NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Vforce: VSIPL++ for Reconfigurable Computing Environments

Author: Nicholas John Moore

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Stefano Basagni	Date
Thesis Reader: Prof. Laurie King	Date
Thesis Reader: Dr. James Lebak	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Vforce: VSIPL++ for Reconfigurable Computing Environments

Author: Nicholas John Moore

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor: Prof. Miriam Leeser	Date
Thesis Reader: Prof. Stefano Basagni	Date
Thesis Reader: Prof. Laurie King	Date
Thesis Reader: Dr. James Lebak	Date
Department Chair: Prof. Ali Abur	Date

Graduate School Notified of Acceptance:

Dean: Prof. Yaman Yener	Date

Copy Deposited in Library:

Reference Librarian	Date

Vforce: VSIPL++ for Reconfigurable Computing Environments

A Thesis Presented

by

Nicholas John Moore

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

December 2007

Abstract

Systems with heterogeneous processing elements, such as commodity software processors combined with special purpose processors like FPGAs or GPUs, offer enormous potential speedups for certain types of workloads. There are, however, significant program development challenges on these systems. Programs written for these systems tend to have a lot of platform specific code integrated into the rest of the application code, making portability difficult. In addition, these systems have different programming models and tools requiring the developer to have hardware specific knowledge in addition to application domain expertise. Compounding these two problems is the short lifetime for these systems. A mechanism for portability across multiple architectures and generations is desirable. This thesis presents Vforce, an extensible framework that extends the VSIPL++ standard to add portable and transparent support for special purpose processors. New library elements that include portable special purpose processor support can be added to VSIPL++ through the use of Vforce's generic hardware interface – the user application code and binary contain nothing that is specific to the special purpose processors. The decision about which, if any, special purpose processor to use to execute the new library element is made at runtime by a hardware resource manager that runs on the system independent of the user application. This manager also provides the information necessary to bind Vforce's generic hardware interface to the specific API used by the selected special purpose processor. The implementation of Vforce and two specific usage examples,

an FFT and an adaptive time-domain beamformer, are discussed. Results for the two examples on a Cray XD1 heterogeneous supercomputer, as well as an analysis of the overhead added by Vforce, are presented. The results demonstrate the portability and performance achievable with the Vforce framework.

Acknowledgements

I would like to thank my advisor Professor Miriam Leeser. This work would not have been possible without her support, guidance, expertise, and extreme patience.

I would also like to thank Professor Laurie Smith King and the rest of the members of the Northeastern University Reconfigurable Computing Laboratory. You are all a pleasure to work with and special thanks goes to Professor King, Albert Conti, Ben Cordes, and Kris Kieltyka for contributing much to the Vforce project.

This research was supported in part by a subcontract from ITT Industries under a grant from the Air Force, and by donations from the Xilinx Corporation. The Cray XD1 used for this project was made available by the Ohio Supercomputing Center.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hybrid architectures that closely integrate traditional software processors with reconfigurable processors, such as FPGAs, have emerged as a powerful computing platforms ranging from embedded systems to supercomputers. In addition, a variety of new hardware architectures have been brought to market and are becoming more popular, including the Cell Broadband Engine and GPUs for general purpose computing (GPGPU). These new architectures promise performance improvements on a range of program types by exposing more parallelism to the application programmer. Part of the parallelism is coarse grained and enabled by processor duplication, such as multiple multiple FPGAs and von Neumann-style processors in hybrid platforms or the multiple Synergistic Processing Engines in the Cell. This type of parallelism is not exclusive these newer hardware architectures, as relatively new multi-core, traditional supercomputer, and distributed architectures all take advantage of this type of parallelism in workloads.

Much of the computation in a variety of workloads is inherently parallel. Even

with out of order execution, modern CPUs are largely serial processors. The Cell's SPEs are SIMD in design and modern GPU designs include a large number of data parallel floating point engines. FPGAs, while more difficult to program, offer the greatest degree of control over the computation implementation and can often take advantage of more fine-grained parallelism as the hardware design can be tailored to the specific application.

While these new platforms offer large potential gains in performance, they present several challenges to application developers. First, applications written for these SPPs often include large amounts of vendor specific code directly in the program, and this code is usually mixed in with the regular application code. This makes it difficult to port the application to new devices, and often requires reimplementing some of the same functionality for the same device in each application. Second, these devices require the programmer to have much more knowledge about the target hardware than programming for traditional CPUs. Without this additional skill it is often very difficult to achieve much of the theoretically available performance. In addition, the Cell and GPUs have relatively immature tools and libraries, while FPGAs are too flexible to create a general tool that generates designs from high-level programming languages as effectively as modern software compilers can compile programs for general purpose processors.

There are many types of special purpose processors (SPPs), but they share a similar set of characteristics and control operations. They require different programming

models and tools than those normally used in traditional software development. The separately or pre-compiled binary or configuration must be loaded onto the device, and usually the data transfer to and from the SPP as well as SPP execution and synchronization must be managed by the user application.

The author worked with other members of the Reconfigurable Computing Laboratory at Northeastern University to develop an extensible framework called *Vforce* (**V**SIPL++ **FO**r **R**econfigurable **C**omputing). Vforce is built on top of the VSIPL++, a C++ based extension of the Vector Signal Image Processing Library. VSIPL++ specifies a library of portable interfaces for data representation and many common image and signal processing algorithms. While promising portability, VSIPL++ also stresses performance and allows implementations to select platform optimized implementations of the library's functionality. More information on VSIPL++ is given in Section 2.1.

Vforce extends VSIPL++ to support SPPs in a way that is transparent to the application programmer and highly portable. Vforce presents the application programmer with additional algorithm implementation classes, referred to as processing objects. These Vforce processing objects encapsulate two implementations of the given algorithm: a software/hardware version that utilizes SPP implementations of the algorithm and a software only version. The software/hardware implementation may make use of any VSIPL++ or C++ code to be executed on a GPP in addition to Vforce's Generic Hardware Object (GHO). The GHO provides a common portable

interface to multiple SPP types, and its methods implement a comprehensive set of the SPP operations shared between device types. Through the use of the GHO and the dual software/hardware and software only implementations, Vforce processing objects are as portable as the VSIPL++ implementation that Vforce is compiled on top of. When an SPP is available to accelerate the given algorithm, the Vforce processing object can use the GHO to control the SPP implementation. If there is no available SPP that can run the algorithm or there is an error during the use of the SPP, the software only implementation is executed on the GPP. More detail on Vforce processing objects and the GHO are provided in Section 3.2.1.

To provide the link from the hardware abstraction offered by the GHO down to the actual hardware, Vforce includes the runtime resource manager (RTRM) and SPP specific dynamically linked shared libraries (DLSLs). DLSLs are pre-compiled shared libraries that can be loaded by the GHO to convert the generic SPP operations provided by the GHO to the specific SPP API calls required to control a specific SPP. As such, the DLSLs are SPP type specific and one must be created for each SPP API supported by Vforce. The DLSL contains all of the platform specific GPP code needed to control the SPP, keeping it separate from the user application binary until runtime.

Like the SPP specific GPP code, the actual kernels executed on the SPPs are not part of the Vforce application binary. Vforce does not compile code for SPPs; it relies on a library of pre-built kernels. This requirement does not remove the need for the

aforementioned hardware specific knowledge from the total application development process, but Vforce facilitates kernel reuse and helps to separate the hardware specific knowledge from application development.

The RTRM exists outside of the user application as a standalone system daemon. It maintains a listing of available SPPs (and their corresponding DLSLs) in a system and waits for a Vforce user application to request an SPP hardware device to execute a specific algorithm. These requests are generated by the GHO when a user application begins executing the hardware/software implementation. The manager indicates to the GHO whether an SPP kernel in the system can execute the desired algorithm and, if so, the DLSL that corresponds to the selected SPP. The GHO then uses the specified DLSL to control the assigned SPP. Section 3.2.2 covers the RTRM and the DLSLs and their interaction with the rest of the Vforce framework.

After introducing the Vforce framework, two cases studies using Vforce to implement applications are provided in Chapter 4. The first application, a parametrized FFT, provides a Vforce SPP implementation for a standard VSIPL++ processing object. The second application is a time-domain adaptive beamformer. Beamforming is a signal processing technique that can focus an array of antennas on signals impinging upon the array from specific directions and/or frequencies. A brief overview of beamforming is provided in Section 2.2. The beamformer provides an example of using Vforce to construct processing objects that are larger in scope than most existing in the VSIPL++ standard and consists of multiple sub-tasks. This increase

in granularity often meshes well with the performance characteristics of SPP devices and makes the computation requirements large enough to obtain significant speedup with a combined hardware and software beamformer implementation utilizing Vforce's ability to perform work on both the GPP and SPP simultaneously. Chapter 4 discusses the SPP kernel implementations for these applications as well as the DLSL and RTRM implementations. Both applications were implemented on a Cray XD1.

Chapter 5 explains the procedures used to test the two applications and analyzes the results. The FFT was tested to show the overhead of Vforce for both possible execution paths. The differences in performance between the FFT Vforce processing object executing the SPP implementation and a C application written to the Cray XD1's FPGA API are examined to show the overhead Vforce adds when utilizing SPP hardware. Likewise, Vforce executing the software only implementation is compared to an equivalent non-Vforce VSIPL++ FFT application. The FFT results indicate that the Vforce framework adds little overhead with the exception of a data copy. Tests and analysis for the beamformer focus on the level of concurrent execution obtained using Vforce and compare the results to a previously existing beamformer implementation on a Mercury Computers system. The beamformer results show that even with the conveniences offered by Vforce, it still allows applications that make full use of concurrent execution between SPPs and GPPs. This is a necessary characteristic for obtaining the maximum performance available on many platforms.

Finally, Chapter 6 concludes and presents possible areas where future work may be directed. In particular, it suggests a possible solution to the data copying problem, which would remove the current bottleneck in the Vforce framework.

Chapter 2

Background

This chapter briefly discusses a couple of topics that are important to understand the work presented in this document, as well as related work.

2.1 VSIPL++

VSIPL++[10] is a C++ language extension of the the Vector Signal Image Processing Library (VSIPL)[25], originally specified for the C programming language. Both the VSIPL and VSIPL++ standards were created and are maintained by the VSIPL Forum[26] of the High Performance Embedded Computing Software Initiative (HPEC-SI). The VSIPL Forum is an open organization made up of industry, government, and academic representatives. The goals for the VSIPL++ standard[27] include high levels of portability, performance, and productivity. The standard specifies a library of classes and functions that includes a lot of functionality useful for signal processing applications, including linear algebra solvers, transforms, filters, statistics, and mathematical functions, among others. Combined with the object-oriented nature of the library, VSIPL++ provides a relatively clear and concise vehicle for high

levels of programmer productivity.

The other two goals, performance and portability, are often considered to be somewhat at odds with each other, and together helped to determine the structure of the library. VSIPL++ separates the implementation of the library's functionality from the program's specification, as shown in Figure 2.1. The user program itself, as long as it uses the standard VSIPL++ API, is portable to any platform where a VSIPL++ implementation exists. To provide performance, a VSIPL++ implementation may rely on high performance libraries that are optimized for the specific machine the program is being run on. In an application that relies on an FFT, for example, the VSIPL++ implementation may itself rely on the FFTW3 library[7], which runs on general purpose processor. There is a reference implementation of the VSIPL++ library provided by HPEC-SI that is a wrapper around the reference implementation of the VSIPL library, which provides the necessary functionality. Other VSIPL++ implementations may rely on other libraries optimized for the target platform, as shown in Figure 2.1.

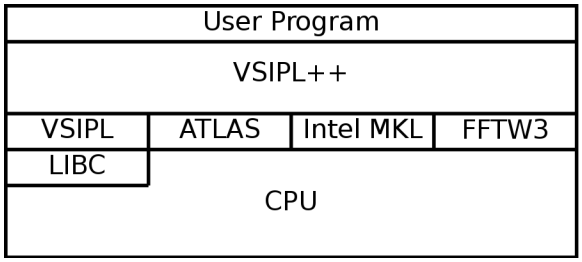

Figure 2.1: A graphical representation of the separation of specification and implementation in VSIPL++

There are two basic types of classes in the VSIPL++ specification: processing

classes and data classes. A processing class implements some specific functionality and acts on any number of data classes, which are storage container types. In VSIPL++, the data classes are called views and may be one, two, or three dimensional. A view isn't associated with any computer memory itself but instead is associated with another VSIPL++ type, called a block. Figure 2.2 depicts how a matrix view is associated with the underlying two dimensional block. A block provides a logically contiguous memory space in one, two, or three dimensions. A view is a particular way to look at the data in the underlying block and is characterized by an offset, a stride, and a length for each dimension.

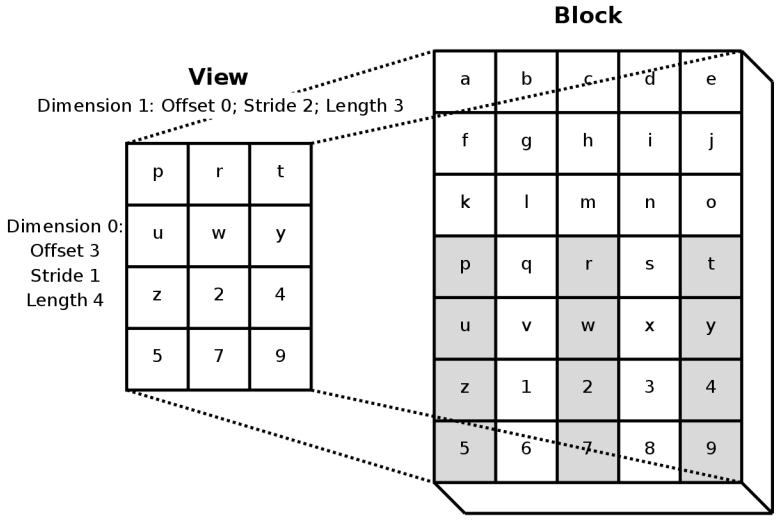

Figure 2.2: A graphical representation of the relationship between views and blocks in VSIPL++

The processing objects in VSIPL++ operate on views, and the view and block mechanism provides a powerful way to select specific subsets of data for operating on without copying the data into a new container. Views into views, called subviews,

can also be created, allowing the user to select rows, columns, diagonals, and most other data subsets of interest, and these subviews can be created to copy the data into new blocks or just reference the same underlying data.

Vforce, discussed in Chapter 3, is built on top of VSIPL++. VSIPL++'s object-oriented design cleanly separates individual processing objects making it easy to add additional processing objects as part of the Vforce framework. The library's focus on portability while maintaining performance is also aligns with the goals of Vforce. VSIPL++ explicitly separates the interface for the processing object from the implementation (Figure 2.1), a concept that is similar to how Vforce selects an implementation of a processing object, software or hardware.

2.2 Beamforming

As a sample application, a Vforce implementation of an adaptive time-domain beamformer was created. The implementation is discussed in Section 4.3, and the performance results of the implementation are covered in Section 5.2. Beamforming is a signal processing technique that controls the sensitivity pattern of an array of antennas. By adjusting the gain and phase or time delay of the incoming sensor data, a beamformer can increase gain in selected directions or frequencies and place nulls in the sensitivity pattern in the direction of interfering signals[22]. Beamforming can be performed in either the frequency domain, where phase shifts are used, or in the time domain, where time delays are used. In both cases weights are computed for

each sensor and for each beam that modify the gain and phase of the sensor reading. Figure 2.3, originally from [2], shows how beamforming is a form of spatial filtering. An incoming plane wave (curvature of the wave front is negligible) will strike the various sensors in a sensor array at different times. By choosing a reference point, the center of the array in Figure 2.3, the relative arrival times (when considering time domain beamforming) will vary. The weights combined with the time delays make it possible for beamformers to be highly selective in the incoming signals that they focus on.

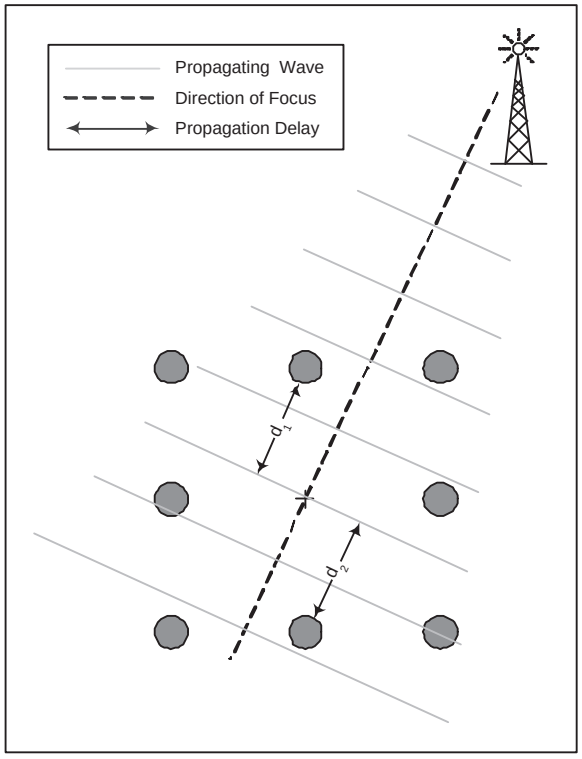

Figure 2.3: A graphical representation of the spatial nature of beamforming

There are a large number of types of beamformers and beamforming algorithms, but there are two basic categories: fixed and adaptive. Fixed beamformers use a

static set of sensor weights and time-delays and work well in situations where the desired incoming signal characteristics are known and the environment is relatively static. Adaptive beamformers continuously update the weights to adjust to changing environmental conditions and changing interference. However, depending on the type of algorithm used, adaptive beamformers can require significantly more processing power to process incoming data at the same rate as fixed beamformers since new weight computation can be complex. As a compromise, an adaptive beamformer can incorporate a static beamformer as well with the goal of reducing the computation requirements involved with a fully adaptive beamformer; this is known as a partially adaptive beamformer. In the case where a beamformer is adaptive, there may be variation in the frequency of the weight updates. The length of the weight update period can be continuous, performing weight computation after every time step, or the beamformer may implement block adaptation, where the weight computation is performed after several time steps. Continuous adaptation generally provides the best response times to changing environments, but can result in significant computational requirements. Setting a longer period between weight computations allows the designer to settle on a trade-off between responsiveness and the required processing power. Finally, all types of beamformers are affected not only by the specific algorithm used, but also the number and layout of the sensors, adding another dimension to beamformer design [23, 22].

2.2.1 Implemented Algorithm

Adaptive beamforming was chosen as a demonstration application for the Vforce framework, because it provides a good vehicle to illustrate the reasons that the Vforce framework was originally created, including adding SPP support to VSIPL++, easing software/hardware co-design, and providing portability while maintaining performance. The algorithm was previously chosen for implementation [2], and the original reasons behind this decision are summarized here.

As mentioned, the basic time domain beamforming operation applies weights to the time delayed sensor readings. These modified sensor readings are then summed to produce the final output for each time step, as shown in Equation 2.1. This step is referred to as weight application in this document.

$$y(k) = \sum_{i}^{n} w_i * x_i(k - d_i) \tag{2.1}$$

where k is the time step, n is the number of sensors in the array, w_i is the weight for the i^{th} sensor, $x_i(k)$ is the sensor reading for the i^{th} sensor at time k, and d_i is the time delay for the i^{th} sensor. This process is repeated for each beam of interested.

The time delays (d_i) calculation is relatively straightforward and is based on the array geometry, the propagation speed of the waves, and the direction the beamformer is looking for an incoming signal. The time delays only need to be recalculated if the location of a sensor, a beam direction, or the propagation speed changes. However, the complexity of the weight computation varies widely between weight computation algorithms. There are numerous weight computation algorithms, the proper selection

of which is beyond the scope of this document. However, for the purpose of demonstrating Vforce, an algorithm was desired that justified moving the beamformer from a monolithic software or hardware implementation to a combined hardware/software co-design. The weight application process is computationally dense with little control overhead other than address generation – a good match for FPGAs and many SPPs in general. There is also a large amount of possible parallelism due to the lack of data dependencies between weight updates. With that in mind, an algorithm that required solving for the least squared error solution to a overdetermined set equations was chosen. While the complexity of the algorithm varies with the number of sensors and beams, it grows in complexity quickly. This type of algorithm is also difficult to implement on FPGAs and displays limited scalability, making it a good candidate for implementation in software. In order to make comparisons easier, the algorithm chosen for implementation here was the same as in [2]. More details on the reasons behind choosing the algorithm and the algorithm itself can be found there.

2.3 Related Work

This section originally appeared in [15], and covers some projects related to Vforce.

As more and more reconfigurable computing platforms become available, the interest in simpler ways to make efficient use of these systems increases. We discuss related work that focuses on reconfigurable application design and portability. This work is similar to Vforce in that it is based on an execution model where pre-existing

FPGA designs are accessed at run time.

Researchers at the University of Kansas have developed *hthreads* for specifying application threads running within a hybrid microprocessor-FPGA system [1, 17]. Their system supports a master-slave model with one microprocessor tied to an FPGA. The support for hardware threads requires part of the system to run in hardware on the FPGA, and requires a fair amount of overhead. Elements of the operating system that handle context switching and semaphores are implemented on FPGA fabric. This reduces the time required for switching context from one thread to another and communicating from one thread to another. This comes at the cost of a distributed operating system and area on the FPGA that cannot be used for implementing a circuit to accelerate an execution thread.

Vuletic et al. [28] use threads both in master-slave mode and in a more general network with FPGAs acting as peer processing elements. Their approach uses threads, and is based on an abstraction layer that uses a virtual memory model. A virtual memory handler must run in FPGA hardware to resolve accesses not in local memory. Similar to *hthreads*, this requires a fair amount of overhead. Like the *hthreads* system, operating system components are moved to FPGA fabric to create a distributed operating system, complicating the system and consuming FPGA resources. In the more general network approach, the hardware must include a communication agent that handles communication over the network as well as resolving memory accesses so that the FPGA can behave as a peer to a processor.

Researchers at the University of Florida have developed a system to provide run time services for systems that include heterogeneous hardware. Their system consists of two parts, USURP (USURP's Standard for Unified Reconfigurable Platforms)[11] and Carma (Comprehensible Approach to Reconfigurable Management Architecture)[5]. Their system supports general distributed systems where individual processors may have an attached reconfigurable hardware accelerator. USURP is built on top of MPI and is distributed, with a small manager running on every node. These researchers propose a standard interface for hardware designers to use at design time in order to support run time portability and services such as performance monitoring and debugging. Their API is lower level than ours, and requires that users make calls to specify and download bitstreams, transfer data, etc. In our model, these operations are hidden inside functions and not exposed to the programmer.

Auto-Pipe[8] is a tool developed at Washington University in St. Louis that aids in the design, evaluation and implementation of pipelined applications distributed across a set of heterogeneous devices such as microprocessors and FPGAs. Auto-Pipe compiles high-level source code, partitions computation and maps components to different processors in a pipelined fashion. The tool suite is broken into stages and applications are designed and optimized in an iterative process. In the final design stage, pipelines can be adjusted in response to run time performance. Currently, this tool focuses on binding functions to resources at design time.

Our approach differs from the above approaches in several important ways. First,

our application code does not change from an all software implementation to a software/hardware implementation. Second, our approach does not require any support on the reconfigurable hardware itself. This makes our approach more flexible since it can make use of any vendor's API. We do not change the way hardware is implemented or used, only the way it is invoked by software. Finally, our approach is lighter weight than other approaches, introducing minimal overhead.

2.4 Summary

This chapter has presented background information on VSIPL++ and beamforming. In addition, it discussed related work by other researchers working on portability and runtime services for special purpose processors. The next chapter presents the Vforce framework.

Chapter 3

Vforce

This chapter presents Vforce, a middleware framework that was developed by the author along with students and researchers in the Reconfigurable Computing Laboratory at Northeastern University. Vforce allows programmers to automatically take advantage of SPPs in their projects while maintaining a high level of portability. Vforce is an acronym that stands for **V**SIPL++ **FO**r **R**econfigurable **C**omputing **E**nvironments. The framework is built on top of VSIPL++ and maintains VSIPL++'s object-oriented paradigms to present a familiar environment while enabling the possible benefits of SPP use and maintaining portability. The framework attempts to achieve these goals automatically so that the effort that is required by the end user is comparable to using only VSIPL++. Vforce has been described in a number of publications [15, 14, 2].

3.1 Goals of the Vforce Project

The projects primary goal is to add SPP support to VSIPL++ in such a fashion that maintains portability, a central goal that HPEC-SI has for VSIPL++. In addition,

this support for SPPs should work for existing VSIPL++ source code, maintaining VSIPL++'s level of portability. This precludes changes to the existing VSIPL++ API, as existing code must run unaffected when run or compiled with Vforce. However, this does not preclude additions to the VSIPL++ specification, which is the route chosen by Vforce to implement its functionality. Maintaining the correctness of programs written to the existing VSIPL++ specification requires the framework to mask all SPP related activity, such as programming and data transfer, as well as prevent any additional error modes from affecting the output of the legacy code. This implies that Vforce must gracefully handle SPP related errors and still produce the appropriate behavior and results. Furthermore, Vforce should be as extensible as possible, allowing new SPPs and new platforms to be supported with relative ease and further increasing the level of portability.

Maintaining high performance on legacy code as well as achieving the performance improvements offered by SPPs was also a goal of the Vforce project. Vforce should introduce as little overhead as possible to prevent performance degradation in applications that do not take advantage of Vforce or on systems where SPPs are not available. Additionally, the APIs and mechanisms used by Vforce should allow Vforce enabled applications to take advantage of as many features of the wide variety of SPP types available as possible. For many applications, this means adding support for concurrent processing of different tasks, usually one on the CPU and another on the SPP, something not included in the current VSIPL++ specification.

3.2 The Vforce Framework

Vforce consists of several components that interact to provide all of the services offered by the framework. These include C++ classes, a system-wide resource manager, SPP control libraries, and SPP kernel libraries. While the C++ classes are compiled into the user application, the remaining components exist independent of the user application and are considered to make up the runtime portion of the framework. The C++ classes include Vforce enabled processing object classes and the Generic Hardware Object (GHO), a class that provides Vforce applications a uniform interface to control SPPs.

3.2.1 Compile Time: Vforce Processing Objects & the Generic Hardware Object

Despite the variety of additional and disparate components in Vforce, the end user still only interacts with processing objects. Vforce processing objects do not change VSIPL++'s concept of a processing object. They exhibit similar behavior when treated like existing VSIPL++ processing objects and only deviate to add extra methods that enable task concurrency. However, in order to work seamlessly in environments both with and without SPP hardware, a Vforce processing object must internally contain two implementations of the given processing object's functionality: one for software execution and one for execution on a SPP. The software implementation may be realized in any fashion and can possibly use other VSIPL++ functionality to perform the appropriate processing. The SPP version takes advantage of the GHO

in order to interact with the hardware-based kernel.

The GHO represents an abstract SPP that can substitute for many types of SPPs. Its methods provide an API that allows Vforce processing objects to interact with SPPs through the same type of operations used by a program written for a specific SPP's API. This generalization of APIs is possible because most SPP APIs have a similar set of functionality, including loading processing kernels, configuration of those kernels, data transfer, and controlling execution and synchronization. The GHO provides methods for these common operation types and allows the Vforce processing object code to be independent of any particular SPP type while maintaining the original SPP control behavior. The SPP version of the algorithm implementation may mix GHO commands with any other code, including VSIPL++ code. The set of operations currently supported by the SPP are listed in Table 3.1.

Table 3.1: The GHO API

GHO Method Description
template<typename, typename>class const_View>void put_data(const_View<T, Block>data, unsigned int bank) template<typename T, typename Block, template<typename, typename>class const_View>void get_data(const_View<T,Block>data, unsigned int bank) Blocking; Sends data to or receives data from, respectively, the SPP
template<typename T, typename Block, template<typename, typename>class const_View>void put_data_start(const_View<T, Block>data, unsigned int bank) template<typename T, typename Block, template<typename, typename>class const_View>void get_data_start(const_View<T,Block>data, unsigned int bank) Non-blocking; Sets up and begins a data transfer between the SPP from the Vforce user application.

Continued on Next Page...

Table 3.1 Continued

GHO Method Description

`int put_data_status(void)`
 	
`int get_data_status(void)`
 	
Non-blocking; Checks whether an outstanding data transfer has completed.

`int put_data_finish(void)`
 	
`template<typename T, typename Block, template<typename, typename>class View>int get_data_finish(View<T,Block>data)`
 	
Blocking; Complete an outstanding data transfer, waiting if necessary.

`void put_const(unsigned long *data, unsigned int num, unsigned int bank)`
 	
`void get_const(unsigned long *data, unsigned int num, unsigned int bank)`
 	
Write or read kernel configuration parameters.

`void kernel_init(char *kid)` Obtain and setup a SPP with the specified kernel. RTRM involved in this step.
`void kernel_dest()` Relinquish ownership of the SPP. RTRM involved in this step.
`void kernel_run(bool blocking)` Start the kernel on the SPP. If `blocking` is true, the call is blocking, otherwise non-blocking.
`bool poll_int()` Check for execution completion of the SPP kernel. Only useful if `kernel_run` called in its non-blocking form.
`void clear_int()` Clear the interrupt state of the SPP and prepare it for another execution.

Note that all the methods above also have the `throw(gho_error)` exception specifier, but it is omitted here.

As mentioned, there are two implementations of the algorithm: one in software only and one that utilizes the GPP. While the software version is needed for executing the given kernel when no SPP hardware is available to carry out the given

task, but it is also important for fault tolerance and hiding all SPP activity from the end user. Whenever the GHO encounters a problem or the SPP indicates an error, the GHO will throw a `GHO Error` exception. This exception, subclassed from VSIPL++'s `computation_error` exception, must be caught by the Vforce processing object, which then transfers execution from the SPP implementation to the software version. The software version itself may end up throwing a `computation_error` exception up to the user program, as specified by the VSIPL++ specification, but the user program will never receive SPP or Vforce related errors.

To seamlessly add SPP support to legacy VSIPL++ applications, Vforce interposes replacement Vforce processing objects in the place of existing VSIPL++ processing objects whenever a Vforce alternative exists. This requires recompilation of the source code, but no source code modification. Currently, this mechanism is enabled by changing the header file include directory given to the compiler from the regular VSIPL++ directory to the location of the Vforce framework, which is a mirror of the VSIPL++ include directory containing links to the VSIPL++ header files. For VSIPL++ processing objects where a Vforce replacement exists, the link is replaced with the Vforce header file and is included directly. A demonstration of interposing Vforce processing objects in place of the original VSIPL++ processing object is the FFT application discussed in Section 4.2.

While replacing existing VSIPL++ processing objects with Vforce processing objects is possible for a number of processing objects from the VSIPL++ specification,

it is not always appropriate. An important consideration for determining the appropriateness of candidate algorithms for acceleration is the granularity of the processing kernels. While SPPs may be able to perform certain classes of operations much faster than the hosting GPP, there are overheads associated with SPP use, including setup (FPGA programming, configuration, etc.) and data transfer. While these overheads vary among SPPs, in general there must be enough work in the algorithm kernel (large enough granularity) so that the processing speedup offered by the SPP compensates for these overheads. While the total speedup for an algorithm, including overheads, may not need to be greater than one for the total application to post speedups greater than one (due to the task concurrency offered by SPPs and taken advantage of by Vforce), in general SPPs become more advantageous as the task speedup increases. The VSIPL++ specification includes a large number of operations, many of which are not large enough to overcome SPP overhead. To combat this issue, Vforce processing objects can be created that perform larger amounts of work. The SPP kernels associated with these processing objects will provide a speedup for the entire algorithm, and the software implementation contains the same code, possibly constructed from many individual VSIPL++ operations. Figure 3.1 represents this relationship and is based on Figure 2.1. The "Vforce VSIPL++" box represents the Vforce framework and indicates that Vforce processing objects may be executed on a SPP, in VSIPL++ software, or a combination of both. The SPP implementation in the Vforce processing object may utilize multiple sub-processing

objects that each may request an SPP, can instantiate and use multiple GHOs, and can use C++ and VSIPL++. The box, no higher than that of VSIPL++, also signifies the fact that Vforce processing objects may be used to replace existing VSIPL++ processing objects, as was previously discussed.

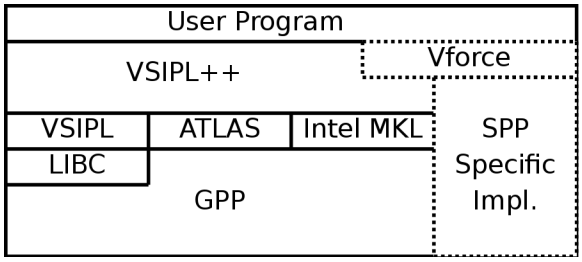

Figure 3.1: A graphical representation of Vforce's relationship with VSIPL++

In order to extract the maximum amount of performance from a machine that contains one or more SPPs, it is often necessary for an application to exploit concurrency: simultaneous execution on both the GPP and SPP. Concurrent execution allows larger speedups than would be possible from the speedup provided by the SPP on its kernel alone. In order to facilitate concurrent SPP and GPP execution, a rudimentary mechanism for task concurrency has been added to the GHO and can be used by Vforce processing objects to allow task concurrency at both the Vforce processing object and end user level. Specifically, the GHO enables concurrency for both kernel execution and data transfer by breaking each into three separate methods: start, status, and finish. A Vforce processing object may use this to run some GPP code while working with an SPP to maximize fine grained parallelism, export the task parallelism to the user through a similar set of start, status, or finish calls,

or both. As discussed in Chapter 4, both the FFT and beamforming applications on the Cray XD1 take advantage of task concurrency - the beamforming Vforce processing object uses it internally and the FFT Vforce processing object exposes task concurrency to the end user.

3.2.2 Vforce at Runtime: the RTRM, DLSLs, the GHO, and SPP Kernel Libraries

The remaining Vforce components, along with the GHO, work together at runtime to provide the rest of Vforce's features. These additional components include the Real Time Resource Manager (RTRM), SPP Dynamically Loaded Shared Libraries (DLSLs), and the SPP kernel libraries.

The GHO provides a common interface that abstracts varying SPP APIs and is compiled into the user application. Since decisions about whether or not to use SPP hardware and which SPP to use are made a runtime, the SPP API to target is not known at compile time. As a result, Vforce includes no code specific to a particular SPP at compile time and instead loads this information at runtime. This is made possible through the use of DLSLs, which contain object code that can be loaded into a program's memory space at runtime. While it is more common for shared libraries to be dynamically linked at loadtime by the operating system's linker, the loading of shared libraries can be controlled by the program itself. Vforce DLSLs contain a predefined set of functions that implement a second API at a lower level than the API that the GHO presents to Vforce processing objects. A particular DLSL contains the

object code necessary to access the corresponding SPP and provide the final step in translating the GHO API to a particular SPP API. Table 3.2 shows the current set of functions that a Vforce DLSL must implement. Once a GHO loads a DLSL using the POSIX dynamic linking API, a Vforce processing object may invoke one of the GHO's methods. The GHO then uses the functions in Table 3.2 to implement the desired functionality. While some of the functions have a direct one-to-one correspondence with those from the GHO API in Table 3.1 (`poll_int` and `pe_poll_int` for example) others do not, in which case the GHO uses a combination of the DLSL functions. The GHO is also responsible for searching the DLSL for the necessary functions and converting from VSIPL++ data types to the native C types used in the DLSL API. A DLSL for the Cray XD1's FPGA Application Accelerators has been created, and it implements all of the functionality in Table 3.2. More details on the XD1 DLSL are provided in section 4.1.1.

Table 3.2: The DLSL API

`DLSL Function` Description
`int pe_kernel_init(char *kid, int program_pe, char *device)` Performs any SPP or kernel initialization required. If `program_pe` is non-zero also program the device (used on systems where RTRM programming is not supported).
`int pe_program_pe(char *kid, char *device)` Program the SPP with the kernel specified by `kid`. Usually called by the RTRM.
`int pe_kernel_dest(char *device)` Performs any finalization required by the SPP after the Vforce user application has finished using the SPP.

Continued on Next Page...

Table 3.2 Continued

DLSL Function Description
`int pe_kernel_run(int blocking, char *device)` Start the execution of a previously programmed kernel.
`int pe_poll_int(char *device)` Check the execution status of the SPP
`int pe_clear_int(char *device)` Prepare the SPP for another kernel execution.

`int pe_put_const(unsigned long *data, unsigned int num, unsigned int bank, char *device)`
 	
`int pe_get_const(unsigned long *data, unsigned int num, unsigned int bank, char *device)`
 	
Write or read kernel configuration constants.

`int pe_put_data(void *data, unsigned int num, unsigned int bank, char *device)`
 	
`int pe_get_data(void *data, unsigned int num, unsigned int bank, char *device)`
 	
Send or receive data to or from the SPP, respectively. Used when the SPP does not support DMA data transfers.

`void* pe_get_dma_buffer(size_t size, char *device)`
 	
`void pe_free_dma_buffer(void *buf, char *device)`
 	
Memory allocation and deallocation, respectively. Used to obtain memory buffers for DMA transactions in the case that the SPP places special requirements on the formatting of the memory blocks used for DMA data transfers.

`int pe_put_data_dma_start(void *buf, unsigned int num, unsigned int bank, char *device)`
 	
`int pe_get_data_dma_start(void *buf, unsigned int num, unsigned int bank, char *device)`
 	
Setup and start a DMA data transfer.

`int pe_put_data_dma_status(char *device)`
 	
`int pe_get_data_dma_status(char *device)`
 	
Check the status of an outstanding DMA data transfer.

Continued on Next Page...

Table 3.2 Continued

DLSL Function Description

`int pe_put_data_dma_finish(char *device)`
 	
`int pe_get_data_dma_finish(char *device)`
 	
Complete and finalize an outstanding DMA data transfer.

`int pe_recover(char *device)` Determines if a SPP device is usable. Called by the RTRM for error recovery.
`int pe_can_do(char *kid, char *device)` Indicates whether or not the SPP device can run the kernel requested by `kid`. Call by the RTRM.
`int pe_get_spp_info(spp_info_struct **pointer, char *device)` Returns a SPP dependent structure containing information about the SPP device characteristics. Can be called by both the GHO (from inside a Vforce user application) and the RTRM.
`int pe_get_kernel_info(kernel_info_struct **pointer, char *device)` Returns a struct that describes the characteristics of the currently programmed kernel.

The DLSL provides a mechanism for adding a SPP API into the application binary at runtime, but there must be a method by which the GHO is told which, if any, SPP to use and what DLSL corresponds to that SPP. This task is assigned to the RTRM. The RTRM is a system-wide daemon that waits for hardware requests coming from a GHO on behalf of a user program. The manager tracks what SPP hardware, if any, is available in the system and determines if a particular SPP is capable of performing the functionality required by user application before responding to GHO requests. If the RTRM is able to find a SPP to carry out the desired functionality, it programs the SPP with the requested kernel, if necessary, and replies to the GHO with a path to the SPP device to use and a path to the corresponding DLSL. If no appropriate

hardware was found the manager indicates this to the user application, and the Vforce processing object transfers execution to the software implementation. The manager also keep track of what kernels are programmed on which SPPs, and will use this knowledge to minimize the impact of device configuration by reusing programmed SPPs whenever possible. The end effects of this arrangement are that the DLSL provides SPP API abstraction and the RTRM provides a hardware abstraction layer, completely removing SPP dependence from the compiled binary.

The current RTRM implementation uses only standard POSIX API calls, making the source directly portable to many platforms. The manager reads in a plain text file that describes the system the manager is run on. It contains two pieces of information per SPP installed on the system: the paths to the SPP device and the corresponding DLSL needed to control the device. The manager also loads the DLSL corresponding to a given SPP in order to control it, but it uses a fewer number of SPP control functions than the GHO. When searching for a SPP to execute a kernel requested by a Vforce user application, the RTRM calls the `pe_can_do` function to query the SPP whether or not it has an implementation of that kernel available. This mechanism was chosen as it allows device vendors to produce a SPP DLSL the opportunity to maintain the library of SPP kernels in any fashion that they prefer and prevents any platform specific SPP code from being required by the manager.

All of the communication between the GHO and the RTRM is handled by UNIX System V inter-process communication (IPC) message queues, part of the POSIX

standard. The RTRM creates a message queue with a predefined identification token that is open to requests from Vforce user applications. Two basic types of messages are defined: one for messages from the user application to the manager and one for messages from the manager to the user application. The messages sent from user applications to the manager can be used to indicate one of three things: 1. `SPP request`, 2. `SPP surrender`, or 3. `SPP error`. The first user-application-to-manager message type is used after a a Vforce processing object makes a call to the GHO's `kernel_init` method. The GHO sends a `SPP request` message containing the name of the desired kernel and the identification token of the message queue where the RTRM should send replies. The reply message queue is created by the GHO before sending the `SPP request` message. Upon receipt of the `SPP request` message, the RTRM begins searching for a SPP to run the kernel by calling the `pe_can_do` function inside each SPP's corresponding DLSL until a capable device is found, or until all installed SPPs cannot run the requested kernel. The manager-to-user-application message type can indicate either of these cases. If the manager reply is negative, the GHO will throw an exception and the Vforce processing object will move to the software implementation. In the other case, the RTRM will program the device, if necessary, and send a reply message containing the location of the SPP and the location of that SPP's corresponding DLSL. The Vforce user application then uses the DLSL to directly control the SPP keeping the manager out of the picture preventing it from becoming a system-wide bottleneck.

Once the user application is done using the SPP, it returns the device to the manager with either the `SPP surrender` or `SPP error` messages. Both messages indicate that the user application is finished using the device, but the GHO will send the `SPP error` message whenever it encounters any unexpected behavior from the SPP. In this case, the manager will then call the SPP's `pe_recover` function in the DLSL. This function may include any type of diagnostics and/or repair functionality desired by the DLSL implementer and ultimately indicates to the RTRM whether or not the device should be added back to the pool of SPP hardware available for use by user applications. Currently there is no reply to a `SPP surrender` or `SPP error` message, allowing the user application to continue as soon as it sends the message.

Since the Vforce framework includes a runtime portion not compiled into the user application, it is possible to add additional services to those already provided by the existing RTRM, DLSLs, and kernel libraries, effectively adding to the user application's capability to take advantage of the target platform without sacrificing user application portability or requiring recompilation. While the current generic manager implementation uses a first-come-first-served and a simple SPP assignment algorithm, more intelligence could be added at this stage. In an environment with heterogeneous processors, the manager could take advantage of the variations in performance characteristics a given algorithm displays on different SPP types. These performance characteristics could be based on latency, throughput, arithmetic precision, or any other desired aspect, or combination thereof. These decisions could be

made on the basis of static kernel profiling information or by runtime monitoring and learning of device performance characteristics. The runtime monitoring performed by the RTRM could also be used to provide such features as load balancing and more advanced fault tolerance mechanisms than those currently provided by the existing framework.

The SPP kernels used by Vforce are part of pre-built library for any given machine and not compiled by Vforce. This allows Vforce to take advantage of the expertise offered by designers with domain specific knowledge of the hardware and potentially better designs for the given algorithms than can be created with most automated design tools. The runtime matching to the pre-made kernels also means that Vforce-enabled program compilation is for software only and does not add signification overhead in the compilation phase. In general, there is a one-to-many mapping of Vforce processing objects-to-SPP kernels, as a single processing object can be used on many platforms and SPP kernels are generally tied to a single machine requiring one for each system where SPP acceleration for the given algorithm is required.

In addition, Vforce imposes minimal requirements on the kernels designed for SPPs, allowing kernel implementers to take advantage or specific piece of hardware's feature set, maximizing performance. As long as the DLSL for the SPP implements the standard interface between the GHO and DLSLs, the functionality can be fabricated in any fashion. The method of implementation must match between the DLSL and the kernel library, but it can be tailored to a specific device. A part of the

abstraction level offered by the Vforce API is the concept of memory banks. A given algorithm only requests data to be sent to or received from a specific bank number. An FFT algorithm, for example, might send data to bank 0 and get results from bank 1. While these banks are specified by the Vforce processing object, they can be mapped either statically or dynamically to a specific hardware memory address by whatever mechanism the DLSL implementer chooses. The same mechanism is used for the `put_const` and `get_const` methods used for configuration. A bank number is converted to a register address.

A Vforce processing object uses a series of GHO API commands to setup, configure, and use the SPPs. The SPP kernels must be designed to accept this series of commands. While this may not be a problem in most cases as the order of interactions are rather straightforward, this could be limiting for certain types of hardware or algorithms. For example, the ability of a hardware implementation to use double buffering or not may change how a program would interact with the hardware. Vforce currently includes a mechanism, provided by the `pe_get_spp_info` and `pe_get_kernel_info` DLSL functions, to return information on both the SPP and the currently programmed kernel, respectively, and can be used to tailor the behavior of the Vforce application. For example, the Cray XD1 DLSL checks for DMA capability in the current bitstream before beginning the first data transfer. If DMA is not supported the DLSL will resort to using a series of register reads or writes. This information can also be accessed at a higher level by the Vforce processing object

where other details, like the aforementioned double buffering, might affect how the processing object chooses to interact with the hardware. This additional feature can add complexity to the implementation of a processing object of a Vforce processing object and increases the size of the compiled program as all possible ways of interacting with the hardware must be compiled into the program as it is not known which to use until runtime, but it also allows for important functionality, such as determining how to pack data for data transfers based on the data type and bus width, for example.

Together the GHO, DLSLs and RTRM completely abstract the details of the target system from the Vforce user application. This not only provides source code portability but also binary portability across platforms that share the same application binary interface (ABI). Figure 3.2 shows the relationship between the various Vforce components as well as the flow of control and data transfer. It is important to note again that the RTRM is not involved in data transfer or SPP control after handing the Vforce user application control of the SPP. All data transfer occurs directly between the user application and the SPP device in the same way a program written for a system's native API would. Once the DLSL has been loaded by the GHO the object code required to control the SPP is in the user application's memory space equivalent to when the API code is compiled into the program. The manager is only involved before and after the user application is working with a SPP.

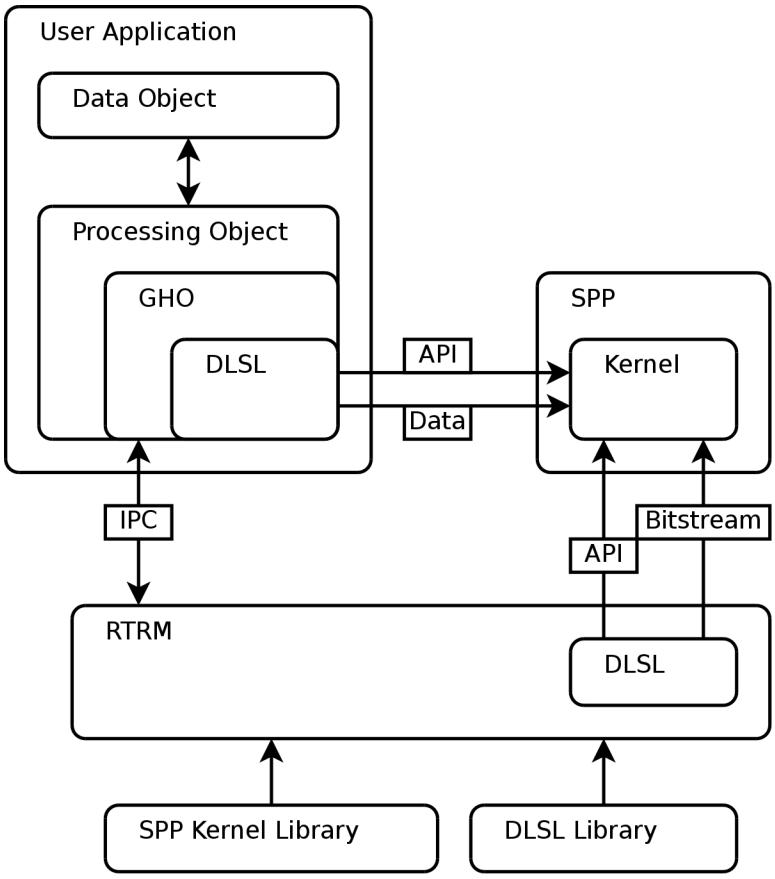

Figure 3.2: Graphical representation of the components in Vforce, their hierarchical relationship, and the types of communication between each.

3.2.3 Extending Vforce

The Vforce framework was designed with modular components in order to make Vforce as portable and extensible as possible, and there are several dimensions along which Vforce can be expanded. First, adding support for new algorithms on an already supported platform requires creating a new Vforce processing object and a SPP kernel for the given algorithm/platform combination. Creating only the Vforce processing object will work, but without a SPP kernel the processing will take place in

software. Additionally, once the Vforce processing object for the new algorithm has been created, the program is still portable, even if other target platforms lack SPPs or the specific SPP kernel since there will always be a software failsafe. Second, adding SPP execution support to a new SPP type for a platform that is already supported by Vforce requires creating the DLSL to control the new SPP type and filling out the SPPs kernel library for the algorithms of interest. Third, to add support for Vforce to a completely new platform a number of components will need to be created. It should again be noted that the application itself will still be portable without any platform specific support as there will be a software only version included in any application that uses Vforce processing objects. If the GHO cannot find the RTRM message queue it will default to software execution. Depending on the specifics of the target platform, a manager may or may not have to be created. The existing manager should compile on most platforms that support POSIX IPC and dynamic linking and provide a base level of manager support. If the target system does not have these capabilities or if more advanced features than those provided by the standard manager are desired, a new manager will have to be created. Next, like the second case for adding new support for a new type of SPP, the creation of a new DLSL and kernels for the desired algorithms may be necessary.

3.2.4 Performance Considerations for Vforce

The Vforce framework offers several benefits, detailed in the preceding sections, including seamless SPP support for existing applications, hiding SPP details from end

users, broad portability, extensibility, and task concurrency. Despite these gains, there are some non-ideal aspects of the framework that must be considered. First, Vforce introduces overhead. Vforce adds a couple layers of indirection in the software stack, uses IPC, loads shared libraries during runtime, and may introduce SPP programming time into what was previously a software only program. In order to help the overall performance of the framework, much emphasis has been placed on minimizing overhead. IPC only occurs before and after the actual use of the SPP device, never during, and this IPC consists of three messages in total. The RTRM remembers and reuses previously programmed SPPs whenever possible to minimize the number of times FPGA programming time has to be incurred. Loading shared libraries on demand is not a zero-time operation, but loading a single library for an SPP incurs a negligible delay compared to the runtime of most useful programs. Dynamically loading libraries is a standard practice today even at the operating system level, where device drivers are loaded and unloaded without restarting the operating system. In addition, the symbols within the shared libraries, namely the functions that the GHO searches for, are resolved on demand and only if used by the user application. Afterwards, the locations of these functions is cached so that the search only happens once.

In the end, these factors have relatively little impact on the performance of applications using Vforce. What is more consequential is the fact that Vforce has been built on top of VSIPL++. This choice was made to make Vforce portable to any

conforming VSIPL++ implementation, but it forces Vforce to respect the same rules about admitted and released data blocks that VSIPL++ users have to follow. Since there is no way to get direct access to the underlying data in a VSIPL++ view that created its own storage, Vforce treats VSIPL++ views as opaque objects. When transferring data to or from a SPP, Vforce copies values one at a time between the memory buffer used for data transfer and the VSIPL++ data block using the regular VSIPL++ view accessors. This data copying introduces non-negligible overhead into the Vforce framework, and excluding programming time for FPGAs, is the dominate source of overhead. The effects of the data copying can be seen in the Vforce FFT application results in Section 5.1. Chapter 6.2 describes a mechanism by which this data copy may be avoided in future versions of the framework. In addition, efforts have made throughout the framework to eliminate any excess data copying. For example, the GHO and existing Vforce processing objects are heavily templated so that they are guaranteed to meet VSIPL++'s by reference semantics eliminating data copy in method and function invocations. The DLSL interface also includes a function that provides the GHO with a memory buffer that is guaranteed to meet the requirements needed to be used for DMA data transfer transactions. This prevents the DLSL from having to make an additional internal copy into a DMAable buffer, as would be the case on the Cray XD1 which requires DMA transactions to use memory buffers that are page aligned and an integer multiple of the systems page size in size.

3.3 Summary

This chapter discussed the Vforce framework in detail. Vforce adds transparent special purpose processor support to VSIPL++ programs, allowing them to utilize existing SPP function kernels. In addition, Vforce maintains portability across systems with and without SPPs and different types of SPPs. This portability is enabled through the use of several runtime components including an independent system wide resource manager. The next chapter presents two case studies used to examine the characteristics of the Vforce framework: an FFT and a beamformer.

Chapter 4

Vforce Case Studies

This chapter presents two different case studies that use the Vforce framework in different ways: an FFT processing object and a beamforming application. Both of these studies served multiple purposes including examining the characteristics of the Vforce framework, investigating practical implementation issues, and showing the validity of the approach the Vforce framework uses to add SPP functionality to Vforce while maintaining performance and portability. Both the FFT and the beamforming application were implemented on the Cray XD1, and this chapter begins with discussion of implementation of the Vforce framework infrastructure on the XD1 before moving on to the implementations of the two applications.

4.1 Cray XD1

The Cray XD1 [3] is a supercomputer introduced in 2004 and includes both x86-64 GPPs as well as FPGA SPPs. A single XD1 chassis encloses 12 AMD Opteron processors and 6 Xilinx Virtex family FPGAs, and up to 12 chassis can be installed in a cabinet [4]. The XD1 supports both single and dual core processors paired into

groups of six 2 or 4-way SMP nodes. Each pair of processors share a Xilinx Virtex-II Pro or Virtex-4 FPGA and are directly connected to the processors' HyperTransport bus via Cray's RapidArray interconnect, which provides high-bandwidth low-latency direct point-to-point connections between processors as well as a mechanism for direct memory access to the CPUs' RAM by the FPGA. Each node (GPP pair) on the XD1 runs an instance of the Linux operating system and includes system libraries modified to use the RapidArray interconnect for inter-processor communication and memory accesses offering up to 8 GB/s bandwidth (with a 1.7 microsecond MPI message latency) between nodes. Figure 4.1 shows a graphical depiction of a single node within the Cray XD1.

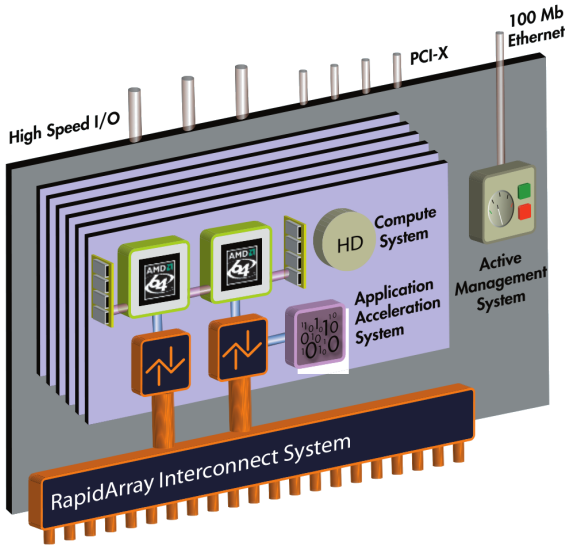

Figure 4.1: A single Cray XD1 node [4]

All of the development done on this project was on a Cray XD1 installed at the Ohio Supercomputing Center in Springfield, Ohio [16]. The particular XD1 system

used for these case studies has Opteron 248 processors, Virtex-II Pro 50-7 FPGAs, and is running Cray software release 1.3 (build 1005), which includes GCC 3.3.3. The Opteron 248 processor is clocked at 2.2 GHz and has 1 MB of L2 cache. Each node of the system has 4 GB of RAM installed, and each node's FPGA is paired with 16 MB of local second generation Quad Data Rate (QDR-II) SRAM, which is arranged into four 64-bit wide 4 megabyte banks. Like double data rate (DDR) SDRAM, QRD-II SRAM transfers data twice per cycle: on the rising and falling edges. However, QDR-II SRAM uses two independent clocks to achieve an effective total of four transfers per clock cycle [18].

4.1.1 DLSL, RTRM, & Kernel Libraries

As mentioned in chapter 3, a DLSL and RTRM have been developed for the Cray XD1. The XD1 currently uses the generic RTRM which compiles easily on the XD1 as it is has available a Linux based operating system and the GCC toolchain. The XD1's DLSL implements all of the features required by the GHO-DLSL interface, as discussed in detail in Section 3.2.2. The XD1 requires that buffers used for DMA memory transfer to and from the FPGA be page aligned and that the sizes of the buffers are integer multiples of the system's page size (using the POSIX functions `getpagesize` and `posix_memalign`). The XD1 DLSL `pe_get_dma_buffer` implementation makes sure these requirements are met. In addition, the DLSL tracks the location and size of these memory buffers to make sure that any attempts on starting a DMA transfer with `pe_put_data_dma_start` or `pe_get_data_dma_start` are called

with buffers that were properly allocated by the DLSL.

Currently, Vforce on the XD1 organizes the kernel library based simply on filenames. The DLSL checks an environment variable for the root path and then appends the file name and device type (the XD1 has two possible device types, the Xilinx Virtex-II Pro or Virtex-4) to come up with a unique bitstream filename. If the file exists, the XD1's `pe_can_do` function will return true and false otherwise. This arrangement makes it easy to add new bitstreams to the kernel library. Other than the memory management, bitstream management, and wrapping the rest of the GHO-DLSL API around the native XD1 FPGA API, the XD1 DLSL makes use of gcc's `__attribute__ ((constructor))` and `__attribute__ ((destructor))` function attributes to create library constructor and destructor functions that perform some initialization, cleanup, and bookeeping to ensure that the DLSL can be loaded and unloaded cleanly. Because of the small size of the existing kernel library for the XD1, the `kernel_info_struct` structure is set to a static set of values that are shared across all of the existing FPGA bitstreams. Finally, the current release of the XD1 does not allow a programmed FPGA to be closed and reopened in a non-destructive manner. Since the RTRM and Vforce user application are separate processes, only one is allowed to open an FPGA device at a time, and as a result of this, the XD1 cannot support manager-side programming of the FPGA. This scenario is handled by the Vforce framework through a flag in the `spp_info_struct` structure which indicates whether or not the device supports programming by the RTRM. Note that

this does not allow the the XD1 to make use of efforts by the RTRM to minimize the number of FPGA configurations incurred, as discussed in Section 3.2.4. The user application has to program the FPGA every time a new SPP device is requested, possibly adding overhead.

4.2 FFT

The first case study using the Vforce framework involves an FFT Vforce processing object that replaces the existing VSIPL++ FFT processing object. In addition to being a drop-in replacement for the existing VSIPL++ FFT that allows backwards compatibility and portability on platforms with and without SPP hardware, the FFT Vforce processing object is an example of a processing object that extends VSIPL++'s capabilities by exposing a level of task parallelism to the user application.

4.2.1 Vforce Processing Object

The FFT Vforce processing object is completely compatible with the VSIPL++ specification for the FFT, except that it only performs one-dimensional FFTs. The class respects all of the template arguments specified for the VSIPL++ FFT including whether argument passing is by reference or by value. In addition to the VSIPL++ specified interface for the FFT, the Vforce processing object replacement adds `start`, `status`, and `finish` methods to the FFT. These, in addition to the `operator()` method, are cumulatively referred to as action methods. The `start` and `status` methods are non-blocking and begin the FFT computation and check

whether the computation has completed, respectively. After the FFT computation has been started, a call to `finish` will block until the FFT has completed. Table 4.1 gives more details about the class' additional action methods.

Table 4.1: The Vforce FFT processing object additional action methods API

Method Description
`template<typename Block, template<typename,typename>class InView>void start(InView<InputType, Block>InputView) throw()` Non-blocking; Starts the FFT computation. If the GHO hasn't already been assigned an SPP device, `kernel_init` on the GHO will be called.
`bool status(void) throw()` Non-blocking; Checks if the FFT computation has completed and returns true if it has.
By value: `template<typename BlockIn, template<typename, typename>class ViewIn>ViewIn<OutputType, vsip::Dense<DimOfView<const_View>::dim, OutputType>>finish(ViewIn<InputType,BlockIn>InputView) throw(std::bad_alloc)` By reference: `template<typename BlockOut, template<typename, typename>class ViewOut, typename BlockIn, template<typename, typename>class ViewIn >ViewOut<OutputType, BlockOut>finish(ViewOut<OutputType, BlockOut>OutputView, ViewIn<InputType, BlockIn>InputView) throw()` Blocking; Waits for computation to finish and returns the results. Does not call `kernel_dest`.

Upon instantiation, the FFT instantiates a GHO, but it does not invoke `kernel_init` on the GHO until the `start` method is invoked. Also, once the FFT computation is completed with a call to `finish`, the Vforce FFT processing object does not call `kernel_dest`, but instead holds on to the programmed SPP hardware until the FFT object is destroyed. This behavior was chosen because of the XD1's limitation on manager-side configuration of the FPGA and prevents incurring the cost of multiple

configuration times when the FFT object is used multiple times. On a system that supports manager-side SPP configuration, this greedy behavior could be dropped, if desired, as the RTRM will reassign the already programmed FFT without incurring the reconfiguration times. The `kernel_dest` method is called when the Vforce FFT processing object is destroyed.

If at any time the GHO indicates an SPP error, the Vforce FFT processing object will use a VSIPL++ FFT processing object as the failsafe. After an SPP error, the `start` and `status` methods return immediately, with the `status` method always returning true, indicating to the user program that the FFT is done. The VSIPL++ FFT is used on the input data once the `finish` method is called and is not instantiated until it is needed the first time. This way the VSIPL++ FFT instantiation overhead, which can be non-negligible depending on the underlying FFT library, is only incurred if the VSIPL++ FFT is needed. Regardless of whether the Vforce FFT processing object is working by value or by reference, internally the VSIPL++ FFT failsafe is by reference to reduce internal data copying.

4.2.2 Cray XD1 FPGA Kernel

To match the Vforce FFT processing object, an FFT bitstream was created for the XD1's FPGAs. The bitstream uses an FFT core generated with the Xilinx CORE Generator [30]. The core was configured to use a 24-bit fixed point data type and perform FFTs ranging form 64 to 32,768 points in size. The data coming from the user application running on the GPP is floating point, so the Northeastern Re-

configurable Computing Laboratory's Variable Precision Floating Point Library [24] fixed-to-floating point and floating-to-fixed point modules are used to convert between the data formats. As the floating point data is transferred by DMA to the FPGA module's QDR-II SRAM, the range of the incoming data is monitored, as shown graphically in Figure 4.2, and an appropriate scaling factor is chosen to maximize the precision of the fixed point format. When the FFT is started, the data is streamed through the floating-to-fixed-point conversion module and into the FFT core. The output is then streamed through the fixed-to-floating-point conversion module and multiplied by the final scaling value (a functional part of the VSIPL++ FFT) in floating point before it is stored in Bank 1. The floating point multiplication is handled by another CORE Generator component. The bitstream makes use of IP from Cray for some tasks common to many bitstream designs, including the register file design, DMA engines, QDR-II SRAM cores, and HyperTransport bus communication core, and the entire design was synthesized, placed, and routed by Xilinx ISE 8.2i.

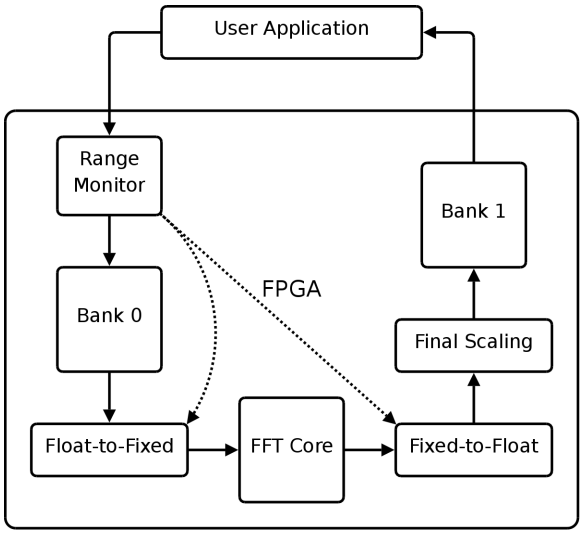

Figure 4.2: Graphical representation of the FFT application

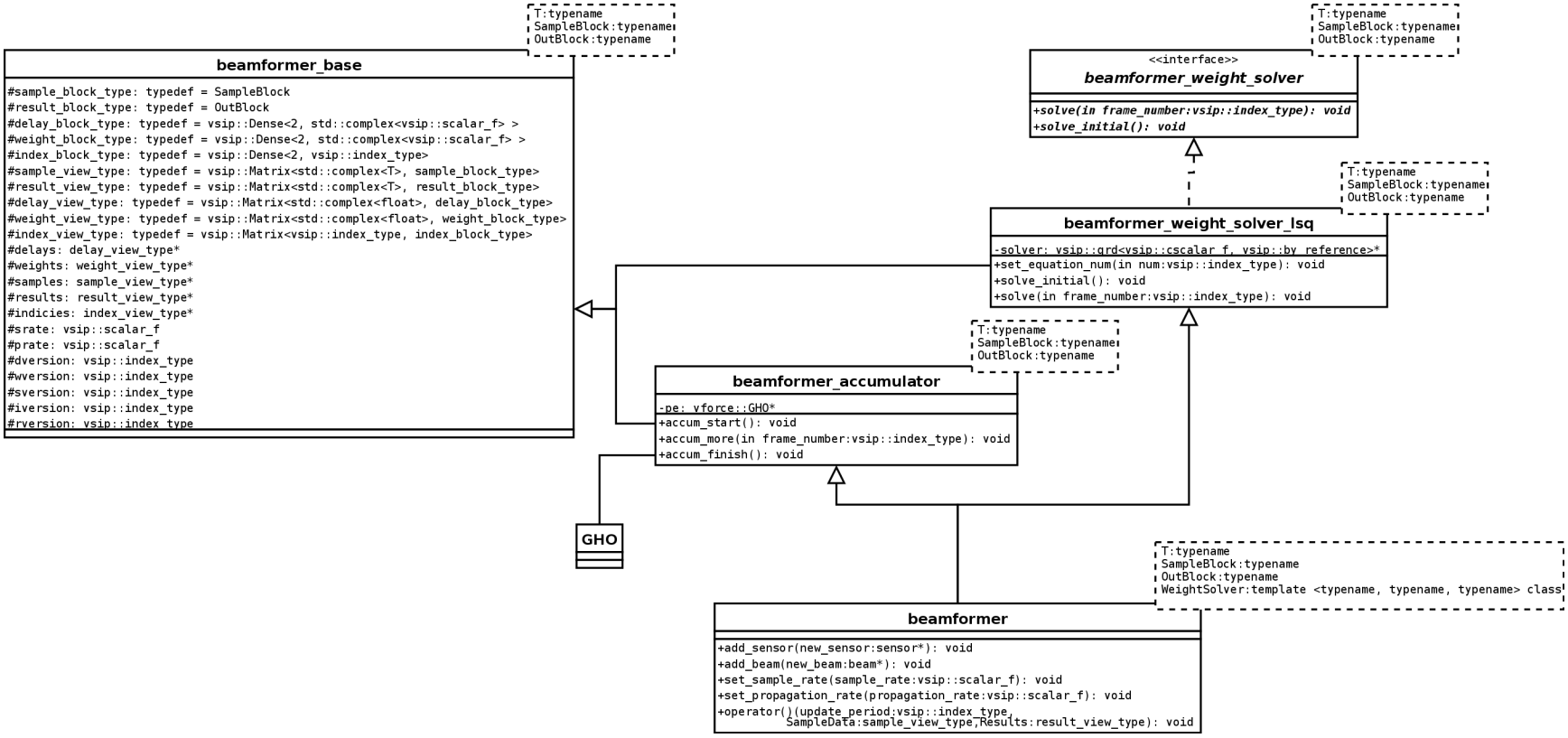

Figure 4.3: UML Diagram for the beamformer Vforce processing object

4.3 Beamforming

The second case study using the Vforce framework is a beamforming processing object. Beamforming was chosen because of the inherent parallelism between the weight application and weight computation sub-tasks, as discussed in Section 2.2.1. In addition, the weight application sub-task is a good fit for many SPPs as the control is minimal while the computation is dense. As opposed to the FFT Vforce processing object, the beamformer implementation does not expose any task parallelism to the user. Instead, the processing object manages the two beamforming sub-tasks internally. Another difference from from the FFT Vforce processing object is that the beamformer is an example of processing object whose scope is larger than any existing functionality in the VSIPL++ specification, and it relies on multiple VSIPL++ processing objects and other VSIPL++ functionality to implement both of the beamforming sub-tasks.

4.3.1 Vforce Processing Object

The weight update and weight application processes are independent, and the design of the beamforming Vforce processing object reflects that. As shown in Figure 4.3, the beamforming processing object breaks the beamformer into a number of discrete components. The `beamformer` class is the user facing portion of the beamformer and includes methods to configure the beamformer by adding beam and sensor locations and setting sample and propagation rate parameters. Currently, sensor locations are

specified by a set of Cartesian coordinates and incoming beams are specified by a look direction and frequency. The `beamformer` class also includes the `operator()` method which performs the actual beamforming for the user. It should be noted that the beamformer maintains state between calls to the `operator()` method, allowing continual processing of more data by successive calls to `operator()`.

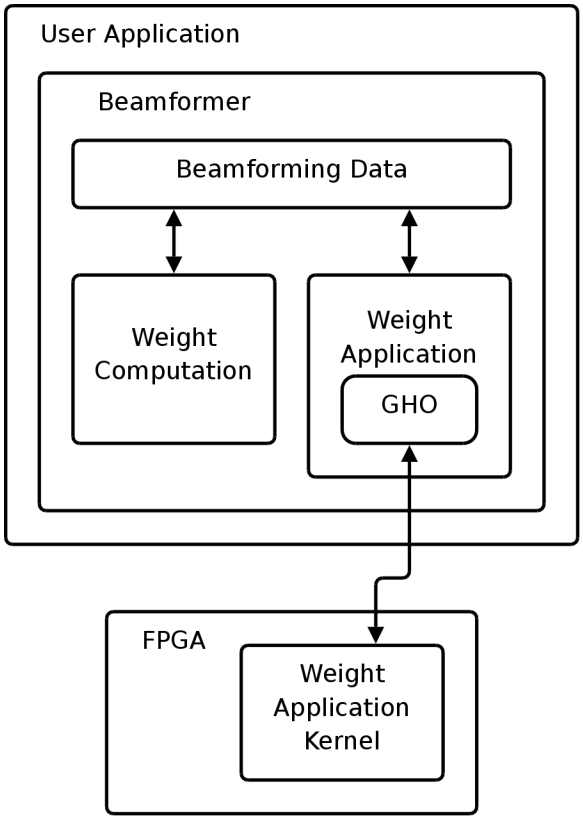

Figure 4.4: A high level representation of the beamforming application

The `beamformer` class inherits from two distinct classes that separately provide the functionality needed for weight computation or weight application. As can be seen in Figure 4.3, both of these two component classes inherit from the same base class, `beamformer_base`, forming a diamond inheritance pattern. This base class

contains no functionality and acts as a container for all of the data needed by the beamformer during processing. Not only do the two sub-task classes inherit from the same class type, but through the use of the C++ `virtual` inheritance specifier, the two classes actual share the same base class including the data. This structure allows each component, including the `beamformer` class, to access the underlying data as if it were native to the component itself and simplifies and reduces the overhead in the interface between all of the classes involved in the beamformer processing object. The `beamformer_base` class also contains the counter variables `dversion`, `iversion`, `rversion`, `sversion`, and `wversion`, corresponding to the beamformer time delays, indicies, results, sensor data, and weights, respectively. When one of the data structures used by the beamformer is modified, the corresponding counter is incremented. These counters are usually unimportant when running only in software as all operations will refer to the original data structures. However, when a portion of the beamformer is implemented in hardware the counters can be used for update synchronization between the beamformer data structures and the copies of those data structure in hardware. The necessary data is only sent if a newer version of the data, indicated by a new value for that data structure's counter, which can help minimize the amount of data transferred and hopefully increase performance. In addition, the entire beamformer class hierarchy is templated to support using whatever block type is desired by the user, which allows for the minimization of data copying by guaranteeing VSIPL++ by reference semantics. The beamformer is also

parameterized by the base data type to support either integer or floating point based beamforming.

Each of the two sub-tasks involved in beamforming can be thought of as individual processing objects, and the design of each represents that. The `beamformer_weight_solver_lsq` class performs the weight computation sub-task. To find the least squared error solution to the weight computation problem the `beamformer_weight_solver_lsq` class relies on the VSIPL++ `qrd` processing object, which contains a least-squares solver. The `beamformer_weight_solver_lsq` class implements a weight computation interface, as specified by the `beamformer_weight_solver` abstract class, shown in the UML of Figure 4.3. This interface was created to allow changing the weight computation class to others that use different algorithms, adding to the flexibility of the beamforming Vforce processing object. The class to use for the weight computation sub-task is a compile time template parameter for the `beamformer` class, allowing the user to specify the weight computation module to be used. The specified weight computation interface specifies only the methods that the `beamformer` class expects to be present and will call during operation. Other methods may be specified and will be exposed to the user, as the weight computation class is inherited with the `public` inheritance specifier. For example, a Frost LCMV[9] weight computation class was also implemented. It includes a method to specify the constraints and allows the user application to update the constraints and recompute the weights. This allows for changing the look directions of the beamformer during operation, functionality

not made available by the standard `beamformer` and `beamformer_weight_solver` interfaces. In order to make comparisons to the Mercury 6U VME implementation (see Section 4.3.3), only the `beamformer_weight_solver_lsq` weight computation class is considered in this document.

The `beamformer_accumulator` class performs the weight application sub-task. As can be seen in the UML of Figure 4.3, the `beamformer_accumulator` class presents a simple interface, similar to that of the FFT, that allows for task parallelism in conjunction with the weight application task. The `beamformer_accumulator` class is also the only class in the beamformer that relies on functionality from the Vforce framework and allows the weight application sub-task to run on a SPP.

When the user calls the `operator()` method on the the `beamformer` class, the class initializes some common values and then coordinates the activity of the two classes. First the initial weights are computed with a call to `solve_initial`, and then a call `accum_start` sends the first sets of weights, indicies, and sensor data to the SPP (assuming an SPP that can run the weight application kernel is available) and starts the weight application on this data. The `operator()` method takes one argument that specifies the number of samples between weight computation updates. Updating weights more frequently requires more processing power to compute the new weight values at the same rate but results in a beamformer that is quicker to adjust to an optimal set of sensors weights and to changing interference. This argument provides an additional mechanism for tailoring the beamforming Vforce

processing object's behavior to the environment and platform that it is being run on. In the steady state case, the `beamformer` class instance alternates between invoking the weight computation and weight application sub-tasks on frames of data that are the length of the update period. If no SPP is available and the entire application is run on a processor, the program alternates between weight computation and weight application (assuming a serial VSIPL++ environment). However, if a capable SPP is present, the weight application task will be run concurrently with the weight computation task.

The `beamformer_accumulator` class was also designed to take advantage of SPP implementations of the weight application sub-task that present an interface that supports double buffering as well as the capability to stream the results back to the GPP. The `accum_start` method sends the first two frames of data to the SPP for weight application to fill the pipeline. Afterward, calls to `accum_more` only send one additional frame of data. With the double buffering mechanism enabled, the weights can be applied to one frame while the sensor data for the next frame can be sent to the SPP simultaneously. If the SPP supports simultaneous reading and writing DMA transfers, the results can be sent back to the GPP as they are produced, eliminating the need to wait for an additional data transfer after each frame. Since the SPP kernel control and data transfer functions can be non-blocking in Vforce, all of these tasks can be started and then control can be returned to the weight computation running on the GPP. This combination of features allows for significant concurrency

inside the beamforming process and results in the data flow behavior represented in Figure 4.5, assuming that new sensor indicies and sensor weights are transferred after each iteration. (Note that the concurrency improves if the new set of weight data is not sent after each iteration.) A single color represents operations being performed with the same frame of sensor data. It takes three update periods for new data to affect the weights being applied to incoming sensor data, which is represented by the dashed arrow. The double buffering and concurrency adds latency to the number of frames it takes for weights reflecting changes in the interference to be applied to sensor data by adding to the effective pipeline length. However, streaming the results back and concurrency help to mitigate these effects. Streaming the results back removes an extra step (retrieving the results), and the concurrency also allows the beamformer to get more peak performance out of the target platform which may allow for the use of smaller update periods.

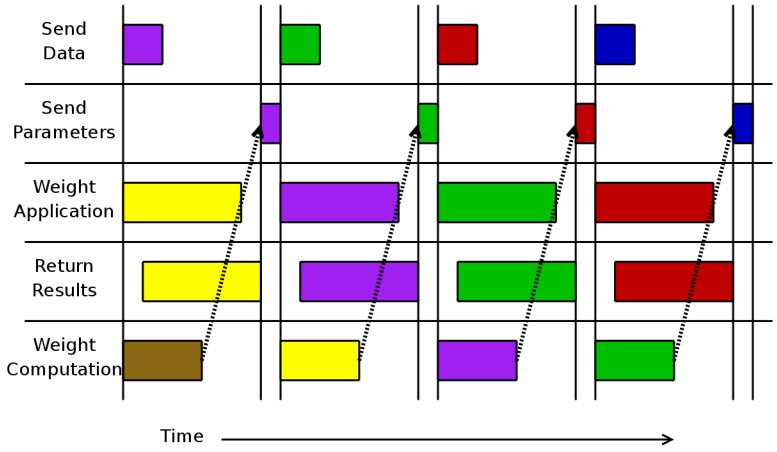

Figure 4.5: A graphical depiction of data flowing through the beamformer's weight application pipeline

The pipeline diagram in Figure 4.5 shows that sending new weights and indicies does not overlap with any other activity in the beamforming pipeline. Currently there is no way to combine multiple data transfers of similar or dissimilar types, preventing one data transfer operation from sending new data, new weights, and new indicies all at once. These must be transferred through multiple discrete DMA transactions, so the longest of them, the data transfer is used for overlapping with the weight application and weight computation sub-tasks.

4.3.2 Cray XD1 FPGA Kernel

To observe the characteristics of the beamforming Vforce processing object, a beamforming weight application bitstream was created for the target XD1's FPGAs. The behavior of the bitstream is dictated by the corresponding Vforce processing object and was described in the preceding section; the details of the bitstream implementation are documented in this section. Like the XD1 FFT bitstream, the XD1 weight application bitstream relies on the communication, register, QDR-II SRAM, and DMA infrastructure provided by Cray, but was synthesized, placed, and routed by Xilinx ISE 9.1i.

As described in Section 2.2.1, the mathematical operation performed by the SPP is a complex indexed weighted multiply accumulate. A block diagram of the XD1 FPGA implementation is provided in Figure 4.6. All data operations are performed in single precision floating point, and the multiply and addition units in the complex multiply and adder tree were generated with a 6-cycle latency using Xilinx CORE Generator.

The accumulator portion of the datapath uses the floating-point accumulator design from the Reconfigurable Computing Laboratory's Variable Precision Floating Point Library [29], and was originally based on a design in [31]. The design uses a single adder to save a significant amount of FPGA area at the expense of accumulator latency. The number of cycles from the time the last accumulator input is applied until the final output is available varies from 9 to 35 cycles depending on the number of inputs, as shown by the graph in Figure 4.7. The accumulator also requires that no inputs are applied to the accumulator while the result is being calculated, or another accumulation will be triggered. This requires a variable length stall in the pipeline while the accumulator completes processing the final outputs. Finally, in addition to these delays, the QDR-II SRAM has a 6-cycle read latency. To compensate for all of these delays while keeping the datapath full requires a pipeline of non-negligible length. Note that while there is significant room for parallelization of the weight application sub-task, the bandwidth of each SRAM bank is limited to 64-bits per cycle. The implementation doesn't use the full bandwidth available as indicies are only 32-bits, but otherwise the design well utilizes the memory hierarchy.

The data is originally transferred to the board using the read DMA engine in the order shown in Figure 4.5. Figure 4.6 shows that the incoming data ends up in one of four logical banks, each of which is mapped onto one of the 4 megabyte SRAM banks. Two banks are used for the incoming sensor data in order to provide double buffering and one each for the sensor indicies and sensor weights. The index data

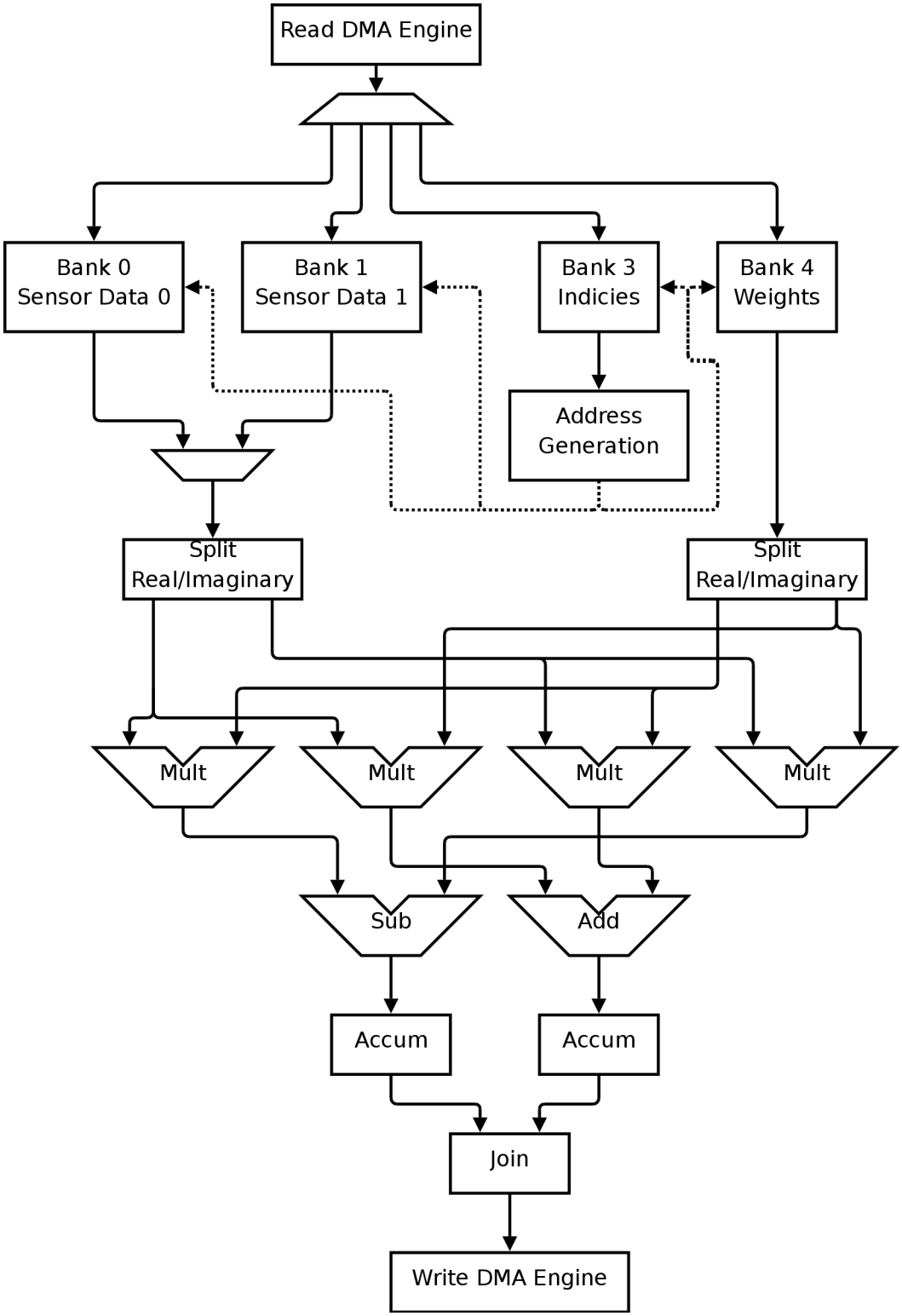

Figure 4.6: Graphical representation of the beamforming FPGA bitstream

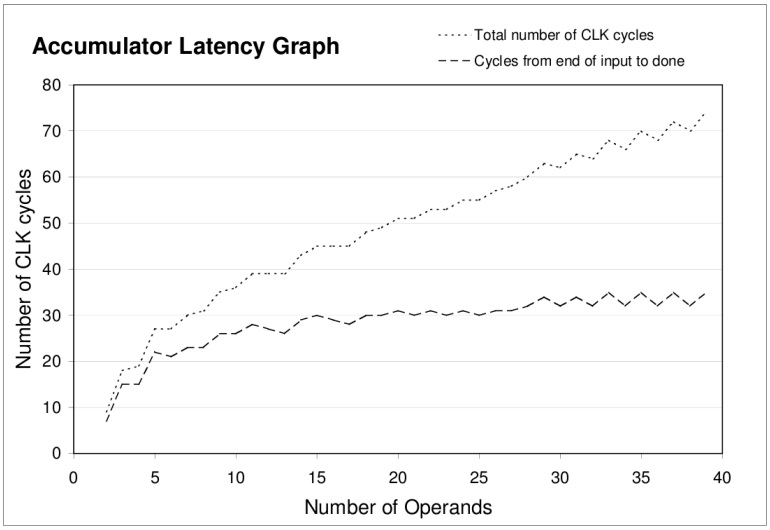

Figure 4.7: A plot of the latency of the accumulator, in cycles, versus the number of input operands

includes one 32-bit signed integer for each sensor for each beam; there is also one complex single precision weight for each sensor for each beam. For the total number of the weights and indicies there are $number\ of\ sensors \times number\ of\ beams$ values, but the 64-bits for the sensor weights take up twice the space. With 4 megabytes of storage, up to 2^{19} weights can be stored on the FPGA at once. The bitstream was designed to be as flexible as possible, and as such is only limited by the size of the memory banks. Any combination of number of sensors and beams can be used as long as their product is less than 2^{19}.

The limited size of the FPGA memory limits the number of time steps that can be processed with each new sensor data transfer to the board. A complex single precision sensor value is generated for each sensor for each time step, so for this design the maximum number of time steps of sensor data in a RAM bank at once is

$\frac{4\ megabytes}{8\ bytes\ per\ sensor\ reading \times number\ of\ sensors}$, resulting in a maximum of 2^{19} time steps for one sensor and one time step for 2^{19} sensors, at the extremes. However, this limitation does not have a significant effect on the range of allowable update periods. Less than a full bank's worth of sensor data can be transferred if small update periods are used. On the other hand, if the selected update period is large enough to result in the sensor data size exceeding the size of the sensor data bank, multiple transfers of sensor data and weight application steps can be performed without transferring updated weights or indicies until all of the update periods time steps have been processed. For tractability of testing, however, sending weights and indicies after each iteration was the only scenario tested, which is the worst case performance scenario and also corresponds to a reasonable range of weight update periods when using a realistic number of sensors.

The control logic in the bitstream manages this wide variety of configurations as well as the long latencies. First, for a single time step for a single beam, the corresponding index and weight addresses are read from the index and weight RAM banks. After the 6 cycle read latency the indicies are used to generate a sensor data read memory address from the current time step, beam, sensor, and index offset. After another 6-cycle memory read latency, the sensor value and the corresponding weight are fed into the multiply-accumulator datapath. After all of the sensor values for that time step and beam have been fed into the pipeline, the control logic stalls until the output value is produced by the accumulator. The result is fed into the write

DMA engine which streams the results back to the user application on the GPP. Once the result is produced, the design moves on to the next time step or beam. Because the accumulator's final output is not immediately available, a stall must be incurred for each time step for each beam, which limits the maximum utilization of the FPGA datapath.

There does not need to be extra delay before switching between banks of sensor data, however. The `beamformer_accumulator` processing object uses a `put_const` to set a flag in a register on the FPGA that indicates that new data is ready in the other sensor data bank and processing can begin whenever the FPGA finishes its previous bank of sensor data. A second register indicates which bank the FPGA is currently processing. This simple mechanism provides the synchronization necessary between the software and hardware components of the application. If new sensor data has been sent to the SPP before the SPP is done processing the previous frame, the flag can still be set and the FPGA will continue as soon as it finishes the previous bank. If more sensor data is ready, the `beamformer_accumulator` processing object will not start another data transfer to wipe out sensor data in the middle of being processed as it waits for the FPGA to indicate it is done by setting the second register flag, which is read with `get_const`. Finally, note that the index and weight data is not double buffered. This forces the FPGA to pause while new index and sensor data is being transferred to the board, but due to the restrictions surrounding sending multiple data types in one transaction, this pause would be necessary anyway.

4.3.3 Mercury Implementation

An implementation of the same beamforming algorithm was done one a machine from Mercury Computer Systems [13], using an earlier version of the Vforce framework. The first implementation targeted a Mercury 6U VME system [19], splitting the weight computation and weight application between the PowerPC GPP and the Virtex-II Pro SPP, respectively. The version of Vforce used for the Mercury 6U VME implementation did not support non-blocking data transfers, which prevented the implementation from taking advantage of a lot of the available concurrency in the application. The new Vforce implementation for the XD1 takes advantage of new features. More details about the implementation and performance characteristics are available in [2], and details are only included or referenced here where needed for purposes of comparison.

4.4 Summary

This chapter covered implementation specific details for Vforce on the Cray XD1, as well as two processing objects created with Vforce and corresponding XD1 FPGA bitstreams. The first, an FFT, is a drop-in replacement for the existing VSIPL++ FFT processing object. The second processing object is a time-domain block adaptive beamformer. The beamforming processing object has a large granularity and breaks the application into two components: weight computation and weight application. Weight computation is always performed in software, while weight computation runs

either in software or on an SPP. The beamformer is also designed to take significant advantage of GPP and SPP processing simultaneously. The next chapter describes benchmarking procedures used to test the two processing objects and presents an analysis of the results.

Chapter 5

Experiments & Results

This chapter covers the experiments that were run using the FFT and beamforming Vforce processing objects. The FFT testing focused on examining the overhead introduced by Vforce, while the beamforming results focus more on task concurrency and performance relative to the previously existing Mercury implementation [2].

5.1 FFT

One of the features of Vforce is that it allows a program to automatically utilize SPP hardware for computation when possible; otherwise it performs the same computation on a GPP. To test the use of Vforce, two scenarios were examined:

1. Vforce hardware versus native API hardware: The Vforce FFT uses one of the system's SPP to run the FFT and is compared to a program written for the specific system that uses the custom API for that system's SPP. This scenario examines the overheads that Vforce introduces when executing kernels on SPPs. The same SPP kernel is used in both cases, so any differences in performance

between the two are due to overhead that Vforce adds.

2. Vforce software versus VSIPL++ software: The Vforce FFT executes the FFT on the system's GPP and is compared to a VSIPL++ program that implements the same FFT. As discussed, the Vforce FFT processing object is an example where the Vforce processing object has the same functionality and granularity as an existing VSIPL++ processing object, and as a result, the Vforce FFT processing object relies on the VSIPL++ FFT to run the GPP FFT when hardware is not available or fails. Like the hardware scenario, the FFT itself is executed by the same FFT library on the same GPP meaning that differences in performance are due to overhead from Vforce.

The tests were run on a Cray XD1, as discussed in Section 4.1. The native application was written in C and uses the Cray FPGA API specified for the XD1's Application Acceleration Processors. The same Vforce application binary was used in both scenarios, and for both scenarios the manager was running. The hardware runs include the RTRM, IPC, and any of the other overhead incurred from using Vforce. To force the Vforce application to fall back to software execution of the FFT in the second scenario, the XD1 FFT kernel was removed from the bitstream library on the XD1. The software timing includes the overhead added by a request to the RTRM for hardware, a kernel lookup (currently implemented by the XD1 DLSL by checking for the existence of a file with a specific name, see Section 4.1.1), and a negative response from the RTRM to the user application. While the VSIPL++ FFT

program is a separate executable from the Vforce FFT program, the code is identical. The only difference between the two are the libraries included when compiling the program.

All three applications are relatively simple and take command line arguments for the size of the FFT and the number of times the FFT should be executed. For any number of iterations, the FFT object is instantiated only once before running the FFT iterations, and each iteration includes the data transfer to and from the FPGA if the FFT is executed in hardware. All the applications also profile the FFT in the same way to provide consistent results. For both scenarios the FFT test scenarios varied in two dimensions: the size of the FFT and the number of iterations. The FFT point sizes tested included 64, 256, 512, 1024, 8192, and 32,768. For each FFT size the number of iterations varied from one to 500 (1, 5, 10, 50, 100, 500) in hardware and one to 10^5 in powers of ten in software. The disparity in hardware and software sizes arose from a bug in the XD1 FPGA API libraries. Between a single pair of calls to `fpga_open` and `fpga_close` there is a limit to the cumulative number of memory buffers that can be registered with the API for DMA data transfer. It makes no difference if the memory blocks are unregistered before registering new blocks. Vforce currently treats all data transfers as independent transactions, so four memory buffers are registered per iteration, one for sending sensor data, one for sending weights, one for sending indicies, and one for receiving results, limiting the number of possible iterations to about 500 in this case.

Starting with the hardware scenario, the FFT performance numbers for the native XD1 application, in MFLOPS, are shown in Figure 5.1. Throughout this section, the number of MFLOPS is calculated using the standard FFTW methodology [6]:

$$\text{MFLOPS} = \frac{5N\log_2(N)}{\text{time for one FFT in microseconds}}$$

where N is the FFT point size. Figure 5.2 shows the same set of benchmarks, but in terms of execution time instead of MFLOPS. The two figures show the performance increases and the average time for an FFT drops as the number of iterations increases. The programming set-up time gets amortized over an increasing number of runs. Note that the execution times for all sizes are nearly indistinguishable as the programming time and cumulative latency of Cray XD1 FPGA API calls dominate the data transfer and processing times, despite the varying amount of work that is performed, which is reflected in the different curves for MFLOPS.

To compare the performance of the native hardware implementation with the Vforce version, a graph of the speed-up for the Vforce implementation is shown in Figure 5.3. The figure shows that the speedup of the Vforce version decreases as the number of iterations and the size of the FFT increase. This difference is the result of the data copying during the DMA data transfer discussed in Section 3.2.4. As would be expected, the relative performance decreases as the amount of data to copy increases. It can be seen however, that in cases where less data is copied, the speedup is near unity, demonstrating that Vforce imposes minimal overhead other than the data copy when using hardware.

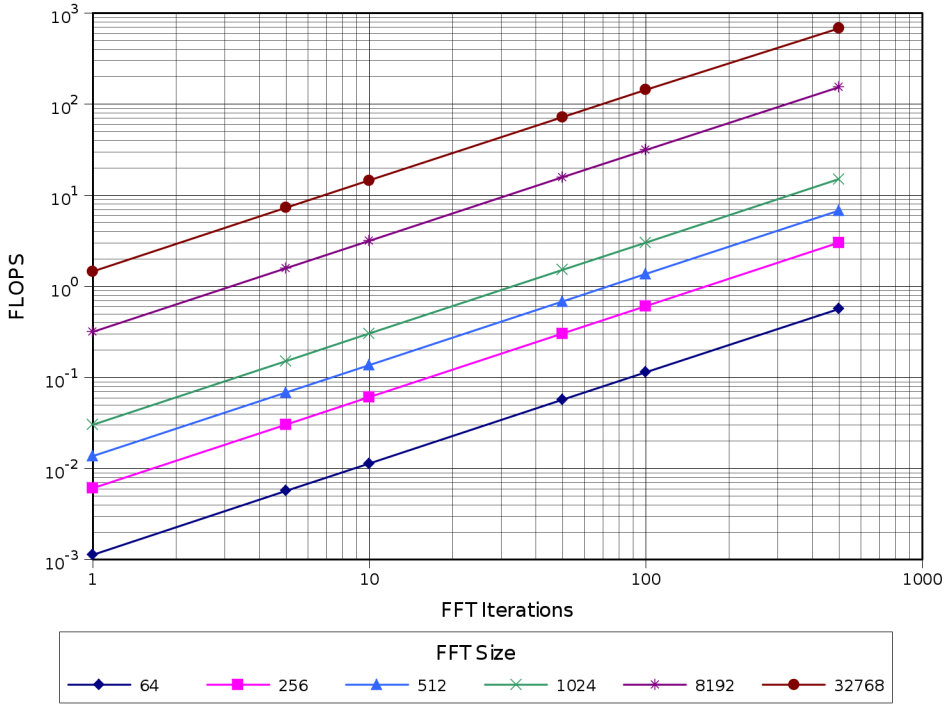

Figure 5.1: Native API hardware FFT performance in FLOPS versus the number of FFT iterations

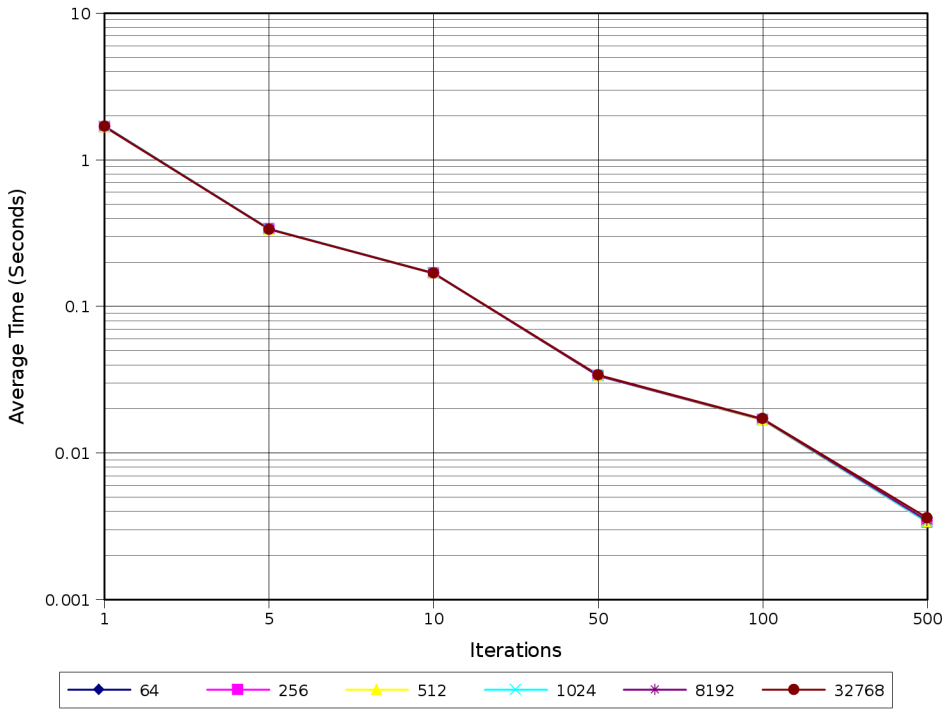

Figure 5.2: Native API hardware FFT execution times in average time per iteration versus the number of iterations

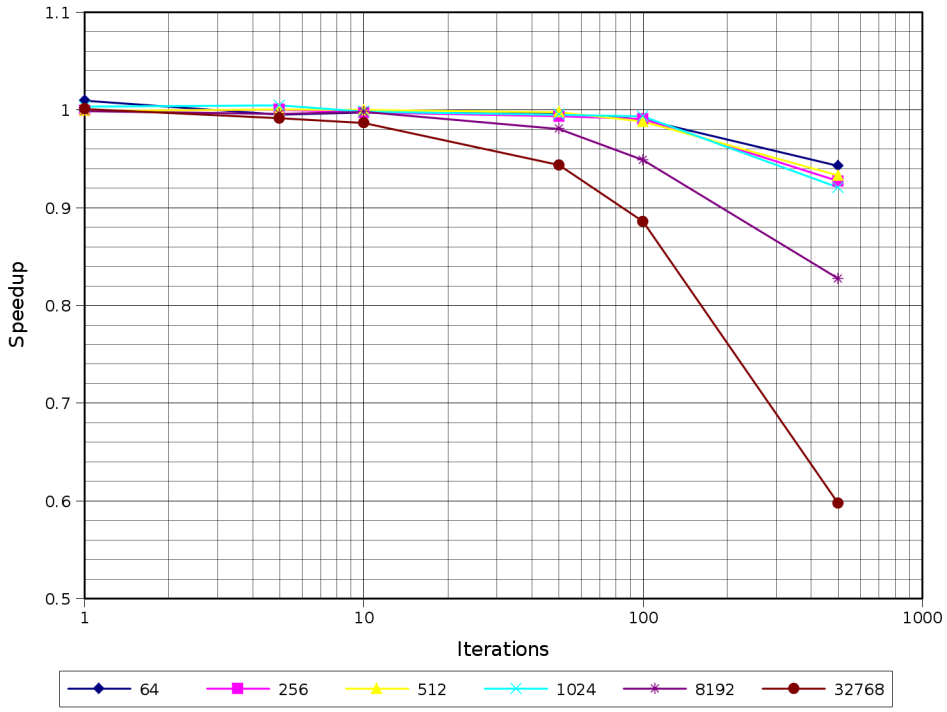

Figure 5.3: Speedup of the Vforce FFT run in hardware relative to the native hardware application versus the number of FFT iterations

This fact becomes more apparent when looking at the software scenario. As shown in Figures 5.4 and 5.5, the VSIPL++ FFT running on the CPU displays a similar overall performance curve to that of the hardware implementation: the average FFT iteration time decreases as the number of iterations increases and the performance increases. Despite running entirely in software, there is still a set-up time associated with initializing the FFT processing object.

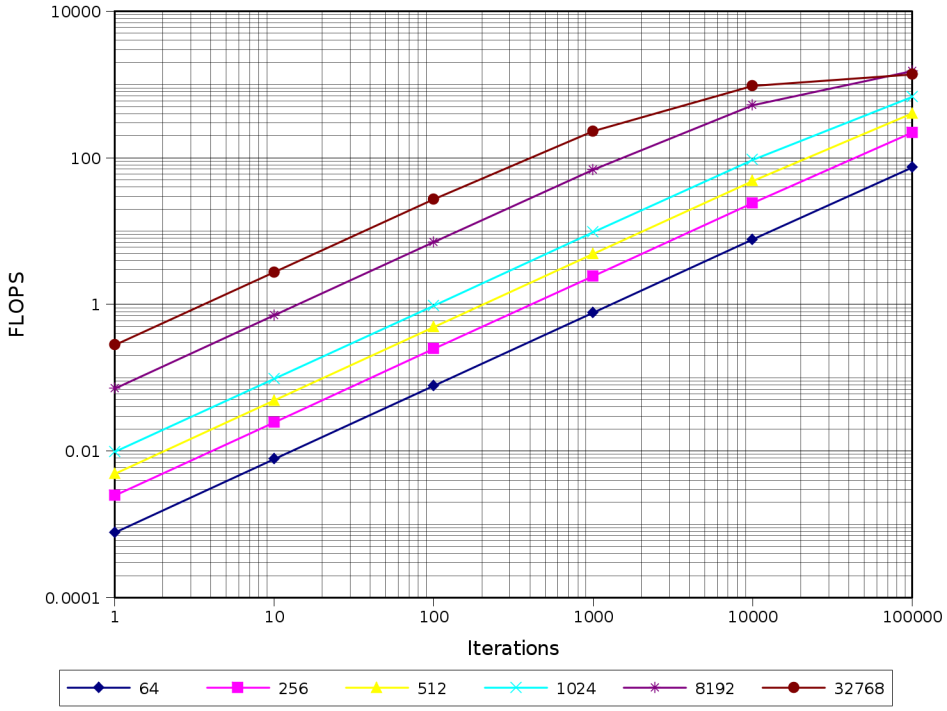

Figure 5.4: VSIPL++ Software FFT Performance

The speedup of the Vforce FFT running in software compared to the VSIPL++ FFT is shown in Figure 5.6. The Vforce FFT object utilizes a VSIPL++ FFT object for the software default, so any additional time taken is due to the overhead of Vforce itself. All of these differences are well within a few percent and most are likely due

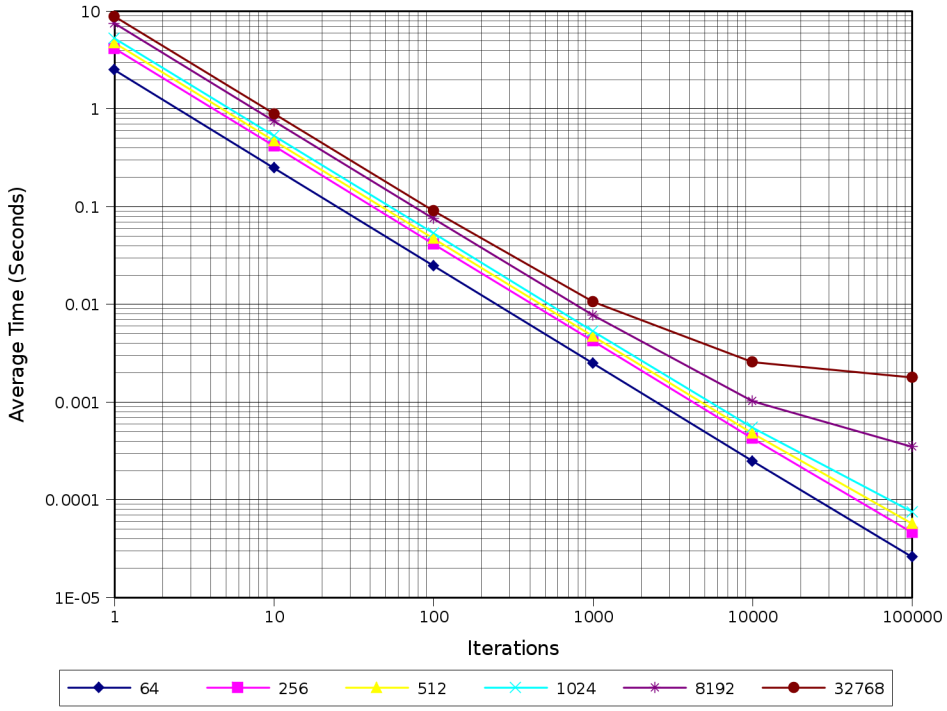

Figure 5.5: VSIPL++ Software FFT Execution Times

to measurement error[1]. It appears as if the overhead, which includes the IPC and kernel lookup, is largely negligible.

Our results show that using Vforce adds very little overhead while providing ease of programming. The Vforce application, nearly identical to the VSIPL++ implementation, was much more straightforward to write than the native API implementation which required many low-level calls in the user code. Without changing the efficiency or adversely affecting the performance of the SPP implementations, Vforce aids portability and productivity. Currently, the one major caveat seen in the

[1]The largest outlier, 64-points at 10^5 iterations, consistently tested faster than the baseline VSIPL++ FFT at this same configuration. The only thought on what might be causing this difference is some kind of difference in cache layout due to the larger size of the Vforce FFT application that is beneficial for performance.

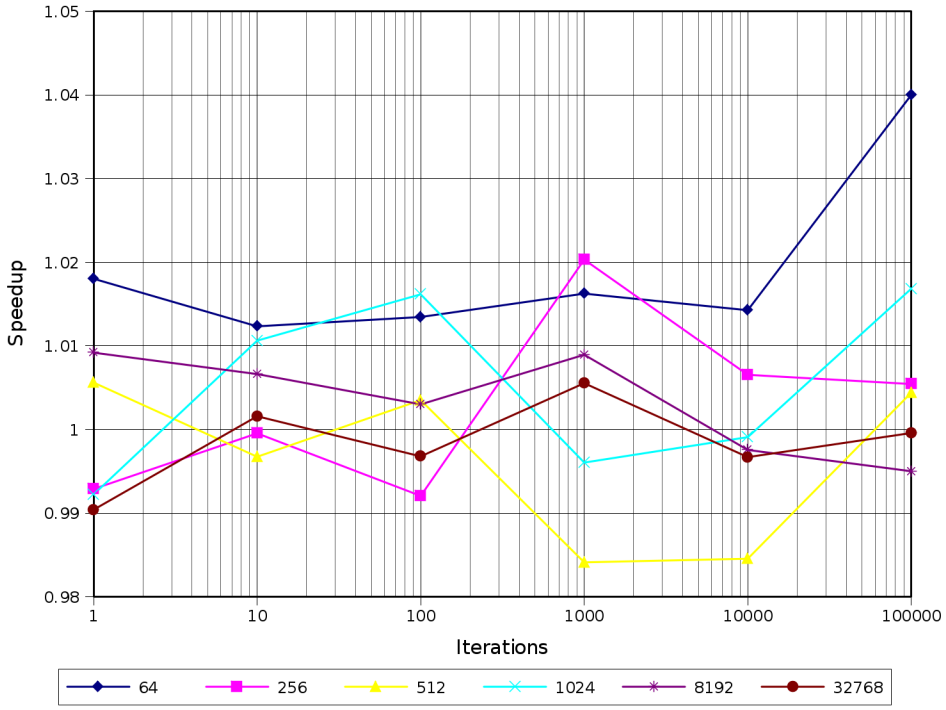

Figure 5.6: Speedup of the Vforce FFT run in software relative to the VSIPL++ FFT

Vforce implementation that uses SPP hardware are the data copy overheads when larger amounts of data has to be transferred. However, there is an identified solution, discussed in Chapter 6.2.

5.2 Beamforming

Like the FFT, the beamformer Vforce processing object was put inside a relatively simple program. The program utilized a data generator that would create sensor data based on the array and beam configuration information. This data is then given to the beamformer processing object for computation. The program profiles the time it takes for the data generation and the beamforming step. In addition, the Vforce processing object also includes a significant amount of profiling for many different parts of the beamforming process, including the data transfers, the weight computations, and the weight applications. This data is exposed to the test bench application for analysis.

A large number of scenarios were tested that included many combinations of the following: 1 to 128 beams, 4 to 64 sensors, 1024 to 131,072 time steps per update period, and 16 to 4,096 time steps of history for the weight update calculation. All the combinations of beams and sensors in power of 2 increments where run with the remaining combinations of options. As mentioned in Section 4.3.2, only scenarios with a maximum of four megabytes of sensor data per update period were tested. At four sensors a maximum of 131,072 times steps could be sent per update period,

and at 64 sensors a maximum 8,192 time steps could be sent per update period. For all cases tested, each was tested with an update period ranging from 1024 to the maximum number allowed based on the number of sensors in the array. Finally, the amount of history used for the weight update was limited to 4,096 time steps or half of the update period, which ever was smaller. For each combination of the other inputs, six adjacent powers of two were tested with the largest set at the maximum allowed based on the stated limitations. For all configurations, the dataset included 2^{20} time steps.

The large number of variables for configuration resulted in a large dataset. In order to filter through the data and provide meaningful insights into the characteristics of the Cray XD1 beamformer, several views of subsets of the results are presented. These views were created by fixing all of the options except one, and then varying that remaining option over all the values benchmarked. Where appropriate, performance results are included for both the entire application as well as only the FPGA accelerated weighted indexed multiply accumulate both with and without the FPGA programming time. The specific set of parameters used to examine the effect of each characteristic are given in Table 5.1. Creating data set views in this fashion helps to isolate how a single parameter affects the performance of the beamformer.

Case	Characteristic Tested	Beams	Sensors	Update Period	Weight History
1	Beams	Varies	32	2048	1024
2	Sensors	32	Varies	4096	256
3	Update Period	16	16	Varies	512

Table 5.1: Specific Combinations of Parameters Used in this Discussion

The following discussion will look at the beamformer performance on the Cray XD1 from two angles. First, to characterize only the FPGA implementation of the weight application component, the times for that portion of the application are examined. Second, the entire application's performance behavior is looked at. In both sections comparisons to the Mercury beamformer implementation are made.

5.2.1 FPGA Weight Application Performance

Figures 5.7, 5.8, and 5.9 show the speedup of performing weight application on a FPGA over running the weight application on one of the XD1's CPUs for cases 1, 2, and 3, respectively. The figures are based on only the weight application portion of the beamforming application – the portion offloaded to the FPGA – and include the data transfer for the weights, indicies, and sensor data for each update period as well as the processing in the speedup calculations. In addition, there are two curves per plot: one that includes the FPGA configuration time and one that does not. The curve that does not include the programming time can be thought of as the steady state speedup of the weight application if the beamformer were to be run continuously.

Figure 5.7 shows the weight application speedup versus the number of beams with 32 sensors and an update period of 2048 time steps and corresponds to Case 1 from Table 5.1. Figure 5.8 corresponds to Case 2 and plots the weight application speedup versus the number of sensors for 32 beams and an update period of 4096 time steps. Finally, Figure 5.9, corresponding to Case 3, plots the same speedup

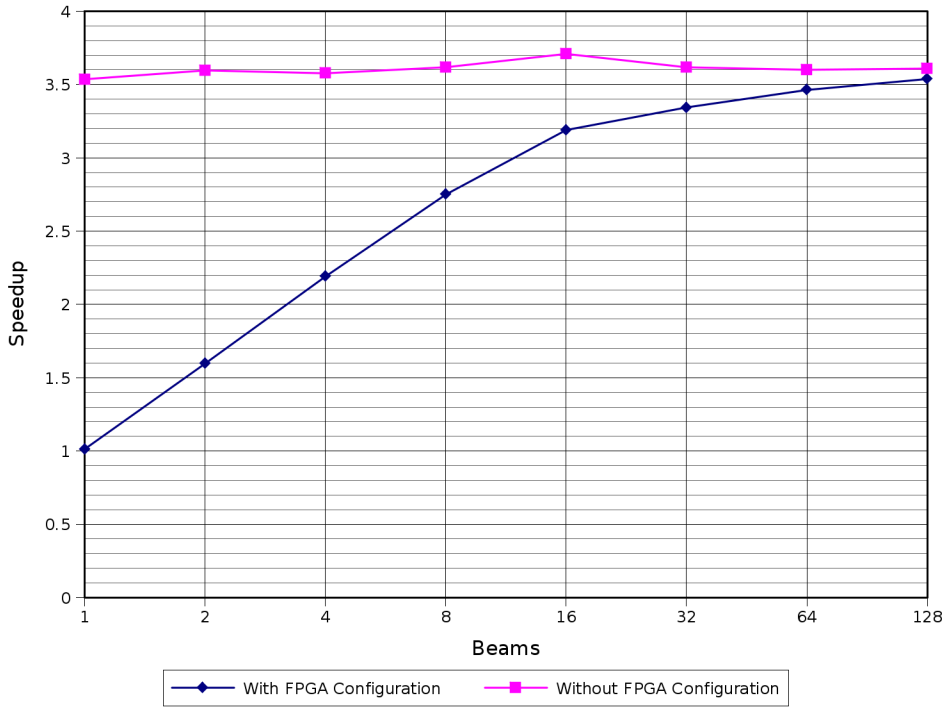

Figure 5.7: Speedup of the Vforce weight application step relative to VSIPL++ software for Case 1: 32 sensors and an update period of 2048 time steps

versus the length of the update period with 16 beams and 16 sensors. Since we are looking at only the weight application portion of the application, the data for this part of the analysis does not include the weight computation history, which only affects the total application. The data was collected using the the `beamformer` Vforce processing object, but with a weight computation class (implementing the `beamformer_weight_solver` interface) that does no work and returns immediately, thus removing the weight computation from the application performance.

First, we consider the results that do not include the FPGA configuration time. Figure 5.7 shows a relatively constant speedup of about 3.5 across all numbers of beams. Figure 5.9 displays a similar relatively constant speedup of around 2.35 when

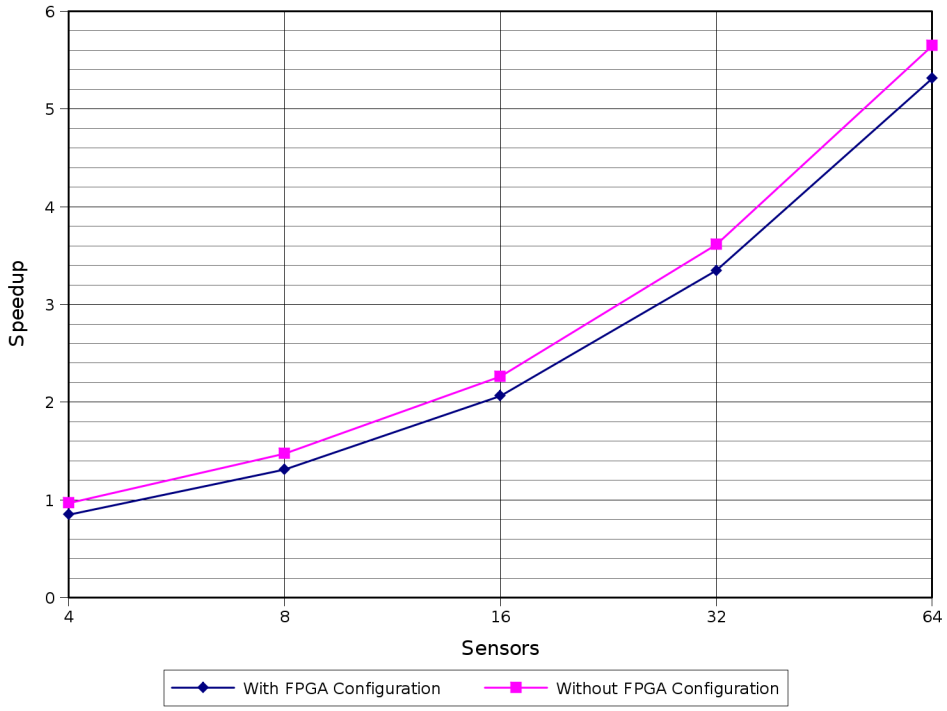

Figure 5.8: Speedup of the Vforce weight application step relative to VSIPL++ software for Case 2: 32 beams and an update period of 4096 time steps

it is the length of the update period that varies. On the other hand, when the number of sensors is changed, as shown in Figure 5.8, the speedup increases from slightly less than one at one sensor to more than 5 at 64 sensors. This behavior can be explained by the design of the XD1 FPGA implementation and the weight application algorithm, which is essentially three nested loops: one for time, one for beams, and one for sensors. For each time step and beam combination, the contribution to the signal of interested provided by each sensor is accumulated, and as discussed in Section 4.3.2, a bubble has to be inserted into the weight application pipeline after each time step and beam combination. However, all the sensor values for the particular time step and beam combination are streamed into the pipeline at

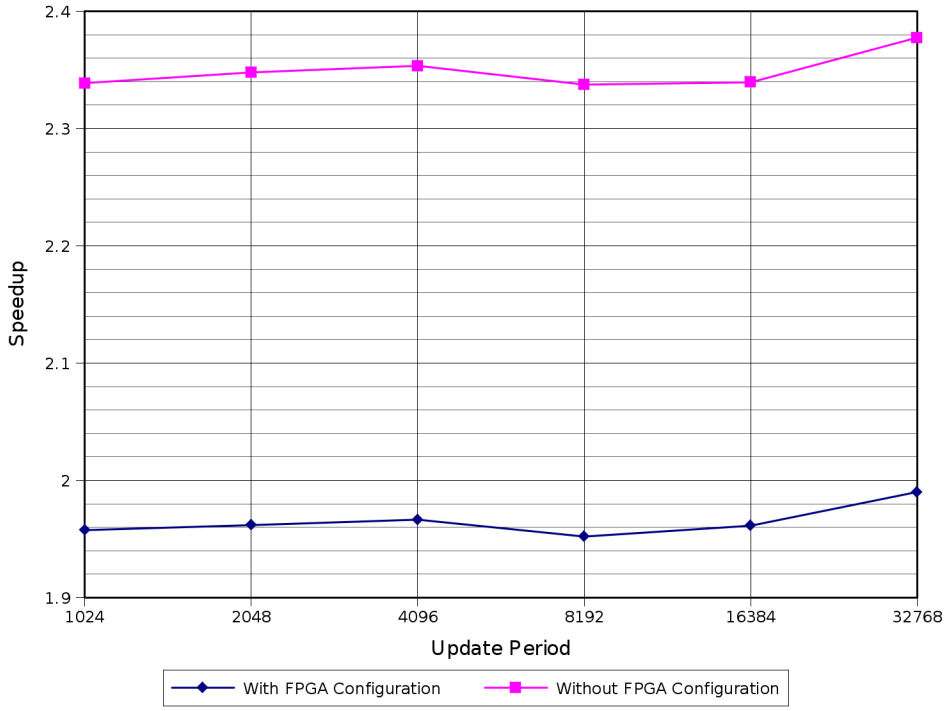

Figure 5.9: Speedup of the Vforce weight application step relative to VSIPL++ software for Case 3: 16 beams and 16 sensors

once, meaning that increasing the number of sensors increases the number of cycles between inserting bubbles which improves the throughput of the weight application and the speedup observed for the weight application.

Changing the number of beams has no effect on the speedup, because the ratio of stall cycles to processing cycles remains the same despite changing the amount of total work done but at the same relative rates. Varying the length of the weight update period also has no effect on the speedup in this case, because of the double buffering used for the sensor data and the streaming of results back to the CPU as they are produced. Shortening the update period increases the amount of data transferred to the FPGA but only in that more parameter data is sent to the board because

it is updated more frequently. However, the overall increase in data transferred is proportionally very small, because the parameter data size is small compared to that of the sensor data. Although the total data transferred does not increase much, the number of individual data transfers increases linearly with the inverse of the update period. In addition, there is an increased number of results transfers from the FPGA back to the CPU. The same amount of sensor data and results are transferred, but with more individual transfers, so the overhead for setting up these individual transfers is incurred more times.

The changing number of data transfers does not have a significant effect on the performance of the XD1 weight application implementation, however, due to the addition of non-blocking data transfers to Vforce and the use of double buffering. Even though the XD1 implementation still sets up a new return data transfer for each weight update period, the transfer is initiated when processing starts and completes automatically as soon as the data pipeline finishes the update period. Likewise, for sending data the XD1 implementation sends new sensor data to one of the double buffered sensor data RAM banks while the FPGA is operating on data from the other bank, effectively hiding the data transfer.

The cumulative time taken sending the parameter data increases as the number of update periods increases, but this value is relatively insignificant for the XD1 FPGA implementation compared to the processing time of about 11 seconds, as shown in Table 5.2.

Weight Update Period	Parameter Send	Data Send (hidden)	Processing
1024	0.0350	0.6361	11.0025
2048	0.0178	0.6336	11.0004
4096	0.0098	0.6351	11.0088
8192	0.0052	0.6286	11.0133
16384	0.0027	0.6368	11.0387
32768	0.0013	0.6361	11.0428

Table 5.2: Cumulative times of various parts of a hardware weight application iteration for the scenarios in Case 3 from Table 5.1.

While the length of the update period has little effect on the XD1 weight application, it has a significant impact on the previous implementation for the Mercury Computers system. The performance of the Mercury Computers system implementation decreases as the number of weight updates increase, as shown in Figure 5.10. The design requires waiting for the FPGA kernel to complete the weight application before initiating the return data transfer. Once the return data transfer is initiated, the program must wait for it to complete before sending new data and starting processing on a new update period.

When programming is included in the performance calculations, the behavior of the performance changes similarly to the way it does when programming time is not considered except that the speedup takes a hit due to the added FPGA setup time. On the XD1, this can easily be seen in the figures for Cases 2 and 3 where the speedup lines are separated by a consistent amount. However, for Case 1, when the number of beams varies, the speedup line including FPGA configuration asymptotically approaches the curve for speedup sans FPGA configuration time. The difference for Case 1 is that despite the data set size being consistent, increasing the number of

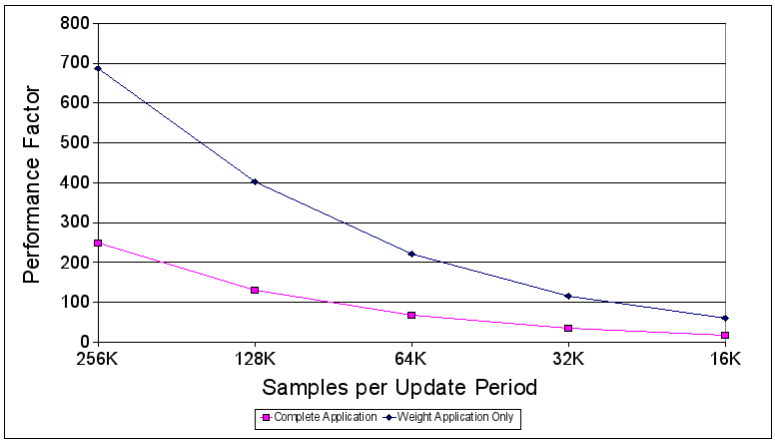

Figure 5.10: Speedup offered by the Mercury beamformer for weight application and the total application as a function of the update period. The graph is a modified version of one that appears in the results section of [2] for 64 sensors, 10,000 beams, and a weight update history of 64 time steps.

beams increases the amount of work performed on the data – the weight application process is repeated for each time step for *each beam*. Varying the number of sensors, as in Case 2, the data set size changes proportionally with the amount of processing required, and varying the length of the update period, as in Case 3, has no effect an the amount of computation required. For Case 1, as the amount of work to perform increases and the runtime increases, the FPGA configuration overhead time is amortized.

5.2.2 Complete Application Performance

It is also important to examine the performance of the entire application, not just the FPGA accelerated portion. Figures 5.11, 5.12, and 5.13 plot the speedup of the entire application for Cases 1, 2, and 3, respectively from Table 5.1. For comparison, this set of figures also show the speedup curves for the weight application sub-task.

In these graphs, the relationships between the speedup of the weight application and the overall application seem complex. The total application speedup does not track the weight application speedup very closely nor consistently. However, observations about the behavior of the beamformer can still be made by considering the weight application speedups as well as the varying time taken to perform the weight computation in software. Figure 5.14 displays the time taken to perform a weight computation on a log scale for a varying number of beams and sensors with update period fixed at 2048 time steps and the weight update history at 1024 time steps.

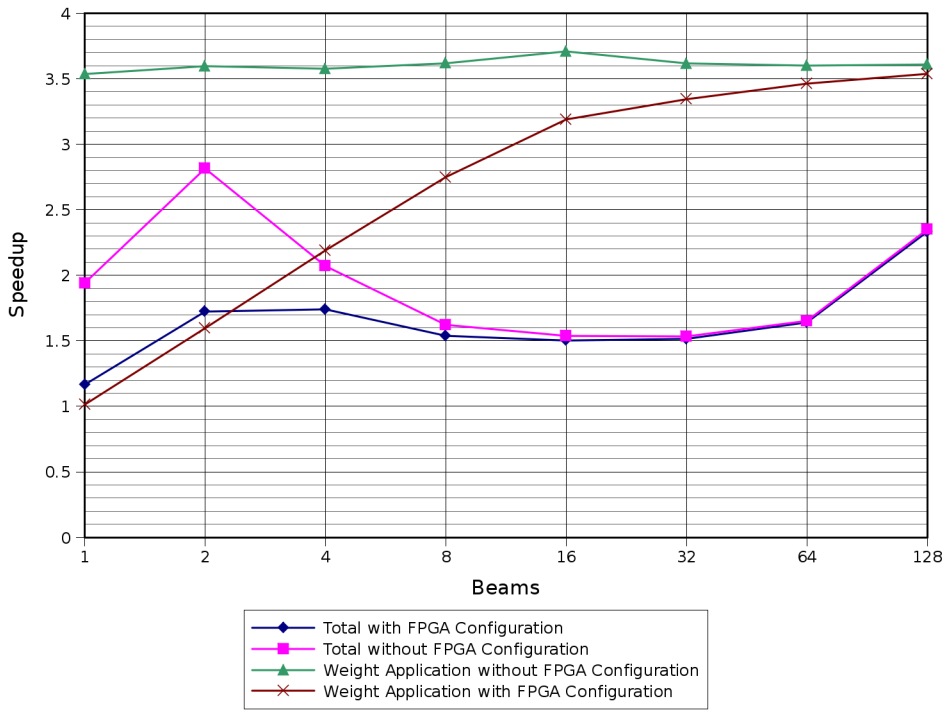

Figure 5.11: Speedup provided by Vforce of the entire application relative to VSIPL++ software with 32 sensors, an update period of 2048 time steps, and a weight update history of 1024 time steps

Generally, these figure appear as expected – there is an exponential increase in

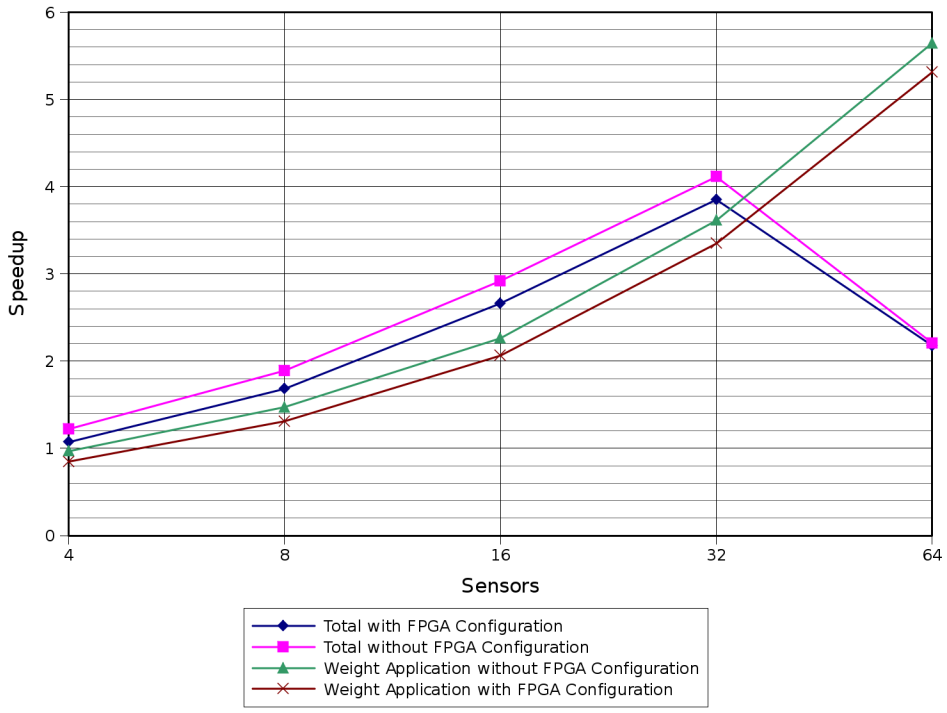

Figure 5.12: Speedup provided by Vforce of the entire application relative to VSIPL++ software with 32 beams, an update period of 4096 time steps, and a weight update history of 256 time steps

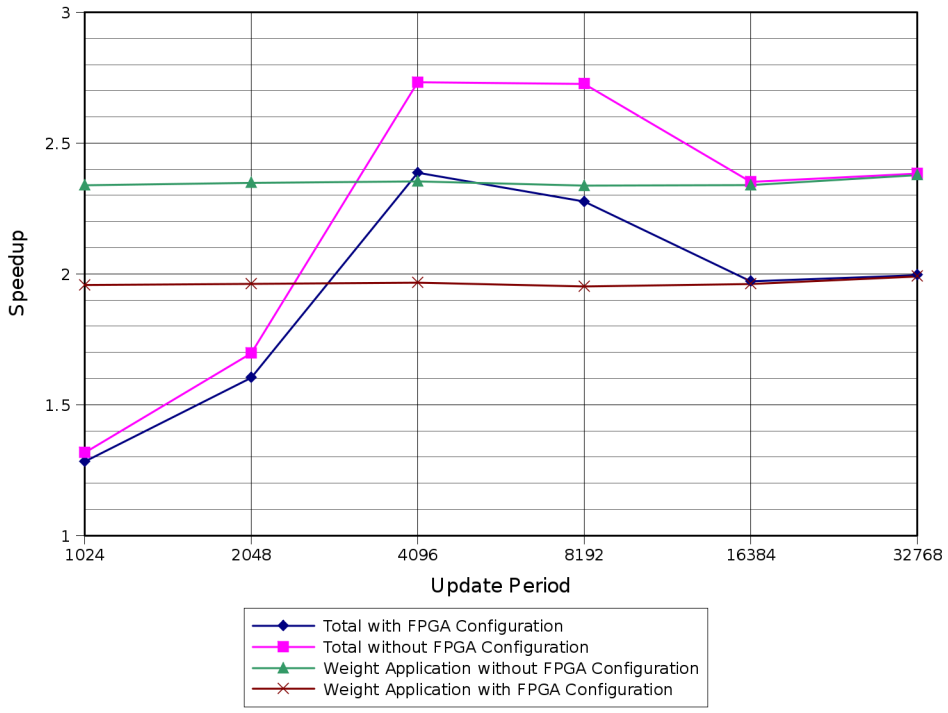

Figure 5.13: Speedup provided by Vforce of the entire application relative to VSIPL++ software with 16 beams, 16 sensors and a weight update history of 512 time steps

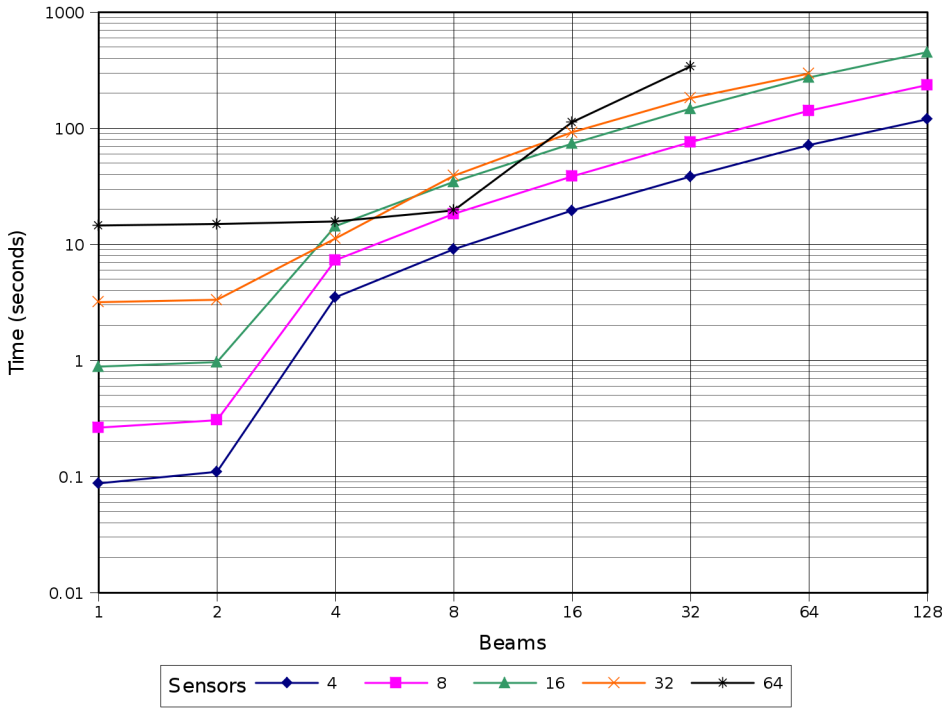

Figure 5.14: Time taken to perform the weight update step with an update period of 2048 time steps and a weight update history of 1024 time steps

the time it takes to compute new weights as the size of the weight update problem increases. However, there is a certain amount of baseline time the weight update component needs to find the new weights that doesn't scale down at the smallest sizes, as can be seen from where the curves deviate from a straight line on the log plot in Figure 5.14. This effect can be seen on the left side of Figure 5.11, where the speedup increases when moving from one to two beams. The time for a weight computation stays about the same between these two data points while the weight application work time increases resulting in the accelerated weight application portion of the program representing a larger ratio of the total execution time. However, this levels out past four beams to come closer to the relatively constant weight application speedup. The final increase at 128 beams is due to the linear least-squares solver, part of the VSIPL++ implementation on the Cray XD1 that is used for the weight computation, started displaying unpredictable behavior. At larger sizes the solver fails, which is the reason for missing data points in Figure 5.14. In addition, when the problem size approaches the failure point, the amount of time the least-squares solver needs to return a solution is erratic, and this combination of behavior is best exemplified by the 64 sensor curve in Figure 5.14. The erratic behavior could be due to either implementation issues in the least-squares solver or memory requirements, which get quite extreme for the larger solver sizes and was a limitation for the beamformer on the Mercury Computers system.

Figures 5.12 and 5.13 offer some insights as well. Figure 5.12 shows that the overall

speedup follows the weight application speedup closely until the weight computation sub-task's least-squares solver starts to behave erratically starting around 64 sensors. What is the most important to note about the figure, however, is that the total application speedup is greater than the speedup of the weight application sub-task alone. This is not normally possible under Amdahl's law, but happens here due to the extra concurrency utilized by Vforce.

Likewise, in Figure 5.13 we note that there is an area where the total application speedup is greater than the weight application speedup. However, in this case the weight computation sub-task size remains the same across all data points so an individual weight update takes, on average, a constant amount of time. On the far left of the graph, the short update periods result in more occurrences of weight computation for a given number of time steps, skewing the ratio of weight computation to weight application towards the software weight computation, which dominates the execution time dragging down the speedup. On the far right of the graph, the relatively infrequent weight computations result in the weight application sub-task becoming the bottleneck. In the middle, however, the weight computation and weight update are well balanced and the maximum overall speedup for Case 3 occurs.

These results are important as the beamformer application, a serial VSIPL++-like software program that will execute correctly in a GPP only environment, will not only automatically make use of a SPP implementation of a portion of the program but also take advantage of task concurrency, thus getting more performance out of

the target system while remaining highly portable. Figure 5.15 plots the observed run time of the beamformer application as well as the times of several smaller parts of the whole beamforming application. The observed run time is plotted as a dotted line with respect to the horizontal axis, but individual parts are shown with solid lines and graphed cumulatively so that each curve is plotted relative to the next lowest curve. The highest solid line represents the sum of all the measured parts. The observed run time behavior further confirms the behavior inferred from Figure 5.13. On the left side of the plot with short update periods the observed run time mirrors that of the sum of the components of the run time except for the weight application, because weight computation time dominates due to the frequent weight computations. The weight application time is hidden by the concurrency. On the right end of the graph at the largest update periods the weight computation becomes infrequent enough to contribute very little to the run time of the application and there is no longer enough GPP software processing to overlap with the weight application occurring on the FPGA, and the observed run time is closer to the sum of the components.

Mercury Comparison

The speedups presented here are smaller than those shown previously [2]. However, there are are a few factors that must be taken into consideration when a comparison between the two beamformer implementations is made. First, due to the difference in age and target markets between the machines, the AMD Opteron 248 in the Cray XD1 is much more powerful than the processor in the Mercury 6U VME, a low

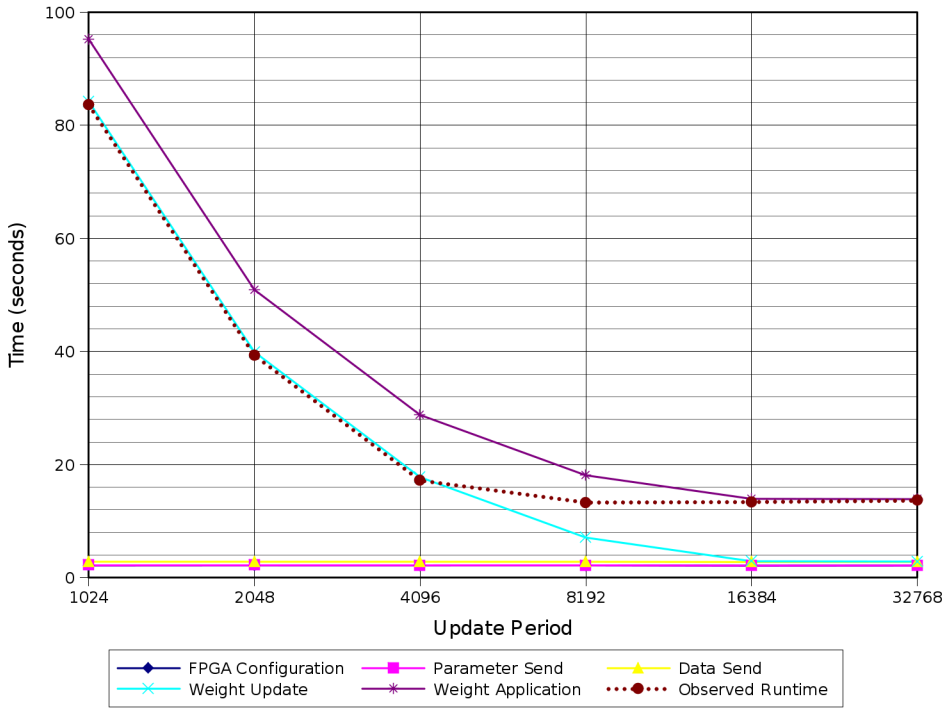

Figure 5.15: Observed run time of the entire application and the sum of the measure parts of the program versus the update period for Case 3.

power 800 MHz PowerPC 7447A. Combined with a much larger amount of RAM, the XD1 is able process larger data sets much faster than the Mercury 6U VME when running only in software. Second, the range of values for the combinations of parameters benchmarked on the XD1 was more realistic than those included in [2] for the Mercury 6U VME, excluding values like 10,000 beams. When comparing similar ranges, but not controlling for processor differences, the Mercury implementation displayed total application speedups of no greater than 16.32 with most speedups under 10. The XD1 speedups for the scenarios presented ranged from 1.22 to 4.11 without the FPGA configuration time included (to compare with the Mercury results, which were calculated without FPGA configuration) and from 1.07 to 3.85 when the speedup includes the FPGA setup time.

Another issue that makes comparisons more difficult is the lack of correspondence between test scenarios. While both systems were tested with a similar numbers of sensors, the Mercury beamformer was tested using powers of ten number of beams while the Cray XD1 was benchmarked on powers of two number of beams, and due to design differences, the Cray XD1 update period is much shorter than that of the Mercury beamformer. Despite this, three plots showing a general comparison are presented in Figures 5.16, 5.17, and 5.18. In all three, the number of sensors and time steps used in weight computation matches. Figure 5.16 shows total application execution times for both beamformers operating on one beam. For Figures 5.17 and 5.18 the range represented by the power of two on either side of 10 and 100,

respectively, is plotted for the Cray. The Cray update period was fixed at 8192 while the Mercury update period varied inversely with the sensor count from 2^{18} down to 2^{14}. In the figures, dashed lines show the weight application only and solid lines represent the entire application.

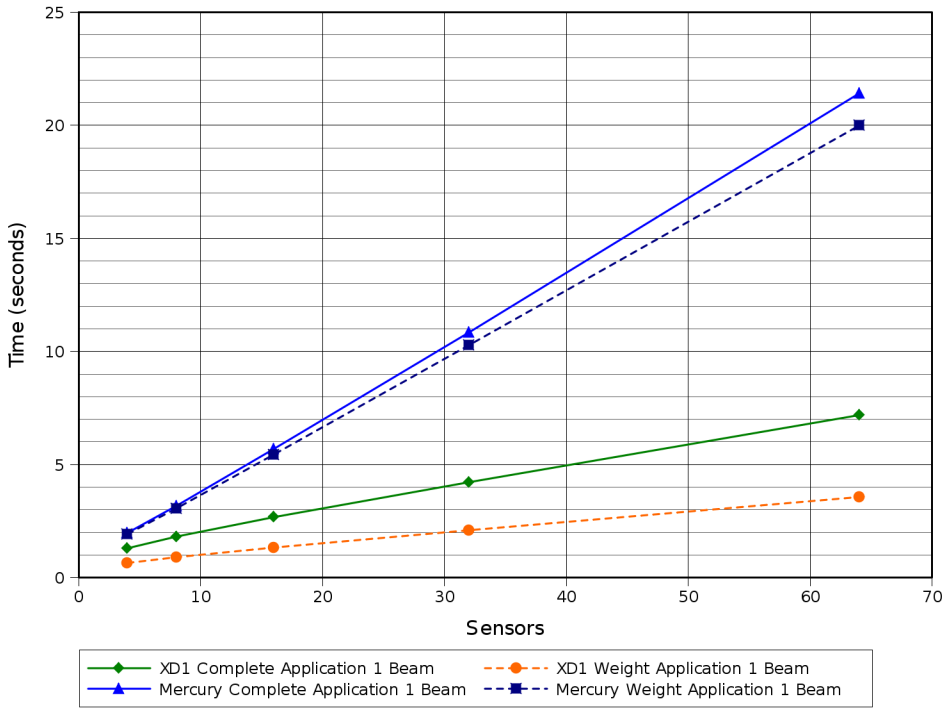

Figure 5.16: Cray XD1 and Mercury 6U VME comparative measured weight application and total application times for 1 beam

The three figures show that, despite the inequalities in the comparison that favor the Mercury implementation, namely the shorter update periods on the Cray system and accumulator design that requires stalls during the weight application, the Cray implementation has performance roughly comparable to that of the Mercury version, with the Cray performing better, relatively, on larger combinations of beams and sensors. Compared to the Mercury implementation, the concurrency in the Cray

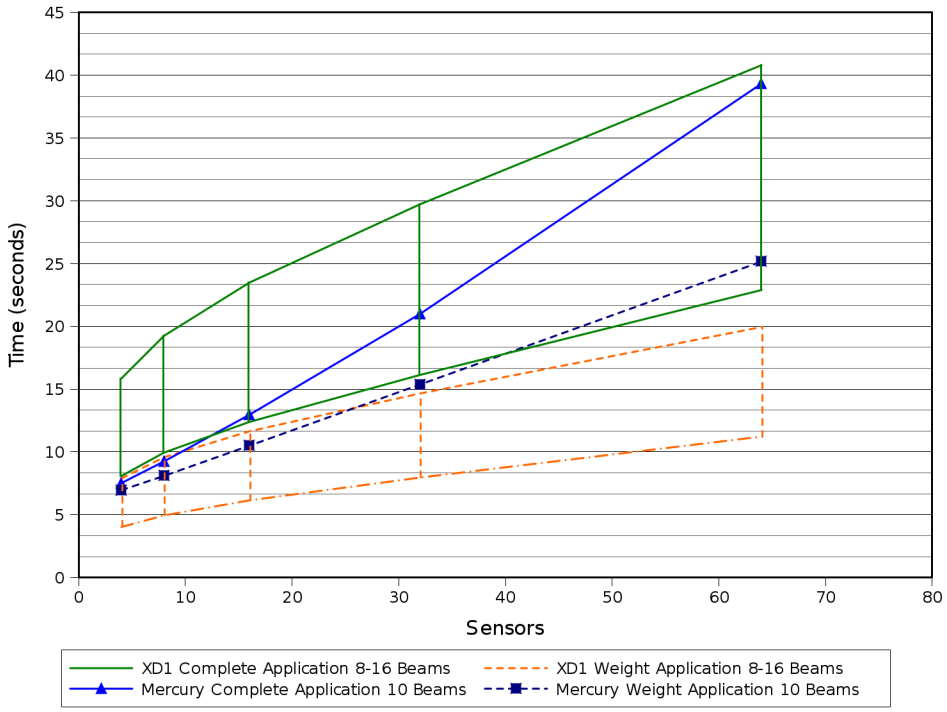

Figure 5.17: Cray XD1 and Mercury 6U VME comparative measured weight application and total application times for approximately 10 beams

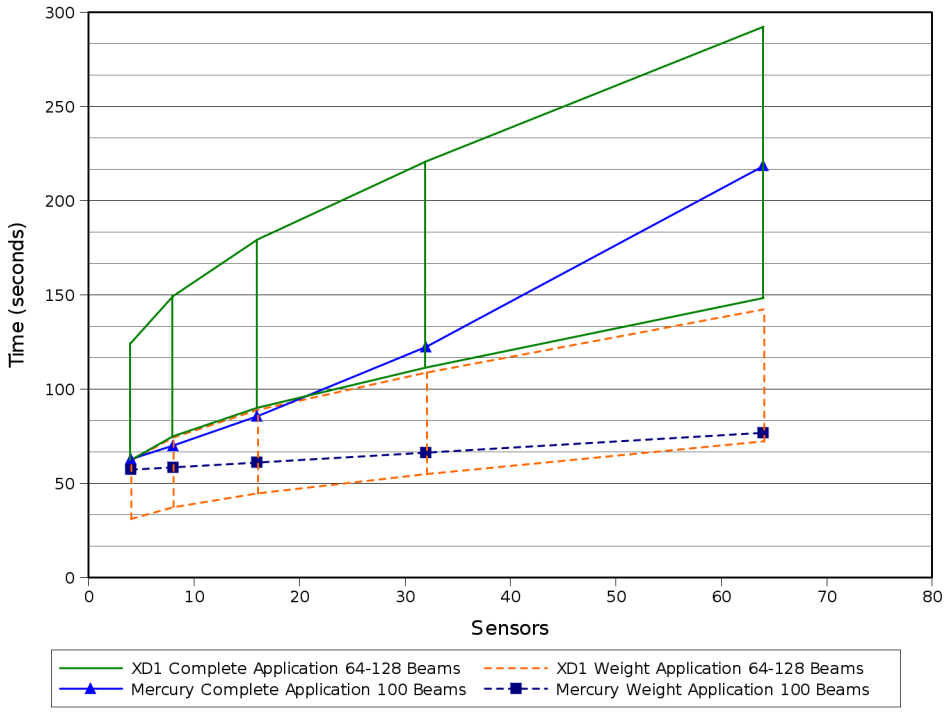

Figure 5.18: Cray XD1 and Mercury 6U VME comparative measured weight application and total application times for approximately 100 beams

implementation decreases the total run time of the application and allows for much shorter update periods that include more data in the weight update process. This results in a beamformer that is more responsive to new interference, despite the extra update period the Cray design requires for new interference to affect the weight computation. Despite the modest speedups, this application displays some of the benefits of using *Vforce*, including making available the most amount of concurrency allowed by the two main subcomponents, resulting in the application extracting as much of the performance increase as the hardware design allows.

5.3 Summary

This chapter presented the testing procedures for the FFT and beamforming processing objects and analyzed the results. Vforce adds insignificant overhead when running in a software only environment. There is some performance degradation when running kernels on SPPs due to data copying during data transfer. The beamforming application provided better-than-Amdahl's law speedup by taking advantage of concurrent GPP and SPP activity, a feature of Vforce. The next chapter discusses conclusions about Vforce made from the two case studies' performance results. In addition, suggestions for future research are provided.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The rapid rate of hardware change in the hybrid supercomputing field enables new advances to be made on the backs of ever increasing processing power. The hardware platforms, however, are a moving target. New systems are introduced regularly, and recently there has been a proliferation of high performance processing architectures, including many-core, GPU, Cell, and hybrid FPGA systems. Hardware resources and APIs regularly change among successive versions of the same product. All of these factors combined make producing portable and maintainable programs a daunting challenge. This thesis presents Vforce, a framework that abstracts the differences among these platforms presenting the user application with a single consistent interface for all special purpose processor types.

In addition to providing a high level of portability, the framework is built on top of VSIPL++. Vforce extends the processing object concept to surround an entire hardware implementation of a particular algorithm so that the end user requires no

knowledge of the target platform or even that a SPP is going to be used. The multiple layers of Vforce, one at the processing object level and one at the DLSL level, allow for flexible expansion of Vforce in multiple independent directions, including adding new processing objects, which are portable, and new hardware platform support through a DLSL or adding to the kernel library. The multiple levels of abstraction also help to boost productivity by maximizing code and hardware implementation reuse. Vforce provides separation between hardware knowledge requirements and application level knowledge requirements.

In addition to documenting the features provided by Vforce and their implementations, this thesis presented two case studies of Vforce usage. The first, the FFT processing object, provides an example of how existing VSIPL++ applications can be run on SPP hardware without modifying the source code and without sacrificing portability or existing performance. When running in software, the FFT provides identical performance to the original, indicating that the framework adds little overhead to applications. The hardware FFT results highlight the importance of minimizing any overhead as an extra data copy operation affected the performance of the hardware version. However, a proposed method to remove this bottleneck is identified in the next section.

The second study, an adaptive time-domain beamformer, examines building a large compound processing object that takes advantage of task concurrency, a feature that is necessary to achieve maximum speedups when using SPPs. The speedup

provided by the hybrid hardware/software implementation on the weight application as well as the entire application, compared to a software only implementation, was smaller for the Cray XD1 than the previously existing implementation for a Mercury Computers system. This is not surprising due to the more powerful baseline processor in the XD1 and different testing scenarios. However, the XD1 implementation provides similar raw performance in terms of the number of time steps processed per second while maintaining a better balance between software and hardware processing and much smaller update periods, increasing the beamformer's adaptive responsiveness.

6.2 Future Work

While Vforce has been shown to provide tangible benefits in many areas, there are a number of things that have been identified as possible ways to improve the framework further. These include eliminating the data copying, changing Vforce to elevate SPPs closer to peer status with GPPs as opposed to the current master/slave relationship, more RTRM functionality, adding support for more platforms, and implementing more applications.

Currently, the biggest drawback to using Vforce on some platforms is the extra data copying that is required when transferring data via DMA. Some machines, like the Cray XD1, place restrictions on the types of memory that can be used for DMA transfers. The XD1, for example, requires that memory blocks used for

DMA transfers be multiples of the machine's page size (4 KB in the case of the XD1) and that the memory blocks are aligned on page boundaries. Once a valid DMA memory block is obtained, the user must register the memory block with the XD1's FPGA API using the `fpga_register_ftrmem` function. A second, but related issue is that since Vforce is built on top of VSIPL++ and designed to work with any implementation, it is not possible to control how the library allocates memory when it allocates memory automatically. As a convenience, when a VSIPL++ view is instantiated it can automatically instantiate its own VSIPL++ block for storage. This block is in an admitted state and cannot be released from the VSIPL++ library's control, which prevents Vforce from gaining access to the data when the library creates its own blocks. This requires Vforce to use the individual `put` and `get` methods on the view object, which likely adds significant overhead on top of the actual copying. Vforce could require users to instantiate their own blocks and manually admit them to VSIPL++, which would allow Vforce to gain access to the data block that a view is associated with directly, but this still does not address the issues related to DMA memory requirements.

To address both issues at once, a likely target for future work is one or more custom VSIPL++ blocks. The VSIPL++ specification details the functionality that is required of memory blocks, and a custom Vforce block that allocates DMA-able memory as well as allows the framework to access the data for transfers at all times could be created. The type of block underlying a view is specified by a template

argument of the view and defaults to the `Dense` block. The user could specify the new Vforce block type in the template arguments to view declarations and switch any block pointer declarations or instantiations to the new block type. The creation of a new block type would not prevent the user from relying on the VSIPL++ views to instantiate their own storage blocks, as the view uses whatever block type is specified by the template argument. This mechanism would seamlessly alleviate many of the data copying problems seen in the results of Chapter 5. While an improvement, the proposed solution does not eliminate data copying in all circumstances. It would not prevent the necessity of data copying if a view with a stride greater than one was sent to an FPGA, for example. The requirements for these memory blocks may vary from platform to platform and from SPP to SPP, so any Vforce block would also need to rely on some kind of abstraction mechanism

While it probably is not possible to elevate the SPPs in a system to the same level as a GPP for a large number of reasons, another area for future work is adding more functionality to Vforce that allows the framework to take better advantage of hardware systems where each SPP is not tightly bound as a slave to a particular processor. This functionality might include SPP to SPP data transfer, multiple outstanding data transfers per SPP (more than the current limit of one in each direction), and tracking or modeling of the location of data within the system.

There are a large number of other areas that could benefit from additional work. The current manager implementation works, but much more functionality could be

added, as discussed in Section 3.2.2. The communication between the RTRM and GHO could be made more robust but would have to be balanced against any additional overhead. Although only two applications have been created for Vforce at this time, it appears as if there will be a lot of redundant code for working with the GHO among Vforce processing objects. Exploring ways to automate or abstract this redundancy may add to the productivity already afforded by Vforce. Finally, future effort can always be directed towards implementing more applications with Vforce and adding support for more platforms. In particular, other platforms that pair FPGAs with GPPs, such as the SGI RASC platform [20] or the SRC-7 [21], are being considered. Very different SPPs including the IBM/Toshiba/Sony Cell Broadband Engine[12] and graphics processing hardware (GPUs) are also under consideration for support with Vforce. In the Cell, the Synergistic Processing Elements (SPEs) can be treated as individual SPPs. For both GPUs and the Cell's SPEs it appears that a library of highly optimized pre-compiled kernels is the best way to achieve maximum performance, the same approach taken by Vforce.

Finally, Section 4.3.2 discussed the implementation of the design of the weight application bitstream for the Cray XD1's FPGAs. As mentioned, the accumulator design uses only a single adder to save resources on the FPGA. As a result, it can take up to 35 cycles from the last input until the final result is calculated, and during this time new inputs cannot be applied to the input. If this accumulator was replaced with an design that used multiple adders, the time to the final output

might be reduced. More importantly, however, is that a different design could allow the loading of the next set of values for accumulation immediately instead of having to insert an extremely long bubble in the pipeline after each beam or time step. This change would result in larger speedups for the weight application general by increasing the weight application pipeline's active to stalled ratio.

Bibliography

[1] E. Anderson, J. Argon, et al. Enabling a uniform programming model across the software/hardware boundary. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, April 2006.

[2] A. A. Conti. A hardware/software system for adaptive beamforming. Master's thesis, Dept. of ECE, Northeastern University, December 2006.

[3] Cray Legacy Products. `http://www.cray.com/service/legacy.html`. Last accessed 19 October 2007.

[4] Cray XD1 Datasheet. `http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf`. Last accessed 19 October 2007.

[5] R. Deville, I. Troxel, and A. George. Performance monitoring for run-time management of reconfigurable devices. In *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2005.

[6] FFTW Benchmark Methodology. `http://www.fftw.org/speed/method.html`. Last accessed 26 October 2007.

[7] FFTW Home Page. `http://www.fftw.org/`. Last accessed 17 October 2007.

[8] M. Franklin, J. Maschmeyer, et al. Auto-pipe: a pipeline design and evalutation system. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[9] I. Frost, O.L. An algorithm for linearly constrained adaptive array processing. In *Proceedings of the IEEE*, volume 60, pages 926–935, August 1972.

[10] High Performance Embedded Computing Software Initiative (HPEC-SI). `http://www.hpec-si.org`. Last accessed 17 October 2007.

[11] B. Holland, J. Greco, et al. Compile- and run-time services for distributed heterogeneous reconfigurable computing. In *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, June 2006.

[12] IBM developerWorks: Cell Broadband Engine resource center. http://www-128.ibm.com/developerworks/power/cell/. Last accessed 30 October 2007.

[13] Mercury Computer Systems, Inc. http://www.mc.com/. Last accessed 19 October 2007.

[14] N. Moore, A. Conti, L. King, and M. Leeser. An extensible framework for application portability between reconfigurable supercomputing architectures. *IEEE Computer Magazine*, pages 39–49, March 2007.

[15] N. Moore, A. Conti, M. Leeser, and L. S. King. Writing portable applications that dynamically bind at run time to reconfigurable hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, April 2007.

[16] OSC-Springfield. http://www.osc.edu/springfield/. Last accessed 19 October 2007.

[17] W. Peck, E. Anderson, et al. Hthreads: a computational model for reconfigurable devices. In *Field Programmable Logic and Applications*, August 2006.

[18] QDR II SRAM: A Design Guide. http://qdrsram.com/technotes/cypress/QDR2_design.pdf. Last accessed 24 November 2007.

[19] RACE++ 6U VME System. http://www.mc.com/products/productdetail.aspx?id=2856. Last accessed 19 October 2007.

[20] SGI RASC. http://www.sgi.com/products/rasc/. Last accessed 30 October 2007.

[21] SRC-7. http://srccomputers.com/products/src7.asp. Last accessed 26 November 2006.

[22] H. L. V. Trees. *Optimum Array Processing*. Wiley-Interscience, New York, 2002.

[23] B. D. Van Veen and K. M. Buckley. Beamforming: a versatile approach to spatial filtering. *IEEE ASSP Magazine*, 5(2):4–24, April 1988.

[24] Variable Precision Floating Point Modules. http://www.ece.neu.edu/groups/rcl/projects/floatingpoint/index.html. Last accessed 19 October 2007.

[25] Vector Signal Image Processing Library. http://www.vsipl.org. Last accessed 17 October 2007.

[26] Vector Signal Image Processing Library Forum. http://www.vsipl.org/forum.html. Last accessed 17 October 2007.

[27] VSIPL++ Specification 1.01. `http://www.hpec-si.org/spec-1.01-final.pdf`. Last accessed 17 October 2007.

[28] M. Vuletic, L. Pozzi, and P. Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers*, 22(2):102–113, 2005.

[29] X. Wang, S. Braganza, and M. Leeser. Advanced components in the variable precision floating-point library. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 249–258, April 2006.

[30] Xilinx CORE Generator System Overview. `http://www.xilinx.com/ise/products/coregen_overview.pdf`. Last accessed 19 October 2007.

[31] L. Zhuo, G. Morris, and V. Prasanna. Designing scalable FPGA-based reduction circuits using pipelined floating-point cores. In *IEEE International Parallel and Distributed Processing Symposium*, April 2005.