A case study demonstrating the extended SIMPPL framework for multi-FPGA platforms

David Dickin, Jason Lee and Lesley Shannon
Simon Fraser University

This paper presents a design implementation for accelerating the Fourier Integral Operator (FIO) kernel of a seismic imaging application. Our design uses an extended version of a previously proposed model for Systems Integrating Modules with Predefined Physical Links (SIMPPL) that provides an architectural framework for designs implemented using multiple Field Programmable Gate Arrays (FPGAs). This extended version of SIMPPL not only reduces design time by accelerating integration, it facilitates multi-FPGA designs by abstracting them as a single reconfigurable fabric. In this paper, we build upon a previous implementation of the FIO kernel to better leverage our target platform, the Berkeley Emulation Engine 2 (BEE2), a multi-FPGA board. Dramatic improvements in performance are attained due to the addition of a more efficient DDR2 SDRAM memory controller and an expanded datapath width. The new implementation presented in this paper provides a 357x increase in throughput compared to an optimized software implementation of the FIO kernel and a 29x increase in throughput relative to our previous FIO kernel implementation.

Categories and Subject Descriptors: C.3.e [**Computing Systems Organization**]: Special-Purpose and Application-Based Systems—*Reconfigurable Computing*; C.3.d [**Computing Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded Systems*; B.9.2 [**Hardware**]: Performance Analysis and Design Aids; C.1.3.f [**Computing Systems Organization**]: Processor Architectures—*Heterogeneous (hybrid) systems*

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: FPGAs, Hardware/Software Codesign, System-on-Chips, Design Methodologies

1. INTRODUCTION

Originally, the Systems Integrating Modules with Predefined Physical Links (SIMPPL) model was formulated as an architectural framework for System-on-Chip (SoC) designs [Shannon and Chow 2007]. SIMPPL allows designers to reduce system integration time and improve the potential of Intellectual Property (IP) reuse for the component modules in their design. Recently, new elements were added to the SIMPPL model to extend the framework to support implementation platforms comprising multiple Field Programmable Gate Arrays (FPGAs) by abstracting them as a single reconfigurable fabric [Dickin and Shannon 2008].

In other recent work, we also presented a prototype system to accelerate a single-precision, floating-point implementation of a Fourier Integral Operator (FIO) kernel for a seismic imaging application [Lee et al. 2008]. Floating point computational

cores have traditionally been uncommon for systems implemented on FPGAs due to their large resource consumption [Ligon III et al. 1998]. However, as the size of modern FPGAs increases, floating point accelerators [Underwood and Hemmert 2004][Morris and Prasanna 2005] are now being realized as SoC designs more frequently [Greenbaum 2002]. Regrettably, for large and complex systems, such as those required to accelerate applications like molecular dynamics [Scrofano and Prasanna 2006], seismic imaging [He et al. 2004], and logic emulation [Varghese et al. 1993], a single modern FPGA possesses insufficient resources. These types of systems, including our FIO kernel, require a multi-FPGA platform for implementation; therefore, our FIO kernel was designed for implementation on the Berkely Emulation Engine 2 (BEE2), a multi-FPGA board offering five Xilinx Virtex II Pro 70 FPGAs.

In this paper, we apply the extended SIMPPL model to the original prototype FIO system. The updated FIO kernel design we present here has several improvements, in addition to leveraging the extended SIMPPL model, to increase its performance beyond that of the earlier prototype. Specifically, these enhancements include:

—The incorporation of the FIO kernel component modules into the extended SIMPPL framework: The cores in the system are augmented with SIMPPL Controllers to handle system-level control and communication. The inter-FPGA communication ports are controlled by the recently introduced SIMPPL Repeaters [Dickin and Shannon 2008].

—The inclusion of a dedicated memory controller: The original FIO system prototype utilized a soft processor, the MicroBlaze [Xilinx Inc.], to perform the off-chip memory accesses via software; this was found to be a performance bottleneck [Lee et al. 2008]. In the new implementation we present here, all off-chip memory access is done via a custom hardware DDR2 SDRAM Memory Controller from the BEE2 design repository [Chang et al. 2005].

—An increased bus width for data transfers: The width of the First-In-First-Out buffers (FIFOs) between all system modules is increased from 64-bit to 256-bit to improve bandwidth between computation cores and memory.

The new implementation for the multi-FPGA BEE2 platform presented in this paper provides a 357x increase in throughput compared to an optimized software implementation of the FIO kernel and a 29x increase in throughput relative to our previous FIO kernel implementation.

The remainder of this paper is organized as follows. Section 2 provides background on computing with multi-FPGA Platforms and Section 3 details the extension of SIMPPL to support multi-chip platforms. The components of the FIO kernel design and their architecture are discussed in Section 4 and an analysis of the system's performance is given in Section 5. Finally, Section 6 concludes the paper and contemplates future work.

2. MULTI-FPGA PLATFORMS

The reconfigurable nature of FPGAs provides several key advantages that make Multi-FPGA platforms attractive for High-Performance Computing (HPC). FP-

GAs provide the opportunity to exploit hardware parallelism to accelerate an application beyond the means of a sequential execution model. Computations are done using the circuitry that is configured to match a specific applications needs. Also, incremental improvements or bug fixes to a circuit are easily accomplished when a design is implemented on an FPGA. A custom designed ASIC can provide greater performance improvments while using a smaller footprint than an FPGA. However, the nonrecurring engineering (NRE) expenses and greater time to market required for an ASIC solution make this approach untenable for low volume projects.

The BEE2 [Chang et al. 2005] is a multi-FPGA platform designed for use as a building block for High Performance Computing (HPC) systems and has been used as the platform for several high-performance applications [Brodersen et al.][Tkachenko et al. 2006]. The BEE2 features five Virtex 2 Pro FPGAs for application implementation. Each of these FPGAs has four independent channels to 1 GB DDR2 memory DIMMs. The FPGAs are connected together using high speed LVDS buses in a ring topology and also have high-speed connections for linking multiple boards together. Figure 1 shows a high level block diagram of the BEE2.

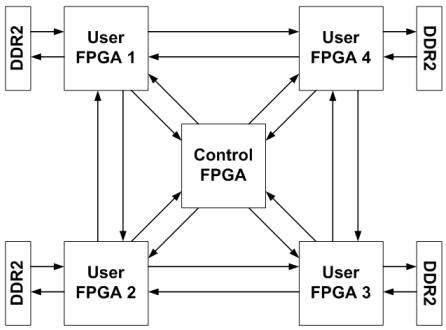

Fig. 1. A high level block diagram of the BEE2 multi-FPGA platform.

3. SIMPPL

3.1 The SIMPPL Model

SIMPPL is a modelling approach for SoC's in order to expedite system integratio [Shannon and Chow 2007]. In this work we define an SoC to be a collection of functional units that interact to perform a desired operation, implemented in the form of an IC. The design could be implementation platform can range anywhere from a fully custom ASIC to an FPGA.

A model of a generic SoC that uses the SIMPPL framework is shown in Figure 2. Under SIMPPL, the functional units of an SoC are termed Computing Elements (CEs). Each CE contains the circuitry to perform a specific computing task. Unidirectional, point-to-point links in the form of FIFOs are used to connect the various CEs together and enable communication. A CE can contain either application-specific hardware to implement functionality or a processing node that

runs software to do the computation. These two options are termed Hardware CE (HW CE) and Software CE (SW CE). Our work here is focused primarily on HW CEs.

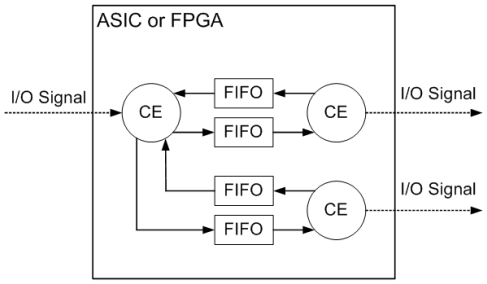

Fig. 2. An example of a system modelled using SIMPPL.

A HW CE is composed of three components designed to separate a functional unit's datapath, system-level control, and communication from each other. The three components are,

—A Processing Element (PE), which encapsulates the modules datapath (e.g. a 32-bit multiplier).

—A SIMPPL Controller, which executes a local program and provides system-level control for the PEs operation and inter-CE communication.

—A SIMPPL Control Sequencer (SCS) that contains the local program for the controller.

A block diagram of a HW CE showing how these three components are connected can be found in Figure 3. The PE is the portion of the CE that does the computing. It has connections to the SIMPPL Controller and optionally the I/O pins of the SoC. The SIMPPL Controller is a lightweight microcontroller. The Instruction Set Architecture (ISA) of a SIMPPL Controller has multiple configurations, allowing a SIMPPL Controller to be customized for a specific PE. The SIMPPL Controller is programmable. It's program is stored in the third component of a HW CE, the SIMPPL Control Sequencer (SCS). SIMPPL imposes some constraints on how the functional units of an SoC are controlled and how they communicate with each other. By following these constraints the functional units can be integrated immediately, tested thoroughly, and reused readily.

Each SIMPPL Controller controls two FIFOs, one to transmit (TX) and one to receive (RX). These FIFOs are used to link CEs together, permitting the exchange of commands and data. Each CE will require one or more SIMPPL Controllers depending on how the CE is integrated into a system (e.g. pipelined, shared memory, connections to multiple other CEs, etc). There are a few common variants of the SIMPPL Controller's ISA to handle different forms of communication. For example, a Full SIMPPL Controller has an ISA supporting both read and write

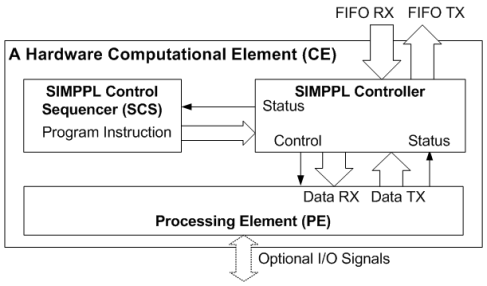

Fig. 3. A block diagram showing the components of a hardware CE.

related instructions. In contrast are the Consumer and Producer SIMPPL Controllers. The former's ISA supporting only data receiving related instructions while the latter's only has instructions related to generating data.

Communication between CEs is packet-based. The packets have a specific format which is illustrated in Figure 4. Each word of a packet is composed of a *Control Bit* and a *Program Word*. The width of a Program Word is not fixed by SIMPPL (32-bits in the figure), but all connected SIMPPL Controllers must use the same Program Word width in order to function properly. The Control Bit is always a single bit, meaning that the width of the FIFOs used must be one greater than the desired Program Word width.

The first word of a packet is the *Instruction Word*, which contains the Number of Data Words (NDW) in the packet and an opcode that tells the receiving SIMPPL Controller how to handle the packet. The Control Bit is set to indicate that this is in fact an Instruction Word. The next word is called the *Address Word* and is only present for certain opcode values. The rest of the packet words are *Data Words*. This is the data payload of the packet. Typically, this data is passed on to the receiving CE's PE for processing.

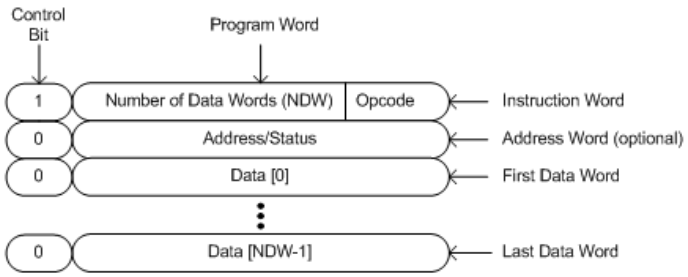

Fig. 4. SIMPPL packet format.

3.2 The SIMPPL Repeater

SIMPPL was originally developed as an architectural framework for SoCs. Our recent work introduced the SIMPPL Repeater to extend the SIMPPL model to support systems implemented on multi-FPGA platform's as well as SoCs [Dickin and Shannon 2008]. The SIMPPL Repeater is used to allow communication between CE's located on different FPGAs. The SIMPPL Repeater is necessary due to how the original SIMPPL model handles an IC's I/O. The I/O ports, and anything outside the SoC connected to these ports, are modelled as being part of a PE in SIMPPL. Thus, two CEs implemented on separate FPGAs can communicate if the communication channel logic is included in both PEs. Figure 5 illustrates this configuration. However, all inter-CE communication is to be done through SIMPPL Controllers. Having CEs communicate without using a SIMPPL Controller negates the reduced integration time and IP reuse benefits of the SIMPPL Model.

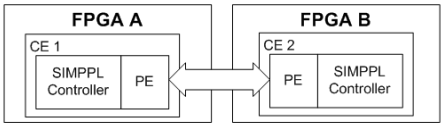

Fig. 5. Inter-FPGA communication between CEs without a SIMPPL Repeater. This approach does not follow the SIMPPL model for inter-CE communication.

To get around the inter-IC communication problem the SIMPPL Repeater was developed. The SIMPPL Repeater replaces the SIMPPL Controller and SCS of a special CE whose purpose is to provide inter-FPGA communication. The PE of this special CE is the I/O ports and associated logic of an inter-FPGA communication channel. The SIMPPL Repeater is essentially FIFO read/write logic that is compatible with the SIMPPL communication protocol. The SIMPPL Repeater can also contain logic to control the inter-FGPA communication PE along with any logic needed to convert the SIMPPL packet data into a suitable format for I/O. Figure 6 illustrates inter-FPGA communication using SIMPPL Repeaters. In previous work, SIMPPL Repeaters were used in tandem with Multi-Gigabit Transceivers (MGTs) to demonstrate a communication channel between two FPGAs [Dickin and Shannon 2008].

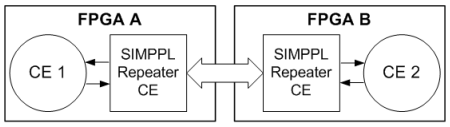

Fig. 6. Inter-FPGA communication between CEs using SIMPPL Repeaters.

Unlike a SIMPPL Controller, a SIMPPL Repeater will not execute the opcode contained in the instruction word of a received SIMPPL packet. This behaviour is

desirable because it allows CEs to be oblivious to whether or not the packets they send are transmitted off-chip. This would not be the case if a SIMPPL Controller was used to control the inter-FPGA communication PE instead of the SIMPPL Repeater. The addition of SIMPPL Repeaters transparently extends the SIMPPL model to include multi-FPGA platforms.

4. CASE STUDY

The following case study, concerning the acceleration of a seismic imaging application, is to demonstrate the use of the extended SIMPPL model with a system implemented on a multi-FPGA platform. The inter-FPGA communication CE used here is not as complicated as the MGT CE used in the previous proof-of-concept system [Dickin and Shannon 2008]. However, this time we are using the SIMPPL Repeater in a system with a legitimate application and five FPGAs as opposed to two.

4.1 The Seismic Application

The purpose of seismic imaging is to obtain an extrapolated wavefield along a planar cross-section of the earth's surface, taken from the surface, to produce an image of underground bodies. Data is taken at multiple points along the earth's surface, where the depth of a body determines the time it takes for a lected wave to return to the same point. An underground body is then extrapolated using a fourier integral operator (FIO) kernel. Figure 7 illustrates an example cross section used to extrapolate the data.

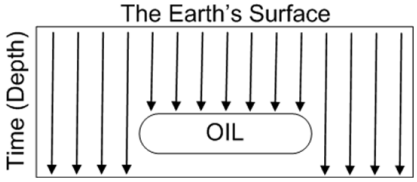

Fig. 7. Cross Section of Earth's Surface.

Our prototype employs a depth stepping method as described by Margrave et al [9]. The mathematical equation of interest is given in equation 1 for the 1-D case.

$$\Psi(x, z + \Delta z, \omega) = \frac{1}{2\pi} \int_{k_{start}}^{k_{end}} \Psi'(k, z, \omega) \times e^{if(x,\omega,k)\Delta z} e^{-izk} dk \qquad (1)$$

where $\omega_{low} < \omega < \omega_{high}$.

$\Psi(x, z + \Delta(z), \omega)$ is the extrapolated wavefield at a depth $\Delta(z)$ below the surface, $\Psi'(k, z, \omega)$ is the measured wavefield at the surface after Fourier transforming along both spatial and temporal coordinates. The function, $f(x, \omega, k)$, is the phase-shift function over a subset of the entire input frequency range forming a window

constrained by ω_{low} and ω_{high}, where outside of this range the frequencies are ignored. These window bounds apply to Equations 2 through 5, but they are omitted from the notation for clarity. Equation 1 can be written in its discrete form after rearranging the terms for simplicity as,

$$\Psi(x, z + \Delta z, \omega) = \frac{1}{2\pi} \sum_{k_0}^{k_{N-1}} \Psi'(k, z, \omega) \times W(x, \omega, k) \tag{2}$$

where,

$$W(x, \omega, k) = e^{if(x,\omega,k)\Delta z} e^{-izk} \Delta k \tag{3}$$

In addition, the wavenumbers, k(j), are found using

$$k(j) = 2\pi \left(\frac{j}{N-1} \right) k_s \qquad j \leq \frac{N-1}{2} + 1 \tag{4}$$

$$k(j) = 2\pi \left(\frac{j - \frac{N-1}{2}}{N-1} \right) k_s - \frac{k_s}{2} \qquad j > \frac{N-1}{2} + 1 \tag{5}$$

where $j = 0...N$

The computational elements of the equation are divided into as many different blocks as possible in order to exploit pipelining and parallel computation to greatly speed up the calculation of a 1-D slice. Theore, Equation 2 was divided into the following blocks for implementation in hardware:

—FFT in spatial domain

—FFT in temporal domain

—Phase matrix generation

—Integration (multiplication and summation)

In our implementation, the algorithm is divided into four major Computing Elements (CEs) for the calculation of the single cross section shown in Figure 7 illustrates the data flow of the algorithm, highlighting these CEs.

The function of each CE in the FIO kernel can be summarized as follows:

—1024 Fast Fourier Transform(FFT): Perform FFTs in the spatial domain

—256 FFT: Perform FFTs in the temporal domain

—Complex Matrix Multiply(CMM): Perform a matrix multiplication on the transformed cross section

—1024 Inverse Fast Fourier Transform(IFFT): Perform inverse FFTs back in to the spatial domain

The phase matrix erred to in the CMM is generated by complex square roots and exponential operations, which are extremely resource intensive. Since this matrix is independent of the data, it can be pre-calculated in advance based on the appropriate velocity model and used repetitively for different sets of data. However, due to the size of the matrix, the entire matrix cannot be stored on-chip and must be buffered from external memory in sections. The actual numerical processing of the transformed data occurs in the complex matrix multiply when the extrapolated data vectors are multiplied with the buffered segments of the phase matrix.

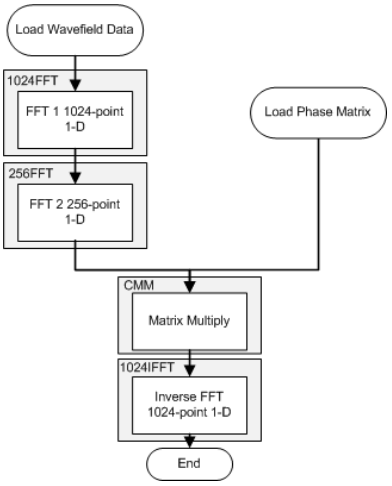

Fig. 8. FIO data Flow.

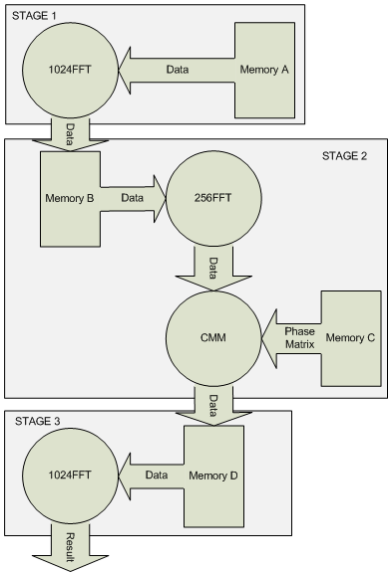

Fig. 9. FIO System block diagram.

4.2 The System Level Implementation

The seismic imaging application can be easily divided to increase the throughput of our overall system. Theore, the algorithm was divided into three separate stages that can be executed in parallel. Figure 9 illustrates the FIO system block diagram partitioned into stages.

Using the BEE2 platform, the flow of data is transmitted between CEs in the multi-FPGA platform. Each stage occupies a separate FPGA on the BEE2 platform. The specific division of stages on FPGAs will be explained in Section 4.3. The architecture consists of three distinct stages. Stage 1 transforms the input data in the spatial domain using the 1024 FFT. The 256 FFT and the CMM make up Stage 2, performing a temporal FFT and processing it by performing a matrix multiply with the phase matrix. The 256 FFT and CMM are grouped together in the same stage since both CEs process the same row of data in the matrix, whereas other stages operate on columns of data. Finally, Stage 3 uses the 1024 IFFT to perform an inverse spatial FFT to provide the final result.

A sample data set consists of 1024 surface sample points (columns) in the spatial domain, which represents the earth's surface. Each column has 256 data samples (rows), representing varying sample times (depth) at each surface point. Initially, the data set is loaded from Memory A and transformed in the spatial domain, row by row, for all time (depth) measurements via the 1024FFT. This FFT must be performed on all rows of the entire matrix before the Stage 2 can begin processing. As each row is completed, the resultant is transferred to Stage 2 and stored in Memory B. When the last 1024-point FFT is completed, the first stage is finished and the second stage can initiate while Stage 1 begins processing a new data set. Stage 3, which consists of 1024IFFT, mirrors Stage 1 in operation. It performs an

inverse FFT along each row in the same fashion once a complete data set has been processed in Stage 2.

When Stage 2 initiates, the 256FFT transforms the data along each column of data in the temporal domain. As each column is completed, the resultant is transferred to the CMM for the matrix multiply in order to allow the 256FFT to continue processing temporal FFTs in parallel. The CMM takes the 256 points and multiplies it with the buffered 256x256 phase matrix loaded from Memory C. The resultant is then transferred to Stage 3 and stored in Memory D. Since the system is split into three stages, the system can act as a three stage pipeline, processing three sets of data in parallel.

4.3 Multi-FPGA System Partitioning

The FIO system is partitioned across the four User FPGAs of the BEE2 platform, as in the our previous work, due to the pipelined nature of the algorithm and the large resource usage of our CEs [Lee et al. 2008]. Block diagrams of the User FPGA sub-systems are shown in Figures 10 through 12. All CEs operate using a 100MHz clock, with the exception of the DDR2 Memory Controller, which operates at 200MHz.

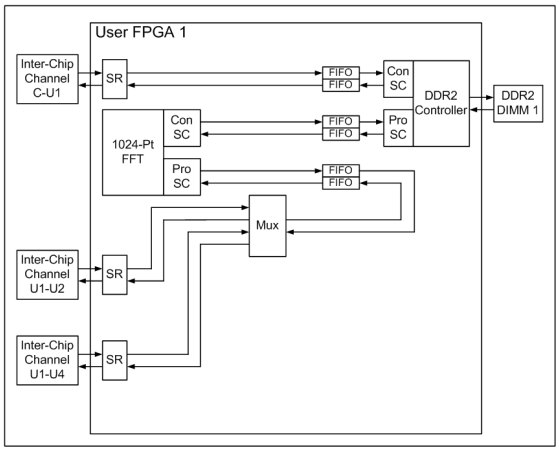

Fig. 10. Block diagram of User FPGA 1.

The FFT_1024 CE is implemented on User FPGA 1, as seen in Figure 10, along with two DDR2 Memory Controllers and three inter-FPGA communication channels. Wavefield data is transmitted to User FPGA 1 from the Control FPGA via the C-U1 channel. The data is written to DIMM 1 until the FFT_1024 is ready to process it. After the FFT_1024 processing is completed, the data is transmitted to either User FPGA 2 or 4 using the appropriate channel. The multiplexor block of Figure 10 alternates sending matrices of data to User FPGA 2 and 4.

Figure 11 shows the CE's implemented on User FPGA 2 and 4. These FPGAs have identical subsystems. The FFT_256 and the CMM CEs are implemented in

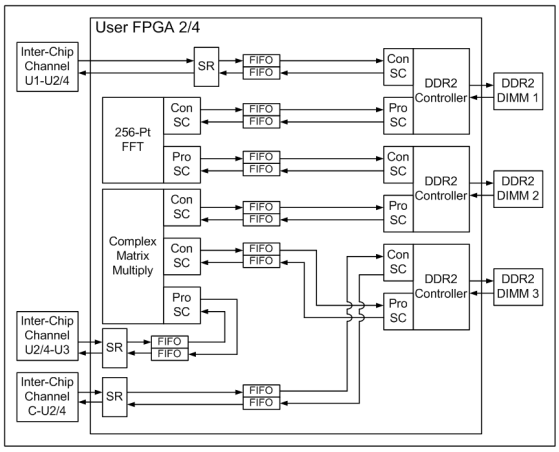

Fig. 11. Block diagram of User FPGA 2 and 4.

this subsystem. This subsystem is duplicated to include two CMM CEs in the system and exploit parallelism to increase throughput. The CMM is the bottleneck of our system, which is explained in more detail in Section 5.

Data is received from User FPGA 1 and buffered in DIMM 1. The FFT_256 CE reads and processes the DIMM 1 data and stores the result in DIMM 2. Once an entire matrix of data has been written to DIMM 2, the CMM CE begins computation. The CMM also reads in the phase matrix from DIMM 3. This phase matrix is pre-loaded into DIMM 3 by the Control FPGA via the inter-FPGA communication channel before the computation begins. All the CMM results are transmitted to User FPGA 3.

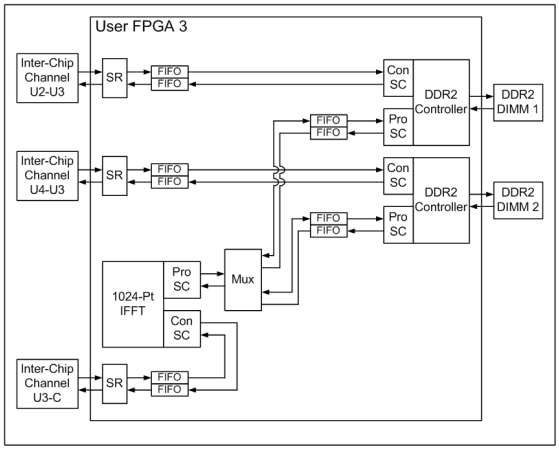

Fig. 12. Block diagram of User FPGA 3.

The final CE, the IFFT_1024, is implemented on User FPGA 3. Data is received from both User FPGA 2 and 4 and buffered in DIMM 1 and 2 respectively. The IFFT_1024 CE alternates reading matrices of data from these two DIMMs. The output is transmitted to the Control FPGA.

4.4 Individual Component Architecture

Implementing applications utilizing floating point arithmetic on FPGAs presents difficult design challenges as basic operations require significant resources. Table I shows some of the resource requirements for performing complex single precision floating point operations and complex 32-bit fixed point operations obtained from Xilinx data sheets [17][18][19].

Operation	Floating Point		Fixed Point	
	FFs	LUTs	FFs	LUTs
Add	1182	1160	116	106
Subtract	1182	1160	116	106
Multiply	1732	1630	170	154

Table I. Arithmetic operation resource usage

Based on these results, the parallelism of computing elements is limited by the resources available on an FPGA. For example, an FPGA with 100,000 Flip-Flops (FFs) could only implement 57 complex floating point multipliers. In order to mitigate the effects of resource usage, designs must utilize a minimum number of operations to complete its operation. The following sections describe the design of the floating point computing elements.

4.4.1 *Fast Fourier Transform.* The Cooley-Tukey algorithm [20] is commonly used to perform a FFT. Many readily available FFT cores already employ this algorithm including Xilinx's block floating point FFT. This algorithm partitions the Fourier transform into smaller operations called butterflies. Each butterfly consists of a radix-2 decimation-in-time (DIT) operation and is repeated to perform a complete FFT. Figure 13 illustrates a single butterfly in an FFT.

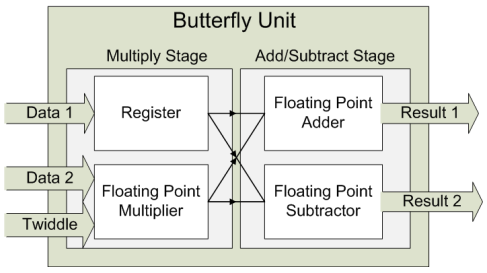

Fig. 13. FFT butterfly.

Each butterfly takes two points of data, multiplies one by a twiddle factor, and subsequently adds and subtracts this value from the initial points of data. The

twiddle factor varies depending on the stage of operation in the FFT and utilizing these different twiddle factors and butterflies, a complete FFT can be computed. To achieve the highest throughput for an 8-point FFT, the design is pipelined into three separate stages of butterflies with each stage consisting of 4 butterflies. The design is shown in Figure 14.

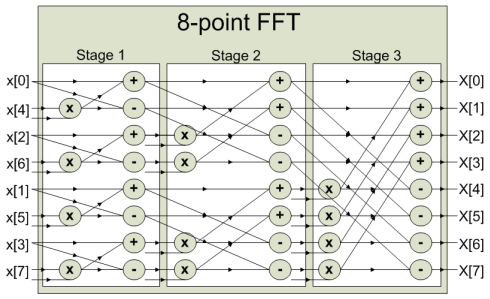

Fig. 14. FFT Pipeline Stages.

The seismic imaging application requires two separate FFTs of at least 1024 and 256-points. When designing these FFTS, the primary concern was the amount of resources a fully-pipelined FFT would require. One butterfly utilized a floating point multiply, subtract, and add, which totaled approximately 4,000 FFs based on the resource usages shown in Table I. As an FFT grows in size, the number of butterflies and thus resource requirements scale at a log-linear rate. For example, a 256-point FFT requires 8 stages of 128 butterflies. If this were to be implemented, the design would require over 4 million FFs, exceeding the total resources available on all commercial FPGAs.

In order to implement a large FFT on a smaller FPGA, the CE is designed to use a fixed bank of butterflies. This is beneficial since it allows the FFT CE to compute large FFTs while minimizing resource utilization. Figure 15 shows the designed architecture for the FFTs.

Similar to Stage 1 of the 8-point FFT shown in Figure 14, the butterfly bank contained a fixed number of butterflies. By using a fixed number of butterflies, resource usage can be minimized, but throughput is decreased when compared to a fully pipelined FFT. However, utilizing a fixed butterfly bank makes large FFTs possible. Similar to Stage 1 of the FFT shown in Figure 14, the butterfly bank contains an array of butterflies. While throughput is decreased in comparison to the fully pipelined FFT given in Figure 5, a fixed butterfly bank makes large FFTs possible.

Utilizing a SIMPPL consumer, data is initially transferred into the FFT and loaded into local memory (BRAM). Once the entire data set is loaded, the FFT can start its computation. The loader takes the data-set incrementally and performs each stage of operation by looping the data through the butterfly bank. The unloader takes the result and loops the data back to the memory block. The loader then repeats the process for a given number of iterations. For example, a 1024-point

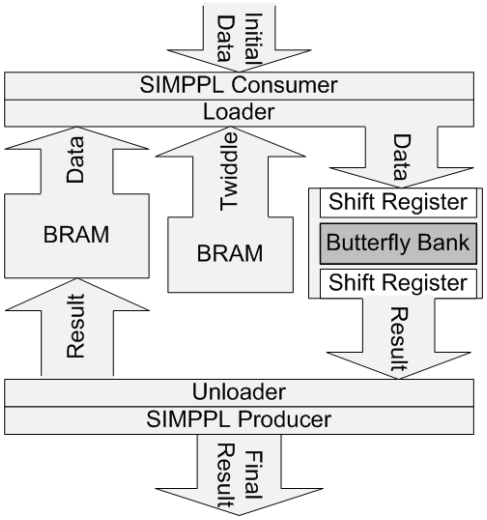

Fig. 15. FFT block diagram.

FFT normally requires 4608 butterflies. If the butterfly bank contains 8 butterflies, the data would loop through the bank 576 times. Once the FFT finishes computation, the SIMPPL producer transmits the result to the next computing element.

As previously described, the number of butterflies grows as a log-linear rate as the length of the FFT increases. Since our design uses a fixed number of butterflies, the latency of our FFT also increases at a log-linear rate as the length of the FFT increases. This is because the number of iterations through the butterfly bank is directly proportional to the total number of butterflies required to compute a fully pipelined FFT. However, simply increasing the number of butterfly banks does not necessarily result in a decrease in latency. The bottleneck in the design is the computational stage that requires the longest latency. This could be the shift registers or the butterfly bank. Increasing the number of butterfly banks would lead to longer latencies in the shift register as more data is required to compute one iteration. Thus, adding butterfly banks does not have a direct improvement on performance. Because of this, to improve throughput in the overall system, it is better to compute multiple FFTs in parallel by implementing multiple FFT CEs.

For our design, the butterfly bank contains two butterflies. In our previous work, the butterfly bank included four butterflies. Since our previous goal was to optimize the maximum operating frequency, the floating point multipliers, adders, and subtractors incurred long latencies (21 cycles) but had a high maximum operating frequency of 195MHz. In order to balance this, we utilized four butterfly banks as it took 24 cycles to load 8 points of data plus 4 twiddle factors. However, in our new implementation, each CE operates at 100MHz. Theore, we are able to lower the latency of the multipliers, adders, and subtractors to 6 cycles. Furthermore, we improved the loader and unloader in the FFT to allow 4 data points and 2 twiddle factors to be loaded in 6 cycles. Thus our current design only uses two butterflies.

Table II shows the resource utilization of our current FFT in comparison to our previous design.

	New 1024FFT	New 256FFT	Old 1024FFT	Old 256FFT
LUTs	10949	11243	18170	18170
FFs	6061	6128	20387	20387

Table II. FFT Resource Usage

As seen in Table II, the new FFT design utilizes approximately 50% of the old FFT's resource utilization. This is due to the reduction in butterfly banks from four to two. Furthermore, the 1024FFT utilizes approximately the same resource usage as the 256FFT. Since our design uses an iterative approach in calculating an FFT, only the control logic needs to be changed to scale the size of our FFT. A longer FFT will be iterated more times through the same butterfly bank as a shorter FFT, thus the resource utilization remains constant.

4.4.2 *Complex Matrix Multiply.* In order to process the transformed data with the phase matrix, the Complex Matrix Multiply (CMM) is used. Figure 16 illustrates the architecture that was designed.

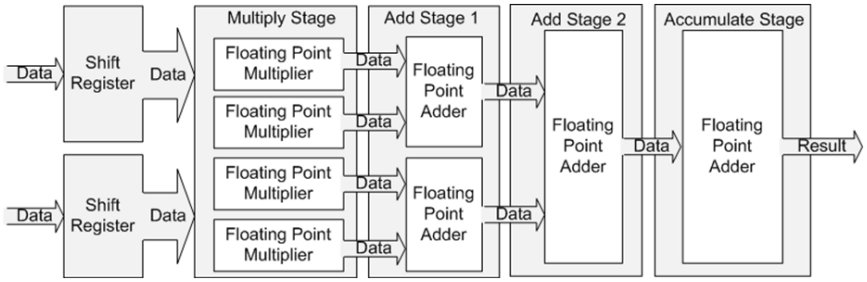

Fig. 16. CMM block diagram.

The CMM contains a SIMPPL producer, SIMPPL consumer, two shift registers, a stage of multipliers, multiple stages of adders, and an accumulation stage. The The number of adder stages depends on how many data points are calculated at a given time. Determining the optimal number of data points used for a CMM is presented in a later section. The purpose of the CE is to take a column of data produced by the 256-point FFT and multiply it by a 256x256 phase matrix.

The CMM multiplies a segment of the 256-point temporal FFT vector by a segment of the phase matrix and accumulates the results. The final accumulation stage sums the results over the number of temporal data points (e.g. 256). For example, if a 256x256 multiply needs to be performed, and the core calculates 8x8 points at a time, the core iterates 32 times and the accumulation stage adds the total from each iteration to output the final result.

Much like the FFT, increasing the number of multipliers does not necessarily lead to a direct improvement in performance. Increasing the number of multipliers result in the shift registers incurring additional latencies as more data is required to

saturate the multipliers. Thus a balance between the multiplier latency and shift register latency is the best performing design. Theore, the best method to increase throughput of the entire system is to implement multiple CMMs in parallel to calculate different rows of data.

For our CMM design, two multipliers are used with two adder stages and one accumulation stage. Our previous design utilized four multipliers with three adder stages and one accumulation stage. Much like our FFT design, the reduction in multipliers is due to a shorter latency for each multiplier. Table III shows the resource utilization of our current FFT in comparison to our previous design. Our current design utilizes approximately 50% less resources in comparison to our previous design. This is due to the reduction in multipliers from four to two, and the elimination of one adder stage.

	New CMM	Old CMM
LUTs	13348	24672
FFs	7812	26643

Table III. FFT Resource Usage

4.5 DDR2 Memory Controller

The Memory CE is composed of two SIMPPL Controllers, a DDR2 memory controller [erence berkeley people], and an asyncronous interface between the SIMPPL Controllers and the memory controller. Each of the memory CEs controls access to a 1GB DIMM of memory. Each FPGA on the BEE2 can have up to 4 of these memory CEs to access the 4 DIMMs of DDR2 SDRAM available. In the memory CE, one of the SIMPPL Controllers is a Producer, while the other is a Consumer. The Producer is used exclusively for read operations and the Consumer only supports write commands. This allows two CEs to connect to a DDR2 memory module, one to write and the other to read. This setup works well with our system design as the DDR2 DIMMs are used to buffer our rather large data sets in between computation-oriented CEs.

The DDR2 memory controller operates in the 200 MHz clock domain, while the SIMPPL Controllers and the rest of the system operate in the 100 MHz clock domain. Initially in the design phase we planned to have the whole system operate at 200 MHz as well. However, the DDR2 memory controller only marginally passes timing requirements at the best of times, and adding the SIMPPL Controllers resulted in unroutable build errors when building the FPGA bitstreams. Asyncronous FIFOs were added between the SIMPPL Controllers and the DDR2 memory controller to eliminate the problem.

4.5.1 *Inter-FPGA Communication CE.* The Inter-FPGA Communication CE consists of a LVDS inter-FPGA communication channel core provided by Berkeley and a SIMPPL Repeater. The channels between User FPGAs on the BEE2 are configured to be 64-bits wide and full duplex. Between the Control FPGA and the User FPGA, the channel width is only 16-bits. The channels operate at 100 MHz, the same frequency as the on-chip system, and so introduce a very small latency when data is transferred between chips. Based on our measurements, it typically

takes 4 cycles to transfer a 64-bit word between User FPGAs. If data is streamed between chips, as in the FIO kernel application, then this time becomes negligible.

4.5.2 *FIFOs.* The FIFO's used in the system are synchronous, first-word-fall-through, 256-bits wide, and 16-words deep. In addition to the 256-bit word size, there is an extra "control bit". This bit is used in the SIMPPL communication protocol and makes the actual size of the FIFO 257-bits.

5. RESULTS

This section explains how the performance of the FIO kernel acceleration system was measured. Section 0**??** describes the performance analysis done for each of the computation CEs and Section 5.2 presents the results of our system.

5.1 CE Performance

The latency of the individual computation CEs is listed in Table IV. This measurement is the amount of time each CE requires to process an entire matrix (1024x256) of data, including the DDR2 memory storage and retrival overhead and inter-CE communication. Interchip communication latencies are not included in this measurement as each core was evaluated with a test setup on a single FPGA. However, since the inter-FPGA communication channels are operating at the same frequency as the computation CE's (100MHz), the latency of transmitting a word of data between User FPGAs is only a few cycles and has a negligible impact on overall system latency. For comparision, the latency of the CEs from our previous work is included [Lee et al. 2008].

Table IV. CE Performance

CE	Latency (ms)	Previous Latency (ms)
FFT_256	0.0822	7.94
FFT_1024/IFFT_1024 no reorder	0.394	37.9
FFT_1024/IFFT_1024 with reorder	5.59	N/A
CMM	31.5	172

Inspection of Table IV shows that the CMM CE has the largest latency of all CEs in our system. This was true in our previous version of the system as well. Therefore, the CMM is the bottleneck restricting performance. To improve performance, the FFT_256 and CMM CEs are duplicated in our design on User FPGA 2 and 4 to allow two matrices to be processed in parallel. This was done in our previous work as well. A new feature added in this work to improve performance, is the data matrices for the CMM are stored in DDR2 memory such that the CMM can read and write data using sequential addresses. Accesses using sequential addresses reduces memory access time overhead. The data is stored sequentially for the CMM by the FFT_1024 and IFFT_1024 "with reorder" CEs. These two CEs access the memory shared with the CMM with a non-sequential addressing pattern to set the data up correctly for the CMM. This increases the latency of the FFT_1024 and IFFT_1024 CEs by an order of magnitude. The "no reorder" entry in Table IV shows the FFT_1024 and IFFT_1024 CE's performance with sequential addressing.

This increase in latency is hidden by the CMM's latency when examining system performance.

5.2 System Performance

The total latency of the system is measured[1] and compared with the equivalent vector optimized Matlab algorithm run in single precision floating point. Table V compares the overall performance of our updated system with a software implementation of the FIO kernel and our previous work. The software algorithm is run on a PC with a 2.13GHz Intel Core 2 Duo and 4GB of RAM. Running the FIO kernel on the BEE2 platform results in 357 times the throughput of the Matlab version. The updated FIO kernel system has 29 times the throughput of the old system. This increase is due to the enhancements outlined in this paper, particularly the use of a dedicated DDR2 memory controller.

Table V. FIO kernel performance comparison

Platform	Latency(s)	Throughput (matrices/s)	Speedup Compared to Software
New BEE2 SIMPPL System	0.0427	63.6	357
New BEE3 SIMPPL System	Doesn't	Exist	M'kay
Old BEE2 System (195 MHz)	0.55	2.2	12.3
Old BEE3 System (248 MHz)	0.43	2.8	15.7
Software	5.62	0.178	1.0

6. CONCLUSIONS AND FUTURE WORK

This paper presents the prototype for an application-specific architecture of an FIO kernel. The system design utilized the SIMPPL architectural framework that was recently expanded for use with multi-FPGA systems. In addition, a detailed study on floating point CEs, particularly the FFT CE, was performed. The system runs on the BEE2 platform and achieves a data throughput 357 times that of an optimized software implementation.

Future work would include porting the application to a more modern multi-FPGA platform. The Xilinx Virtex II Pro FPGA that inhabits the BEE2 is becoming dated, and a new version of the platform, the BEE3 has been developed that utilized Virtex 5 FPGAs. We would expect to see significant performance gains if the system were able to operate at the higher frequencies that a newer FPGA would allow. Furthermore, significant resource usage and latency is incurred due to the normalization and denormalization of floating point numbers. We will also investigate internal representations of floating point input data within the FIO kernel that reduce or limit these operations, such as block floating point, to further decrease

[1]The individual CEs have been mapped onto the BEE2 platform and have been verified to run on the Control FPGA and communicate with the DDR2 DIMMs. However, there have been problems loading the final design onto the entire BEE2 platform due to a lack of debugging facilities on the User FPGAs and problems with data integrity on the inter-FPGA communication channels. System performance results found in Table V.

latency. However, it will be necessary to verify that the required computational precision is maintained before these representations can be employed.

REFERENCES

Brodersen, R., Chang, C., Wawrzynek, J., and Werthimer, D. BEE2: a multi-purpose computing platform for radio telescope signal processing applications. *EECS, University of California, Berkeley, Space Science Laboratory, SETI Institue, http://bee2. eecs. berkeley. edu/papers/BEE2_radio_telescope_whitepaper. pdf*.

Chang, C., Wawrzynek, J., and Brodersen, R. W. 2005. BEE2: a high-end reconfigurable computing system. *IEEE Design and Test of Computers 22,* 2 (March), 114–125.

Dickin, D. and Shannon, L. 2008. Extending the SIMPPL SoC architectural framework to support application-specific architectures on multi-FPGA platforms. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*. 67–72.

Greenbaum, J. 2002. Reconfigurable logic in SoC systems. In *Proceedings of the IEEE Custom Integrated Circuits Conference*. 5–8.

He, C., Lu, M., and Sun, C. 2004. Accelerating seismic migration using FPGA-based coprocessor platform. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. 207–216.

Lee, J., Shannon, L., Yedlin, M. J., and Margrave, G. F. 2008. A multi-FPGA application-specific architecture for accelerating a floating point Fourier Integral Operator. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*. 67–72.

Ligon III, W. B., McMillan, S., Monn, G., Schoonover, K., Stivers, F., and Underwood, K. D. 1998. A re-evaluation of the practicality of floating point operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*. 206–215.

Morris, G. R. and Prasanna, V. K. 2005. An FPGA-based floating-point jacobi iterative solver. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*. 420–427.

Scrofano, R. and Prasanna, V. K. 2006. Preliminary investigation of advanced electrostatics in molecular dynamics on reconfigurable computers. In *Proceedings of ACM/IEEE Supercomputing Conference*. 45.

Shannon, L. and Chow, P. 2007. SIMPPL: an adaptable SoC framework using a programmable controller IP interface to facilitate design reuse. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 15,* 4, 377–390.

Tkachenko, A., Cabric, D., and Brodersen, R. 2006. Cognitive Radio Experiments using Reconfigurable BEE2. In *Asilomar Conference on Signals, Systems, and Computers*.

Underwood, K. D. and Hemmert, K. S. 2004. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the IEEE Symposium of Field Programmable Custom Computing Machines*. 219–228.

Varghese, J., Butts, M., and Batcheller, J. 1993. An efficient logic emulation system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 1,* 2 (June), 171–174.

Xilinx Inc. MicroBlaze Procesor Reference Guide. `http://www.xilinx.com`.