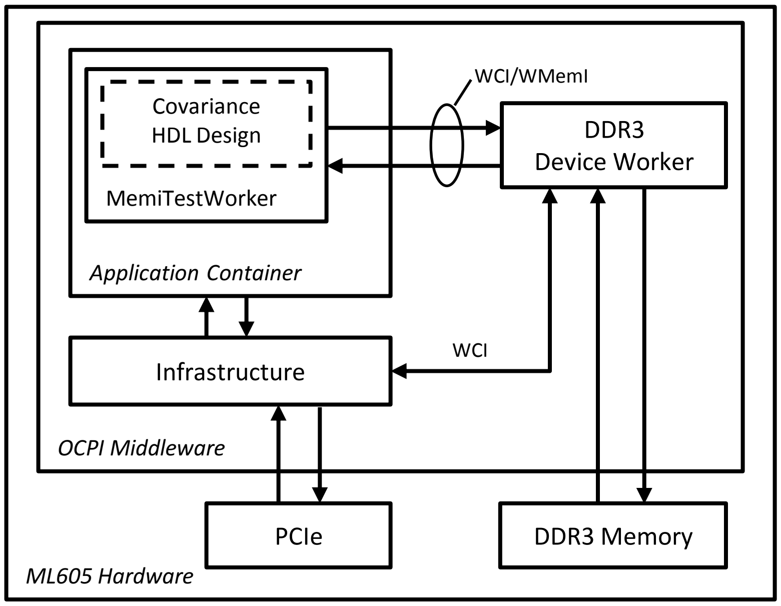

Figure 3.5: OCPI Hierarchy on ML605

At the deepest level within OCPI are *workers*, which perform tasks and obey standard OCPI data and control interfaces. This represents where our covariance design is implemented. The standard interfaces used in OCPI are known as the Worker Control Interface (WCI), Worker Streaming Interface (WSI), Worker Message Interface (WMI), Worker Memory Interface (WMemI), and Worker Time Interface (WTI). For our application, the WCI and WMemI interfaces are the most important, as they allow control of our coprocessor and status of its operation (WCI), and movement of data to and from the DDR3 memory (WMemI).

With the need for an OCPI worker identified, two major options are considered:

1. Turn the existing VHDL design code into a standalone OCPI worker, compatible with OCPI control and data interfaces.

2. Instantiate the design in an existing worker that already obeys these interfaces.

To maintain simplicity, the second option was chosen, and an existing OCPI worker was identified that enabled a rapid solution. The "WMemiTestWorker [1]" was identified as the worker to interface with our design. This worker was designed to test the DDR3 memory over the WMemI interface, and therefore already demonstrated proper worker-based operation of the WMemI interface. This worker was selected for modification to integrate with the covariance VHDL design.

3.4.1 OCPI Worker Modification

Much of the OCPI source code is written in Bluespec System Verilog (BSV), a high-level language than can be compiled into efficient and synthesizable Verilog code [17]. BSV uses "interface" constructs to represent the boundaries of modules, and "methods" to drive data across these interfaces, as shown in Figure 3.6. These methods have "ready" and "enable" signals that control when they are allowed to execute. A simple Verilog "wrapper" is used to adapt our VHDL design to the BSV OCPI worker that requires these interface constructs. We call this wrapper "CovIF."

To allow CovIF to exchange pixel and covariance data with our VHDL design, it must extend an existing BSV interface, or create one of its own. Because our data transfers behave much like FIFOs, an existing BSV FIFO interface was extended. The wrapper acts as a virtual FIFO, asserting empty when new pixel data is needed, and full otherwise. A separate covariance FIFO added to MemiTestWorker is filled

[1]From OCPI repository: https://github.com/ShepardSiegel/ocpi/commit/3b64696

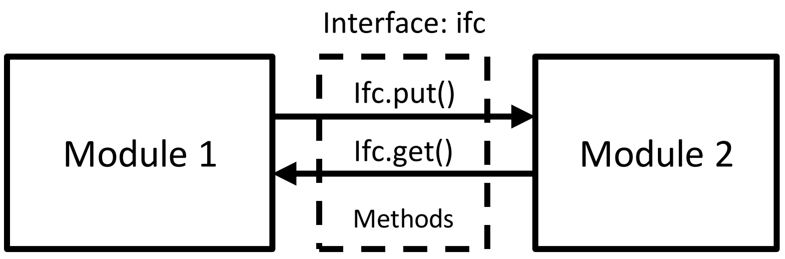

Figure 3.6: BSV Methods Moving Data Between Modules via Methods

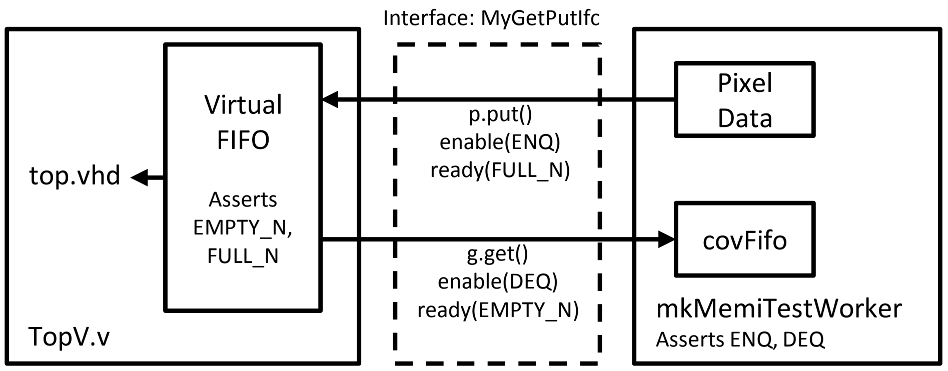

Figure 3.7: Adapting VHDL Design to BSV Using a Virtual FIFO

by our VHDL covariance enqueue signal (passed through the wrapper), and drained by the OCPI worker as the values are written to DDR3 memory. Figure 3.7 shows the arrangement between the BSV OCPI worker, the Verilog wrapper, and our VHDL design.

BSV's implementation of *atomic operations* and *rules* simplifies the design of this wrapper. The BSV FIFO interface methods, *put* and *get* each have a *ready* signal, and these signals are connected directly to the virtual FIFO's FULL_N and EMPTY_N signals, respectively. This means that the put and get methods cannot execute unless they are declared ready by the virtual FIFO.

Because the BSV code is rule-based, if any code within a rule has a method that cannot execute, then no other code in that rule will execute. Figure 3.8 shows a code sample from MemiTestWorker that illustrates this concept. This eliminates the need to write complicated control logic for data flow to/from the OCPI MemiTestWorker and the Verilog virtual FIFO. A preliminary version of the virtual FIFO with its adopted BSV FIFO interface was compiled and simulated prior to integration with the WMemiTestWorker.

WMemiTestWorker's BSV code was further modified to read the proper number of 128-bit data words from the DDR3 memory and write the covariance values to DDR3 at the end of each execution. Dataflow over WMemI is controlled using WMemI *read requests, read responses, and write requests.* Read requests are sent with the desired memory address, and the read response (when it arrives) contains the 128-bits of data. Write requests are issued with the address and data all at once. WMemI read and write requests originate from the modified MemiTestWorker, since this worker contains the WMemI interface instance as well as the interface instance for the covariance module.

OCPI does not currently "guard" the WMemI read response channel, which means that the data attached to a read response must be stored immediately, or it will be lost. As mentioned in Section 3.3.2, it is desirable to maximize the number of DDR3 read requests in flight, but our covariance module is not always ready to receive and store all read response data the moment it is received. Therefore, guard-

```
interface Put p;
  //method put() asserts ENQ and DIN
  //to virtual FIFO only when FULL_N is
  //asserted by the virtual FIFO and the
  //put() method is called by the BSV module
  method put(DIN) enable(ENQ) ready(FULL_N);
endinterface

//explicit rules: wci.isOperating && isTesting
//must be true for rule to fire
rule write_alg (wci.isOperating && isTesting);

  //get data from DDR3 read response FIFO
  //implicit rule for first(): rdRespFifo not empty
  let aData = rdRespFifo.first;

  //send to virtual FIFO (covariance VHDL)
  //uses method put() from interface Put
  //implicit rule for put(): virtual FIFO not full
  cov.streamP.put(aData);

  //dequeue data from rdRespFifo
  //implicit rule for deq(): rdRespFifo not empty
  rdRespFifo.deq;

  //increment read responses dequeue cnt
  wmemiRdRespDeq <= wmemiRdRespDeq + 1;
endrule
```

Figure 3.8: BSV Rule and Method Control

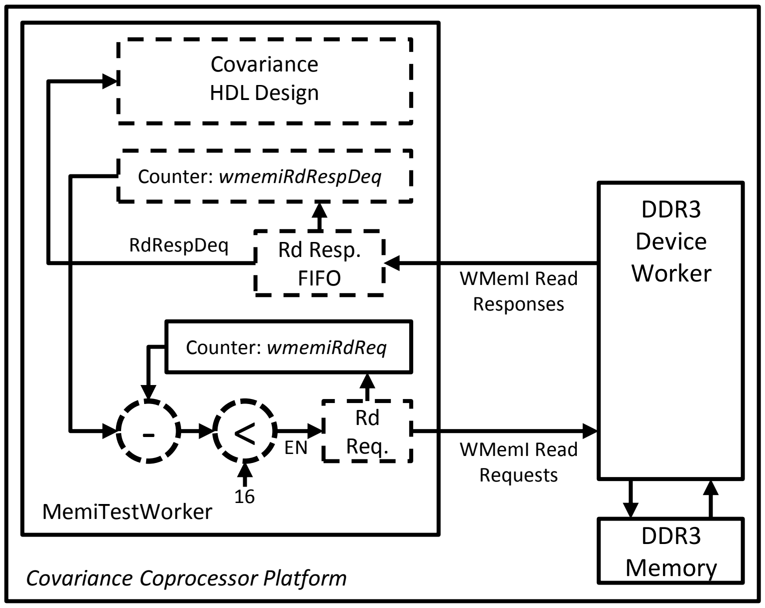

Figure 3.9: Metric for Guarded Read Responses

ing was added for the WMemI read response channel. This ensures that a sufficient number of DDR3 read requests can be in flight, while ensuring that data contained in the read responses is not lost if the covariance module is not ready for the data.

This guarding was accomplished using a FIFO to store WMemI read response data until it is needed. A read response FIFO depth of 16 was selected to match a corresponding OCPI WMemI data FIFO. This change necessitated a need for a system that ensures that no more than 16 requests are in flight at a given time. This system uses a simple metric, (read requests made - read responses dequeued from FIFO) to decide when to send a new read request. The counters used in this metric are reset at the end of every processed image. The system is illustrated in Figure 3.9.

Final modifications were made to the OCPI worker to allow for control during testing. WCI control registers were modified to allow setting the number of pixels to be read from memory, the maximum number of read requests in flight, and the number of times to loop the calculation. Continuous execution was added to assist with real-time power measurements.

3.4.2 OCPI Software Modification

In addition to the OCPI BSV modifications, software modifications were made to an OCPI C code utility called "swctl." The utility's capability to write to ML605 DDR3 memory was enhanced to load an input pixel data file and write it to the DDR3 memory.

3.4.3 OCPI Build Script/Project Modification

The building of an OCPI-based design is accomplished through the use of the GNU *make* tool and UNIX shell scripts. These scripts were modified to add additional functionality and ease the building of the covariance coprocessor design. One script, *build_fpgaTop* was modified to include the Xilinx IP cores used in the covariance design. To aid with FPGA design timing closure, this script was also modified to optionally use the Xilinx SmartXplorer tool to run multiple MAP and PAR strategies during a build. Xilinx project files used by OCPI were modified to include the IP cores added to the design.

Table 3.2: Coprocessor FPGA Resource Utilization

FPGA Resource Type	Available	Covariance + Virtual FIFO	Full Coprocessor
Slice Registers	301,440	51,082 (16%)	83,559 (27%)
Slice LUTs	150,720	38,940 (25%)	80,829 (53%)
RAMB36E1s	416	0 (0%)	39 (9%)
RAMB18E1s	832	308 (37%)	311 (37%)
DSP48E1s	768	502 (65%)	502 (65%)

3.4.4 FPGA Resource Utilization

The FPGA resource utilization of the covariance coprocessor is provided by the Xilinx design tools. Resource utilizations for two phases of the design are provided here. Table 3.2 provides the major resources required for the covariance implementation with Verilog virtual FIFO, and for the fully-functional covariance coprocessor design after integration with OCPI. It can be seen that the final design utilizes less than 55% of the total resources in each category, with the exception of the DSP48E1s. This shows that the covariance calculation design is fully realizable in the ML605 platform with the OCPI middleware from an FPGA resource standpoint.

3.4.5 FPGA Timing Analysis

An FPGA timing analysis is provided by the Xilinx design tools during the build process. There are several clock domains in the full coprocessor design, and each must achieve a minimum specified frequency. Of most importance to our design is the 125MHz clock domain. This is the clock domain in which the covariance calculation and OCPI MemiTestWorker operate. This clock domain, as well as all others in the coprocessor design met their timing requirements. This shows that the current parallelized covariance calculation design is fully realizable in the ML605 platform with the OCPI middleware from an FPGA timing standpoint.

3.5 Summary

This chapter described the methodology and design of the covariance processor. Figure 3.10 shows the final covariance coprocessor design. This includes the ML605, OCPI middleware, Verilog wrapper, and the VHDL sample covariance implementation. Details of the data throughput, FPGA resource utilization, and maximum operating frequency were provided. The OCPI middleware and BSV source code were introduced, and the modifications made to both were discussed. The next chapter covers the experimental setup and testing of the coprocessor.

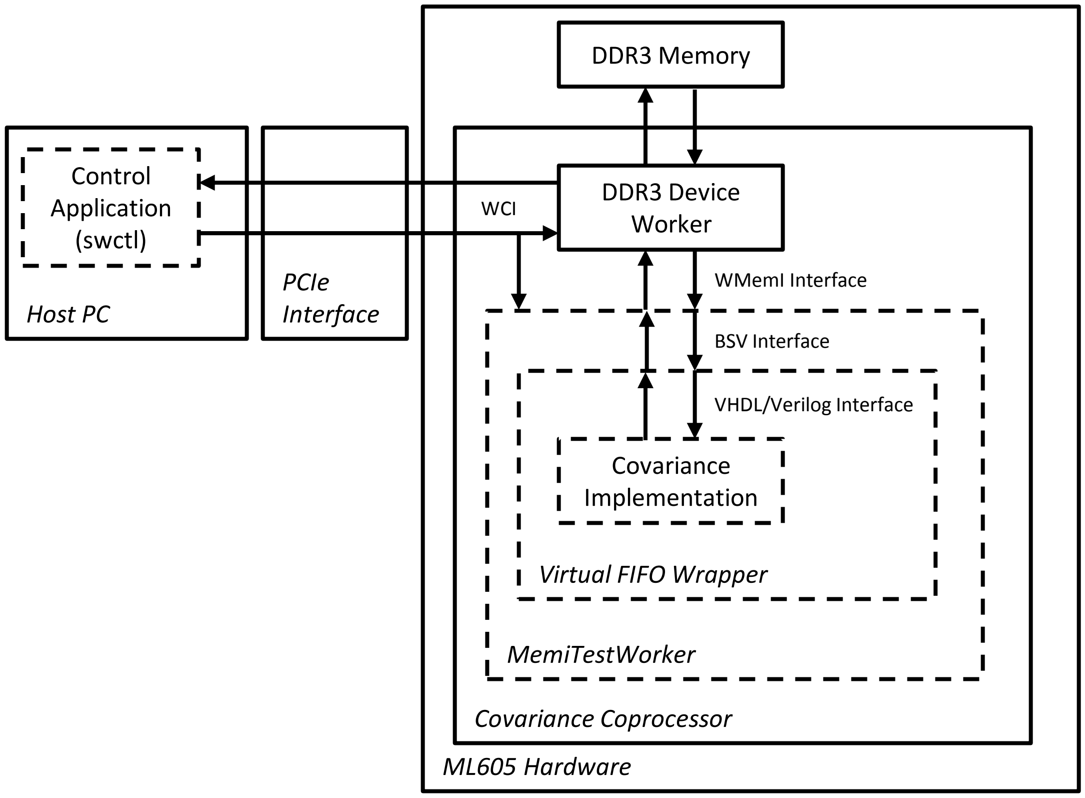

Figure 3.10: Final Covariance Coprocessor Design

Chapter 4

Experimental Setup and Results

This chapter discusses the hardware, software, diagnostic tools, and other resources used in the testing of the covariance coprocessor. The results of this testing are also presented. This includes HDL simulation (validation) as well as MATLAB and C validation of the sample covariance calculation. Runtime and power measurements of the covariance coprocessor and comparison platform are also included.

4.1 Hardware Platforms

Two separate hardware platforms are used in this research. One serves as the host PC for all of the ML605 covariance coprocessor experiments, and the second is used as a reference platform for comparison. The coprocessor host PC is used as a control and diagnostics tool, and therefore its CPU, memory, and storage are not actively needed by the coprocessor during processing.

The reference PC was chosen specifically to model processing hardware that is typical of SWaP-constrained payloads. It is a single-board computer (SBC) with limited external interfaces and small physical footprint. Table 4.1 lists the hardware

Table 4.1: Hardware Platforms for Testing

	Host PC	Reference PC
Model	Custom	Congatec conga-BM67
Motherboard/ Carrier	Asus Sabertooth P67	ACTIS Computer KCAC-0320
CPU	Intel Core i7 2600K, 3.4 GHz	Intel Core i7 2710QE (SV), 2.1 GHz
Memory	16GB DDR3	512MB DDR3
Disk Drive	128GB HDD	12GB USB
OS	Red Hat Linux	CentOS Linux
Coprocessor	ML605	None

details of the two platforms.

4.2 Diagnostic Tools and Software

Diagnostic tools and software were used for validation and measurement. Validation of hardware includes ensuring that the FPGA is performing the proper operations at the proper time, and that the operations themselves are producing the expected results. This is accomplished through HDL simulations, as well as real-time measurements performed by diagnostic tools, such as the ChipScope Pro package from Xilinx. This tool provides an Integrated Logic Analyzer (ILA) that runs within the FPGA and can be configured to capture any signal of interest. Our modified OCPI "swctl" tool is also used on the Host PC to collect information from the coprocessor via the WCI interface that is used to assess proper performance.

Software is used to evaluate coprocessor performance. Both MATLAB and C code were used to evaluate the performance of just the sample covariance calculation. C

Table 4.2: Software Used in Testing

	Host PC	Reference PC
Sample Covariance Validation	MATLAB, C	
HDL Validation	ModelSim	
Coprocessor Validation	OCPI, ChipScope, MATLAB	
Sample Covariance Performance		C
Coprocessor Performance	OCPI, ChipScope	

code was used on the reference platform for this calculation to provide a point for comparison with our full covariance processor. Both OCPI and ChipScope Pro were used to measure performance metrics of the final coprocessor design in real-time during its operation. Table 4.2 lists the software tasks, and the software tool and platform on which they operated.

4.3 Design Validation

This section will focus on the validation of the sample covariance calculation. The scope of the software validation is limited to assessing the relative error between the hardware implementation and a MATLAB implementation.

4.3.1 Hardware Validation in Software

A "design for test" approach was taken during the design of the hardware implementation, keeping in mind that the VHDL code will later be validated using a testbench. Prior to integration with OCPI, the design was fully tested with a testbench written in VHDL to validate performance and accurate computation. Figure 4.1 gives a di-

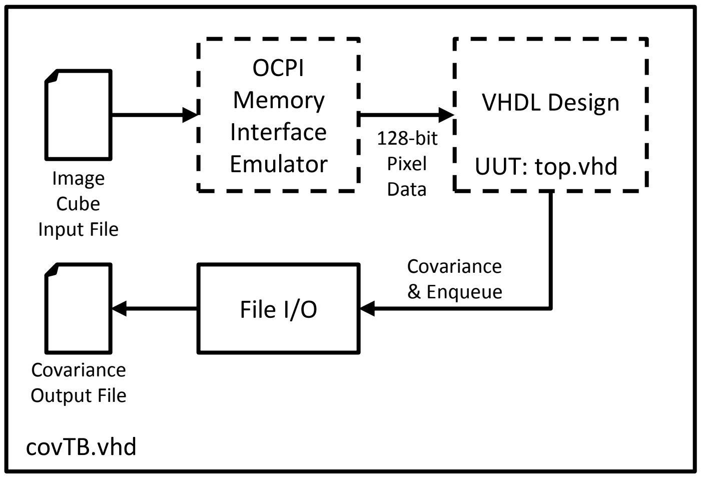

Figure 4.1: Hardware Validation Testbench Diagram

agram of the testbench setup. This testbench instantiates the design, provides HSI data to the design, and accepts the output sample covariance matrix.

MATLAB was used to generate both a full 512x614x50-band hyperspectral test image from actual AVIRIS data, as well as a smaller 10-pixel dataset that was used for early rapid testbenching. These were stored in a text file that was read by the testbench, and provided to our design in the same 128-bit wide format as the DDR3 memory interface. Output covariance results from the design were accepted by the testbench, and stored into a data file for later validation.

ModelSim was used to simulate the testbench for both the 10-pixel and full-image cases, with the latter taking over an hour to simulate. The 10-pixel case was used for testing, and was extremely useful in catching code bugs and control logic issues. The full-image case was used to validate the final design once, before integration with OCPI for the higher-level coprocessor design. A snapshot of a 10-pixel sim-

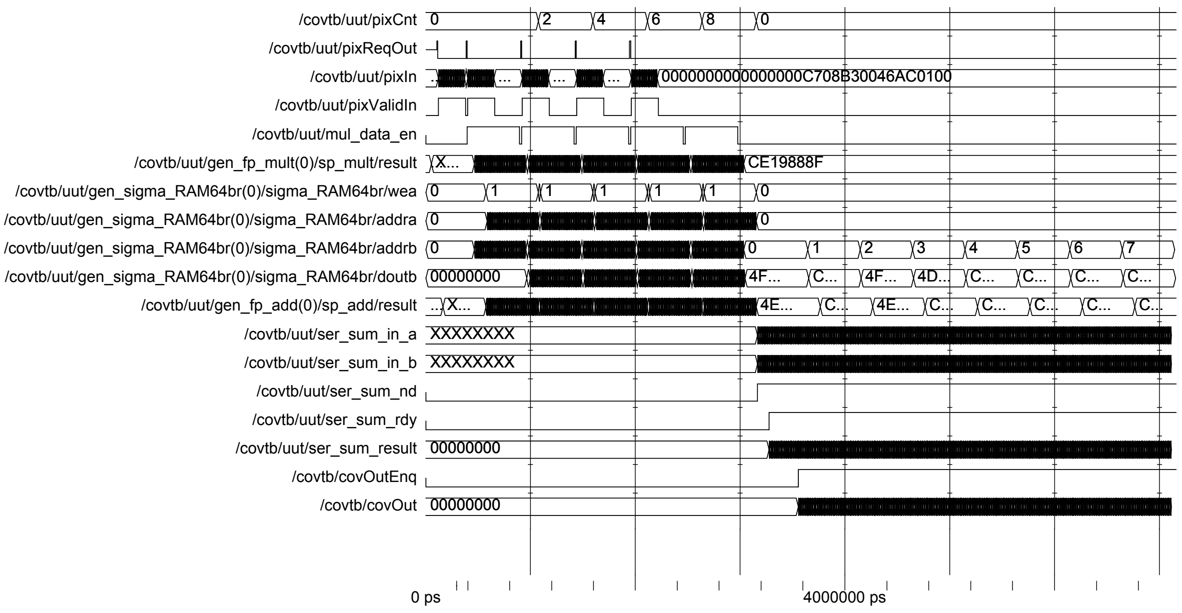

Figure 4.2: Ten Pixel Hardware Simulation

ulation is shown in Figure 4.2. This figure at a high level shows that: two pixels are prefetched on the data interface (pixIn) at the beginning of execution, the block RAM (sigma_RAM64br) read and write addresses (addrb and addra, respectively) change in a staggered manner, avoiding collisions in our accumulator, and that a steady stream of results from the adder (sp_add) is maintained. The start of the serial operations (sum and divide) can also be seen in this figure, as well as the final covariance output and enqueue signal (covOut, covOutEnq).

MATLAB scripts were written to load the covariance matrix output of the testbench into a MATLAB variable that is used to validate the results. Because our datapath uses single-precision values, verification of our implementation in MATLAB required some additional work. Associativity is not guaranteed in floating point arithmetic: sometimes $(x+y)+z \neq x+(y+z)$. Because of the parallel pixel processing, our implementation performs additions in a different order than MATLAB's built-in covariance function *cov()*. In single-precision, this is enough to prevent a simple one-to-one comparison of the covariance matrices to validate correctness.

To deal with this, a custom MATLAB script was written to perform a covariance calculation in the same exact manner as the FPGA hardware. The pseudocode for this script is provided in Algorithm 1. This algorithm emulates the parallel nature of the HW computation, and also ensures that the scalar division occurs at the end of the sum of products, as is done in hardware. It also attempts to maintain the same ordering of the additions in the dot products as the hardware.

```
Data: input image cube
Result: covariance matrix, $\Sigma$
for i over all bands do
    for j over all bands do
        Construct vector of odd-numbered pixels for single band i;
        Construct vector of odd-numbered pixels for single band j;
        Perform dot product and store;
        Construct vector of even-numbered pixels for single band i;
        Construct vector of even-numbered pixels for single band j;
        Perform dot product and store;
        Add dot products to obtain $\Sigma_{ij}$
    end
end
Divide matrix by scalar $N_{bands} - 1$;
```

Algorithm 1: Initial MATLAB HW Validation Algorithm

For the 10 pixel test image, this MATLAB algorithm provided a covariance matrix that exactly matched that of the HDL simulation. When executed to validate the full 512x614x50 image cube, it did not provide the same results. After investigation, it was determined that MATLAB performs optimizations in the dot product calculations once the input vectors are of larger sizes. This was altering the order of the additions in the dot products and thus changing the validation results. Because validation of a full image against the HDL simulation is crucial to verification of the design, a different MATLAB algorithm was constructed. Algorithm 2 gives the final MATLAB validation algorithm.

```
Data: input image cube
Result: covariance matrix, $\Sigma$
for i over all bands do
    for j over all bands do
        Set A, B, tA, tB to zero;
        Construct vector of odd-numbered pixels for single band i;
        Construct vector of odd-numbered pixels for single band j;
        A = elementwise product of vectors;
        Construct vector of even-numbered pixels for single band i;
        Construct vector of even-numbered pixels for single band j;
        B = elementwise product of vectors;
        for k over length of A,B do
            Accumulate $k^{th}$ value of A into tA;
            Accumulate $k^{th}$ value of B into tB;
        end
        $\Sigma_{ij} = tA + tB$;
    end
end
Divide matrix by scalar $N_{bands} - 1$
```

Algorithm 2: Final MATLAB HW Validation Algorithm

This version, although extremely inefficient for MATLAB to execute, did provide an exact match to the HDL simulation results for both the 10 pixel and full image cube cases. This confirmed the expected operation of the hardware implementation over a full-sized image.

A C version of the validation algorithm was structured similarly to the hardware architecture, and also provided an exact match for a full-sized image. The algorithm is provided below in Algorithm 3

```
Data: input image cube
Result: covariance matrix, $\Sigma$
for pixel $x_k$, $k = 0, 2, 4, \ldots, N_{pix}$ do
    Construct two pixel vectors;
    $\overrightarrow{\mathbf{a}} \leftarrow x_k$;
    $\overrightarrow{\mathbf{b}} \leftarrow x_{k+1}$;
    for i over all bands do
        for j over all bands do
            $\alpha_{ij} = a_i a_j$;
            $\beta_{ij} = b_i b_j$;
        end
    end
end
for $i \leftarrow 0, 1, \ldots, N_{bands} - 1$ do
    for $j \leftarrow 0, 1, \ldots, N_{bands} - 1$ do
        $\Sigma_{ij} = \alpha_{ij} + \beta_{ij}$;
        $\Sigma_{ij} = \Sigma_{ij} / N_{bands} - 1$;
    end
end
```

Algorithm 3: C HW Validation Algorithm

Relative Error Analysis

Single precision floating point has been used successfully in HSI data processing, and has yielded acceptable performance [9]. However, in addition to validating the proper implementation of the hardware architecture, it is also of interest to compare the single-precision results of this implementation to results with double-precision. This can be done by computing the *relative error* of the covariance computed in single-precision versus double-precision. Relative error in this case is defined as:

$$\eta = \frac{|x_d - x_s|}{|x_d|}, \tag{4.1}$$

where x_d is the double-precision result, and x_s the single-precision result.

Equation (4.1) is used to compute relative error of the covariance matrix of the full 512x614x50 AVIRIS image cube. For this calculation, the single-precision covariance matrix obtained from the HDL simulation was used. The double-precision covariance matrix was calculated using MATLAB's cov() function. Figure 4.3 provides a histogram of the relative error. All but two of the 2500 covariance values have less than 0.2% relative error versus a double-precision implementation. The maximum relative error was approximately 1.2%.

4.3.2 Validation in Hardware

In Section 4.3.1, software tools were used to validate the hardware design prior to integration with OCPI, and it was determined that the relative error of our implementation is small when compared with a double-precision implementation. Because of the complexity of the OCPI middleware, validation of the full covariance coprocessor (OCPI+Covariance) did not occur via software HDL simulations, but instead at the hardware level once our covariance implementation was integrated with OCPI. Because our design and its Verilog wrapper/FIFO were validated by simulation, and because the interfaces between OCPI and our design were well-defined ahead of time, this "bottom up" integration was straightforward. This section discusses the testing

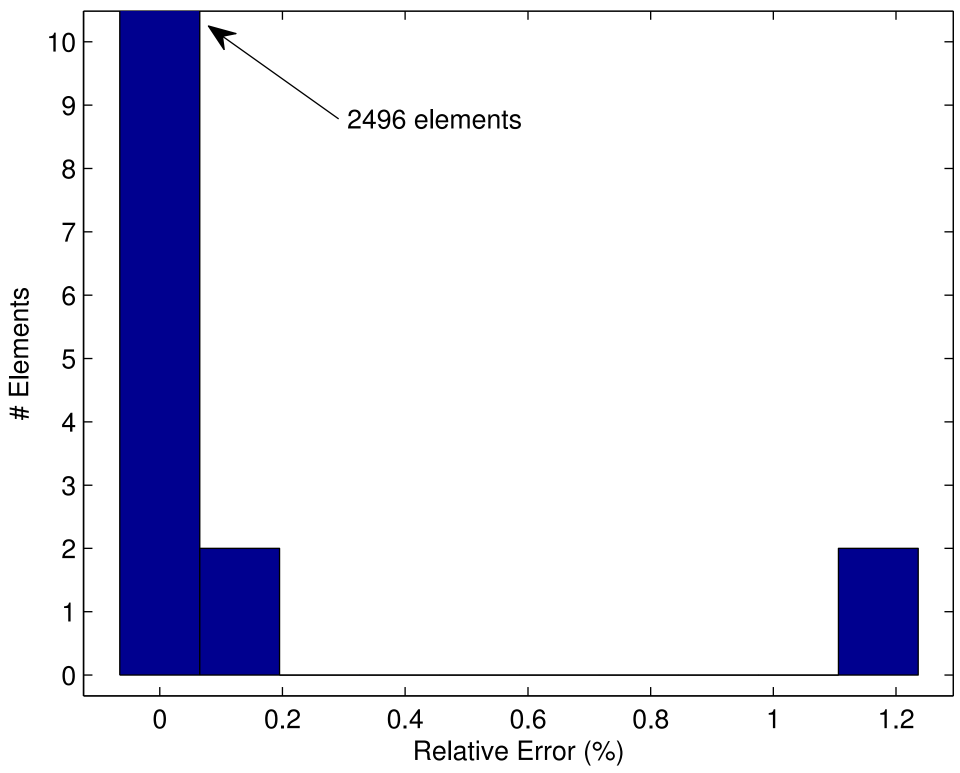

Figure 4.3: Relative Error of Covariance Matrix - Single vs. Double-Precision

that was performed on the fully-integrated hardware design.

Validation of DDR3 Memory Interface

In Section 3.3.2 we noted that it is important to keep the OCPI's DDR3 device worker filled with read requests in order to maximize the data throughput of the interface. It is also important that the read responses (requested data) are captured when they are available, or they will be lost. Both of these were validated in hardware. This was performed by monitoring the WCI control registers in the coprocessor during the first initial test runs.

The MemiTestWorker provided in OCPI, by default, keeps count of the number of memory read requests, read responses, and write requests in its set of WCI registers, listed in Table 4.3. These registers are used in the validation of the DDR3 memory interface. The shaded rows indicate registers that were added or modified in our modified MemiTestWorker to assist in this validation, and to help control overall coprocessor operation.

Validation of the DDR3 was performed using the modified OCPI tools to: program the FPGA bitstream to the ML605's FPGA, configure and start the OCPI middleware and MemiTestWorker, write the HSI image cube to the ML605 DDR3 memory, and command the coprocessor to begin the calculation. As mentioned in Section 3.4.1, it was necessary to add guarding to the read response channel on MemiTestWorker's side of the DDR3 interface. The need for guarding was discovered during this testing, as data loss was encountered.

Table 4.3: WCI Registers in Modified MemiTestWorker

Address	Name	Function	Read/Write
0x00	tstCtrl	Unused	RW
0x04	seqLen	# of 128-bit words per image cube	RW
0x08	wmemi.status	Status of DDR3 device worker	R
0x0C	loopCount	# of image cubes processed by coprocessor	R
0x10	errorCount	Unused	R
0x14	loopDuration	Duration of last coprocessor loop (# clocks)	R
0x18	numLoops	# of times to loop coprocessor ($0 = \infty$)	RW
0x1C	wmemiWrReq	# of WMemI write requests issued	R
0x20	wmemiRdReq	# of WMemI read requests issued	R
0x24	wmemiRdResp	# of WMemI read responses received	R
0x28	testStatus	Test Status (0=stopped, 1=running)	R
0x2C	maxRdReqInFlight	Max. # of WMemI read requests in flight	RW
0x30	testStart	Write to begin coprocessor image processing	W
0x34	testStop	Write to stop coprocessor image processing	W

The ChipScope Pro diagnostic tool was used to confirm the source of the data loss, and confirm proper operation after read response guarding was added. Figure 4.4 shows an ILA capture from ChipScope Pro of the 128-bit DDR3 memory read responses before guarding. The value outlined in red is an occurrence of valid data that is arriving earlier than it should, indicating that valid data just prior has been dropped. Figure 4.5 shows a capture of the same section of DDR3 read responses after guarding was added. The pixel data that was previously dropped can be seen in green, and proper performance of the DDR3 memory interface was validated.

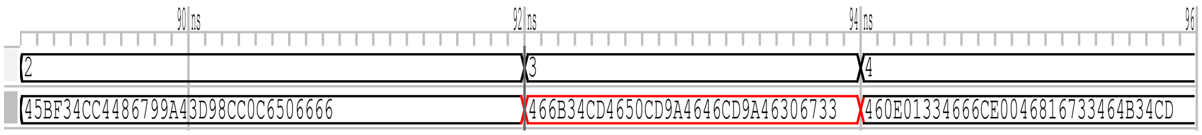
90 ns
92 ns
94 ns
96
2
3
4
45BF34CC4486799A43D98CC0C6506666
466B34CD4650CD9A4646CD9A46306733
460E01334666CE0046816733464B34CD

Figure 4.4: Dropped Data With > 4 Read Requests in Flight

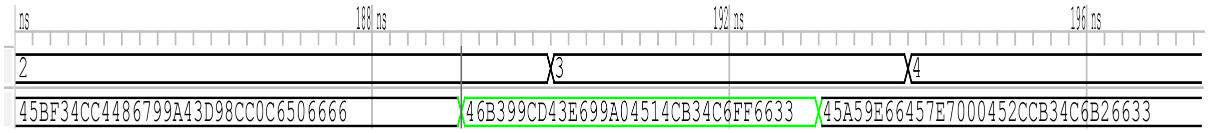
ns
188 ns
192 ns
196 ns
2
3
4
45BF34CC4486799A43D98CC0C6506666
46B399CD43E699A04514CB34C6FF6633
45A59E66457E7000452CCB34C6B26633

Figure 4.5: Read Responses After Added Guarding

Validation of Covariance Coprocessor

During the validation of the DDR3 interface, it was clear that the proper pixel values were being enqueued to the Verilog wrapper that contains the covariance module. Validation of the coprocessor as a whole required executing a test loop, verifying the expected number of WMemI write requests, and checking the contents of DDR3 memory to where the covariance was written.

The input image cube pixels and covariance result reside in different locations in the DDR3 memory. The DDR3 memory space for the ML605 is shown in Figure 4.6. For this design, the memory is byte-addressable and partitioned into 2^8 pages of 2^{19} bytes each (128 MB total). After a loop of execution, swctl confirmed that the correct number of write requests to DDR3 were reflected in the worker register. The swctl program was then used to set the DDR3 page register and perform a readback of the 10,000 byte covariance matrix. This data was compared to the HDL simulation

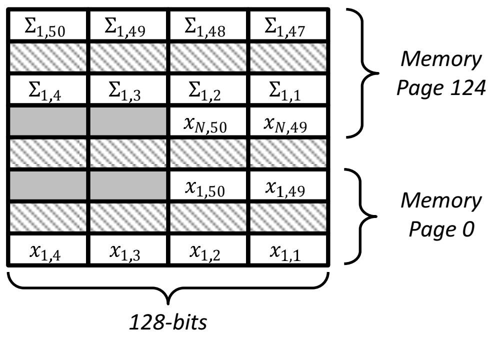

Figure 4.6: ML605/OCPI DDR3 Memory Space and Contents

results. The data matched, and therefore proper operation of the final design was validated. To confirm proper loop-to-loop operation, several loops of execution were performed, using the loopCount control register, and the resulting covariance was also found to be correct.

4.4 Coprocessor Performance

4.4.1 Runtime Performance

With the full coprocessor operating as designed, measurements were then made to determine the runtime of the coprocessor for the 512x614x50 image cube. The WCI *loopDuration* register in the modified OCPI worker was used to record the number of clock cycles that the covariance module takes to perform the computation for each loop of execution. These clock cycles include the movement of pixel data to the covariance module from ML605 DDR3 memory, and the movement of covariance results to ML605 DDR3 memory. The movement of the image cube from the Host PC to the ML605 DDR3 memory is not included.

On average, the coprocessor took 10,286,763 clock cycles to process a full image cube, or 82.3 ms. This corresponds to slightly over 12 image cubes per second processing throughput. The image cube requires approximately 1.57 GFLOP to compute, and therefore the coprocessor currently achieves 19.1 GFLOPS.

4.4.2 Power Consumption and Energy

Power consumption of the coprocessor was measured in two ways. The first method utilized the built-in capability of the ML605 evaluation board to monitor several of its power supply currents. This is accomplished through the use of the FPGA *system monitor* and additional circuitry present on the ML605.

The system monitor is a feature of the Virtex-6 FPGA that, when enabled, can perform analog-to-digital conversion of external signals. The ML605 is instrumented with five current-sense circuits that output an analog voltage proportional to the current passing through a current-sense resistor. This voltage is sampled by the FPGA system monitor and used to calculate the power consumption of the corresponding voltage rail. The OCPI ML605 middleware was modified to enable the system monitor in our coprocessor.

Figure 4.7 provides a photo of the typical current-sense circuit on the ML605. The large resistor (R365) is the current-sense resistor, and the IC beside it (U76) is a differential amplifier that provides the analog voltage measured by the system monitor. Table 4.4 provides a list of these voltage rails, along with the current-sense resistor values and analog gain of the amplifiers. This information is used to compute

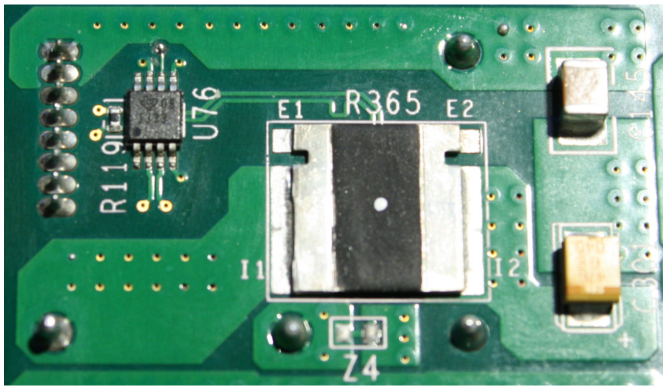

Figure 4.7: ML605 Current Sense Circuit

Table 4.4: ML605 Voltage Rails with Current Sensing

Voltage Rail Desc.	Voltage	Current Sense Resistor Value	Gain	System Monitor (C)urrent and/or (V)oltage Monitoring
Main ML605 Power Input	12V	$2m\Omega$	50V/V	C,V
FPGA VCCAUX	2.5V	$5m\Omega$	24.7V/V	None
FPGA VCC1V5	1.5V	$5m\Omega$	24.7V/V	None
FPGA VCCINT	1.0V	$5m\Omega$	24.7V/V	C
FPGA VCC2V5	2.5V	$5m\Omega$	24.7V/V	None

the voltage rail power consumption. Note that supply currents for only two of the five rails can be monitored by the system monitor.

The measurements from the system monitor are available in real-time using the Xilinx ChipScope Pro diagnostic tool. The supply voltage and current for the main ML605 +12V power input was monitored during coprocessor idle and active processing states. These results are provided in Table 4.5. Additionally, the total ML605

Table 4.5: ML605 +12V Supply Voltage, Current & Power Consumption

	ML605 Power-On	Coprocessor FPGA Bitstream Loaded	OCPI DDR3 Initialized	Coprocessor Running
Voltage (V)	12.17	12.17	12.17	12.14
Current (A)	1.21	1.81	1.86	2.13
Power (W)	14.7	22.0	22.6	**25.9**
Energy (J)	N/A	N/A	N/A	**2.1**

supply current was measured using a digital multimeter (DMM) wired in-line with the ML605 power connector. These measurements agreed with the system monitor measurements.

To obtain supply current measurements for the four FPGA supply rails in Table 4.4, a DMM was used to measure the current-sense circuit output voltages directly. This allowed a determination of the FPGAs contribution to the ML605 total power, and the increase in FPGA power during active processing. A breakdown of all voltage rails measured and power consumption for each is provided in Table 4.6. The total FPGA and ML605 power consumption, and the energy required for the covariance calculation is also provided in this table. Note that the ML605 power consumption increased by only 3.3W (15%) between idle and active processing states.

4.5 Comparison Platform Performance

The reference PC listed in Table 4.1 was used as a performance and power comparison platform. This platform consists of a Congatec conga-BM67 Single Board Computer

Table 4.6: FPGA & ML605 Power Summary

		Supply Currents by Mode			
Rail	Voltage	ML605 Power-On	Coprocessor FPGA Bitstream Loaded	OCPI DDR3 Initialized	Coprocessor Running
VCCAUX	2.5V	0.17A	0.86A	0.80A	0.81A
VCC1V5	1.5V	≈ 0A	1.03A	1.0A	0.95A
VCCINT	1.0V	1.87A	3.59A	4.03A	6.0A
VCC2V5	2.5V	0.05A	0.04A	0.04A	0.04A
ML605 Input	12.17V 12.14V	1.21A -	1.81A -	1.86A -	- 2.13A
Total FPGA Power (W)		2.42	7.39	7.63	**9.55**
Total ML605 Power (W)		14.7	22.0	22.6	**25.9**

(SBC) mounted to an ACTIS Computer KCAC-0320 carrier board that provides access to external peripherals and power. The BM67 contains a 2.1 GHz Intel Corei7 2710QE processor, 512MB of DDR3 memory, and runs CentOS Linux from flash memory.

4.5.1 Runtime Performance

Measurements were made to determine the runtime of the SBC on the 512x614x50 image cube. A multithreaded version of Algorithm 3 was written to perform calculations in parallel, much like the HW coprocessor. Each thread processed an equal share of the total number of pixels. Because the SBC platform's quad-core Intel Corei7 processor [18] supports up to eight threads, both four- and eight-thread versions of the code were executed. The code was compiled using gcc 4.7.2 with the *-lpthread -lrt*

-m32 -march=corei7 -mfpmath=sse -Ofast -flto compiler flags. As with our FPGA-based platform described in Section 3.3.1, the input pixel data is mean-subtracted per (2.1).

On average, the SBC took 79ms to process a single image cube using 4 threads, and 69ms using 8 threads. This corresponds to about 12.7 and 14.5 image cubes per second processing throughput, respectively. These measurements place the comparison platform at 22.8 GFLOPS during the covariance calculation.

4.5.2 Power Consumption and Energy

Power measurements were performed with a digital multimeter wired in series with the +12V DC power input to the SCB to measure average current. Measurements were taken during an idle state and during the execution of the four- and eight-thread code. The power measurements of the SBC and energy required for the covariance computation are included in Table 4.7. The ML605 coprocessor power and energy are included for comparison.

From these measurements, it can be seen that the SBC platform requires less power and energy when idle than the ML605. This is due to frequency scaling of the Corei7 CPU on the SBC, which lowers the clock frequency during idle states, reducing power consumption. However, the SBC platform dissipates more than twice as much power as the ML605 during active processing. The energy required for processing on the SBC is about twice that of the ML605 coprocessor.

Table 4.7: SBC Power and Energy vs. Coprocessor

	SBC Idle	SBC Running 4 Threads	SBC Running 8 Threads	ML605 Coprocessor
Voltage (V)	12	12	12	12.14
Current (A)	1.07	4.6	5.2	2.13
Power (W)	12.8	**55.2**	**62.4**	**25.9**
Processing Energy (J)	N/A	**3.8**	**4.3**	**2.1**

4.6 Size, Weight, and Power Implications

The performance, power, and FLOPS measurements are further utilized to investigate the attractiveness of both the covariance coprocessor and SBC from the SWaP perspective. One such metric to consider is the number of FLOPS per Watt of power consumed ($FLOPS/W$). This is a popular metric in the field, but it omits the weight portion of the system design, which is also of high importance for current and future airborne and spaceflight processing systems. Therefore, we also consider another metric, $FLOPS/(W \cdot kg)$ during the evaluation of these types of systems.

The ML605 and SBC were weighed to support this SWaP analysis, and the results are provided in Table 4.8. For this analysis, the average power seen during processing was used. This indicates that the ML605 performance in FLOPS/W is twice that of the SBC, and 2.75 times the SBC in FLOPS/(W·kg).

Table 4.8: ML605 and SBC SWaP Comparison

	ML605	SBC
FLOPS	19.1G	22.8G
Power (W)	25.9	62.4
Mass (kg)	0.336	0.476
FLOPS/W	**737M**	**365M**
FLOPS/(W·kg)	**2.2G**	**0.8G**

Chapter 5

Conclusions and Future Work

This research investigated and implemented a parallelized sample covariance calculation, targeted to an FPGA-based platform. This platform, the Xilinx ML605 was chosen because of its potential for lower size, weight, and power. An open-source middleware framework, OpenCPI, was utilized to extend this implementation into a completed coprocessor. This coprocessor was tested and found to provide around 12 sample covariance calculations per second on 512x614x50 AVIRIS hyperspectral image cubes. Performance was evaluated and found to be promising for use on SWaP-constrained HSI platforms, providing a greater than 2X improvement in power and energy dissipation over a CPU-based platform.

Contributions of our research include being the first coprocessor to perform FPGA-based covariance calculations on HSI data having a large number of spectral bands (50). This is considerably larger than related work, which processed the equivalent of 6 spectral bands. Additionally, this is the first time that a VHDL/Verilog user application has been successfully integrated and tested at this low a level inside the

OpenCPI ML605 framework. We have shown that FPGA-based processing platforms are an attractive option for the processing of HSI data, and are competitive from a size, weight, and power perspective.

5.1 Future Work

Because the coprocessor has a large percentage of FPGA resources still available, there is an opportunity to both expand the capabilities of this coprocessor platform and change the placement of the coprocessor in a future system.

FPGAs are commonly used to interface with imaging sensors of all types. Therefore, interesting future work will be to interface this coprocessor platform directly with an HSI sensor. This provides an improved environment, as it eliminates the need to transfer image data from the Host PC to the coprocessor. Only the results will need to be transferred externally by the coprocessor.

One way to accomplish this is to use the FMC expansion connectors on the ML605 board to interface with external cameras, using standard digital connections for commercial and scientific cameras such as *Camera Link*. A COTS FMC Camera Link interface board [19] has been purchased for use in future research to connect to a camera. A new OCPI device worker needs to be written in BSV to handle the interface with this expansion card and camera.

To improve the functionality of the covariance coprocessor, there are three major areas for future work. The first is to add the mean computation and mean-subtraction steps of the sample covariance calculation to the hardware implementation. This should be fairly straightforward to implement.

The second area is to change the input pixel format to use 16-bit 2's complement values for the pixels, and perform the conversion to floating-point inside the coprocessor. Because most cameras are 16-bits or less, this change would double the data throughput of pixel data into the coprocessor.

The final area for future work is the addition of HSI processing steps after the sample covariance calculation. The primary goals would be to add covariance matrix inversion, followed by a hyperspectral matched filtering algorithm, such as Adaptive Coherence/Cosine Estimator (ACE) [7].

Bibliography

[1] J. R. Schott, *Remote Sensing : The Image Chain Approach: The Image Chain Approach*. Oxford University Press, USA, 2007.

[2] G. A. Shaw and H. K. Burke, "Spectral imaging for remote sensing," *Lincoln Laboratory Journal*, vol. 14, no. 1, pp. 3–28, 2003.

[3] S. Cook and J. Harsanyi, "Onboard processor for compressing HSI data," in *Proceedings of the 31st Applied Imagery Pattern Recognition Workshop (AIPR02)*, 2002.

[4] Mercury Federal Systems, Inc. OpenCPI - open component portability infrastructure. http://opencpi.org. [Online; accessed 21-Feb-2013].

[5] J. M. Scott III, "Open component portability infrastructure (OpenCPI)," Mercury Federal Systems, Inc., Tech. Rep., Nov. 2009.

[6] C. I. o. T. Jet Propulsion Laboratory. AVIRIS - airborne visible / infrared imaging spectrometer. http://aviris.jpl.nasa.gov/index.html. [Online; accessed 21-Feb-2013].

[7] D. Manolakis, D. Marden, and G. A. Shaw, "Hyperspectral image processing for automatic target detection applications," *Lincoln Laboratory Journal*, vol. 14, no. 1, pp. 79–116, 2003.

[8] A. Paz and A. Plaza, "Clusters versus GPUs for parallel target and anomaly detection in hyperspectral images," *EURASIP Journal on Advances in Signal Processing*, vol. 2010.

[9] S. Bernabé, S. López, A. Plaza, and R. Sarmiento, "GPU implementation of an automatic target detection and classification algorithm for hyperspectral image analysis," *IEEE Geoscience and Remote Sensing Lett.*, vol. 10, no. 2, Mar. 2013.

[10] J. S. Costa, "Band selection techniques for hyperspectral chemical agent detection," Master's thesis, Northeastern University, 2011.

[11] M. L. Pieper, D. Manolakis, R. Lockwood, T. Cooley, P. Armstrong, and J. Jacobson, "Hyperspectral detection and discrimination using the ACE algorithm," in *Imaging Spectrometry XVI*, 2011.

[12] A. Plaza and C. Chang, "Clusters versus FPGA for parallel processing of hyperspectral imagery," *International Journal of High Performance Computing Applications*, vol. 22, no. 4, pp. 366–385, 2008.

[13] NVIDIA Corporation, "NVIDIA GeForce GTX 680 whitepaper," http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf, [Online; accessed 1-Feb-2013].

[14] B. Yao, H. Li, T. Zhou, B. Chen, and H. Yu, "Real-time implementation of multiple sub-array beam-space MUSIC based on FPGA and DSP array," in *Fifth IEEE International Symposium on Embedded Computing*, 2008, pp. 186–191.

[15] Military Embedded Systems. Two new rad-hard ICs from Texas Instruments enable power savings. http://mil-embedded.com/news/two-new-rad-hard-ics-from-texas-instruments-enable-power-savings/. [Online; accessed 1-Feb-2013].

[16] S. Martelli, D. Tosato, M. Cristani, and V. Murino, "Fast FPGA-based architecture for pedestrian detection based on covariance matrices," in *2011 18th IEEE International Conference on Image Processing*, 2011, pp. 389–392.

[17] R. S. Nikhil and K. R. Czeck, *BSV By Example*, 1st ed. Bluespec, Inc., 2010.

[18] Intel Corporation. Intel core i7-2710qe processor. http://ark.intel.com/products/53472. [Online; accessed 26-Feb-2013].

[19] Integre Technologies LLC. IP-CamLink. http://www.integretek.com/products/FMC200.html. [Online; accessed 1-Feb-2013].

Appendix A

Glossary of Acronyms

ACE Adaptive Coherence/Cosine Estimator.

AVIRIS Airborne Visible/Infrared Imaging Spectrometer.

BRAM Block RAM.

BSV Bluespec System Verilog.

COTS Commercial Off-The-Shelf.

CPI Component Portability Infrastructure.

CPU Central Processing Unit.

DDC Digital Down-Conversion.

DDR Double Data Rate.

DMM Digital Multimeter.

DSP Digital Signal Processor.

FF Flip-Flop.

FFT Fast Fourier Transform.

FIFO First In First Out.

FLOPS Floating Point Operations Per Second.

FMC FPGA Mezzanine Card.

FPGA	Field-Programmable Gate Array.
GPS	Global Positioning System.
GPU	Graphics Processing Unit.
HDL	Hardware Description Language.
HSI	Hyperspectral Imaging.
ILA	Integrated Logic Analyzer.
IMU	Inertial Measurement Unit.
LUT	Lookup Table.
MAC	Multiply-Accumulate Operation.
OCPI	OpenCPI.
PAR	Place And Route.
PC	Personal Computer.
PCI	Peripheral Component Interconnect.
QML	Qualified Manufacturer List.
RAM	Random Access Memory.
SBC	Single-Board Computer.
SVM	Support Vector Machine.
SWaP	Size, Weight, and Power.
TFBE	Target Free Background Estimation.
UAV	Unmanned Air Vehicle.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.
WCI	Worker Control Interface.

WMemI Worker Memory Interface.

WMI Worker Message Interface.

WSI Worker Streaming Interface.

WTI Worker Time Interface.

Appendix B

HSI Image Cube Parameters

- AVIRIS data product ID: “f970619t01p02r02c”
- AVIRIS data product type: Atmospherically corrected reflectance
- AVIRIS data product site name: Cuprite
- Spatial dimensions: 512x614 pixels
- Total number of bands: 224
- Number of bands used: 50

Appendix C

List of Software, Middleware, Diagnostic Tools, and Operating Systems

Software

- Xilinx ISE Design Suite 14.1 (Host PC)
- MATLAB 2011b (Host PC)
- OpenCPI “swctl” utility (Host PC)

Middleware

- OpenCPI (ML605)

Diagnostic Tools

- Xilinx ChipScope Pro (Host PC)
- ModelSim (Host PC)

Operating Systems

- Windows 7 64-bit (Host PC)
- Red Hat Linux Enterprise 5.9 64-bit (Host PC)
- CentOS 6.0 32-bit (Reference PC)