SANDIA REPORT

SAND2006-6888
Unlimited Release
Printed November 2006

FPGAs in High Performance Computing: Results from Two LDRD Projects

K. Scott Hemmert, Keith D. Underwood, Craig D. Ulmer, David C. Thompson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories

SAND2006-6888
Unlimited Release
Printed November 2006

FPGAs in High Performance Computing: Results from Two LDRD Projects

K. Scott Hemmert
Keith D. Underwood
Scalable Computing Systems Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
kshemme@sandia.gov
kdunder@sandia.gov

Craig D. Ulmer
David C. Thompson
Visualization and Scientific Computing Department
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94551-0969
cdulmer@sandia.gov
dcthomp@sandia.gov

Abstract

Field programmable gate arrays (FPGAs) have been used as alternative computational devices for over a decade; however, they have not been used for traditional scientific computing due to their perceived lack of floating-point performance. In recent years, there has been a surge of interest in alternatives to traditional microprocessors for high performance computing. Sandia National Labs began two projects to determine whether FPGAs would be a suitable alternative to microprocessors for high performance scientific computing and, if so, how they should be integrated into the system. We present results that indicate that FPGAs could have a significant impact on future systems. FPGAs have the *potential* to have order of magnitude levels of performance wins on several key algorithms; however, there are serious questions as to whether the system integration challenge can be met. Furthermore, there remain challenges in FPGA programming and system level reliability when using FPGA devices.

Acknowledgment

Arun Rodrigues provided valuable support and assistance in the use of the Structural Simulation Toolkit within an FPGA context. Curtis Janssen and Steve Plimpton provided valuable insights into the workings of two Sandia applications (MPQC and LAMMPS, respectively).

Contents

Preface 14

Summary 15

Nomenclature 16

1 Introduction 17

2 Related Work 21

Floating-Point Algorithms 21

3 High Performance Floating-Point Modules 23

General Implications for Floating-Point Implementations 23

Floating-Point Unit Implementation 24

General Considerations 24

Adder Implementation 25

Multiplier Implementation 27

Characteristics 29

4 Implications of Trends in FPGA Performance 31

Trends in FPGA Performance 31

Addition 32

Multiplication 33

Division 33

Multiply Accumulate 35

Analysis 35

Analysis Retrospective 37

Kernels Explored in Simulation 38

Matrix Multiply 38

FPGA Implementation 40

Performance Comparisons 41

Memory Requirements 42

Matrix-Vector Multiply 43

FPGA Implementation 43

Performance Comparisons 46

Fast Fourier Transform (FFT) 47

Parallel FFT Implementation 49

Pipelined FFT Implementation 50

Parallel-Pipelined FFT Implementation 52

Comparison of Architectures 53

FPGAs vs. CPUs 54

Analysis Summary 56

5 Implementations on a Real World System **57**

The Cray XD1 57

Development Components 58

CPU-FPGA Data Transfers in the XD1 59

Application Examples 60

k-Nearest Neighbors (kNN) 62

Stiffness Matrix (STIFF) 66

Ray-Triangle Intersection (RTI) 70

Isosurfacing (ISO) 75

Application Summary 79

XD1 Observations 80

6 System Simulation **83**

Motivation and Methodology 83

Motivating Applications 83

Methodology 84

Cray XD1 84

Pentium-4 Xeon Workstation 85

Structural Simulation Toolkit 86

Benchmarks 86

Approach 86

Measurements from the XD1 88

Simulation Results 90

Summary 93

7 Challenges to Deploying FPGAs **95**

Algorithms 95

Programmability 95

Portability 96

Architecture 96

Cost 96

Reliability 97

8 Conclusions **99**

References **100**

List of Figures

1.1 A high-level representation of an FPGA 18

3.1 IEEE double precision floating-point format 23

3.2 General computation layout for the floating-point adder 26

3.3 General computation layout for the floating-point multiplier 28

4.1 Double precision floating-point performance 34

4.2 Floating-point double precision multiply accumulate performance 35

4.3 Basic FPGA property trends 36

4.4 Maximum achievable performance versus memory bandwidth and matrix size 39

4.5 Matrix multiply implementation 40

4.6 A comparison of double precision floating-point matrix multiplication performance on CPUs, FPGAs, and RC platforms 41

4.7 Matrix multiply internal memory analysis 43

4.8 Maximum sustainable matrix vector multiply performance versus memory bandwidth 44

4.9 Multiply-accumulate unit 45

4.10 Matrix vector multiplication implementation 45

4.11 A comparison of double precision floating-point matrix-vector multiplication performance on CPUs, FPGAs, and RC platforms 46

4.12 Basic butterfly operation and hardware implementation 48

4.13 Two variations of 8-point, radix-2 FFTs 48

4.14 Architecture of the Parallel FFT 50

4.15 Architecture of the Pipelined FFT 51

4.16 Architecture of the Parallel-Pipelined FFT 52

4.17 A comparison of the performance and memory requirements of FFT implementations 53

4.18 FPGAs vs. CPUs for FFT on current and projected technologies 55

5.1 The Cray XD1 Architecture 58

5.2 Data transfer characteristics of the XD1 60

5.3 The pseudo-code for the kNN algorithm. 62

5.4 Architecture of the kNN Hardware Accelerator 64

5.5 The pseudo-code for the modified kNN algorithm. 64

5.6 Performance gain of FPGA accelerator for kNN. 65

5.7 Pseudo-code for the stiffness matrix computation at a single mesh point. Matrix indices are omitted in this listing for brevity. 67

5.8 Architecture of the STIFF Hardware Accelerator 69

5.9 Pseudo-code for the Möller-Trumbore Ray-Triangle Intersection (RTI) Algorithm 70

5.10 Pseudo-code for RTI Work in a Photon Mapping Application. 71

5.11 Architecture of the Ray-Triangle Intersection hardware accelerator 73

5.12 Speedup of the RTI Hardware Accelerator over Software 74

5.13 Exterior and interior views of a hand by isosurfacing 75

5.14 Rather than processing an entire cube at a time, the isosurfacing accelerator processes slices of data. 77

5.15 Architecture of the isosurfacing accelerator. 78

6.1 The XD1 architecture 85

6.2 The traditional (a) and modified (b) benchmarks 86

6.3 Blocking vs. Non-blocking approaches 87

6.4 Timeline for blocking and non-blocking operations 87

6.5 Buffer approaches for FFT and DGEMM 88

6.6 XD1 performance comparison for FFT and DGEMM 89

6.7 Validation of the simulation models for **(a)** FFT and **(b)** DGEMM 90

6.8 Simulated impact of bandwidths for FFT and DGEMM 92

6.9 Simulated impact of blocking operations for FFT and DGEMM 92

List of Tables

3.1 Special values in the IEEE 754 format 24

3.2 Original Floating-point Units versus Optimized Units (size in slices, latency in cycles and speed in MHz) 25

3.3 Floating-point Unit Characteristics (size in slices, latency in cycles and speed in MHz) 30

4.1 Parts used for performance comparison 32

4.2 Parallel pipeline shapes 54

4.3 Memory requirements summary 54

5.1 Characteristics of the four XD1 accelerator designs 79

6.1 Typical system parameters 84

Preface

As requested by the IAT, this final report combines the results and experiences from two LDRD projects exploring the potentials of FPGAs for impacting high performance computing. The two projects (High Performance Processing architecture (67016) and Enhancing Simulation Performance on Clusters with Configurable Auxiliary Devices (67018)) began as independent efforts, but converged in their final year. The different approaches may be apparent throughout the document, but we have attempted to merge the results to present a cohesive conclusion.

Summary

Field-programmable gate arrays (FPGAs) are a technology that has received a lot of attention in the high performance computing (HPC) community over the last few years. While FPGAs are widely accepted to offer a tremendous value in metrics such as performance and performance/power for a significant class of applications, they have not traditionally been used for the scientific computing applications common on most HPC platforms. In early literature, results indicated that FPGAs were a poor match to floating-point arithmetic — and this view was still prevalent in the fall of 2003 when Sandia National Labs started two projects to explore the use of FPGAs in HPC environments.

In New Mexico, the “High Performance Processing Architecture” project began exploring the capabilities of FPGAs to support double precision floating-point arithmetic. Our goal was to determine what levels of peak performance could be achieved on an FPGA and to determine how common scientific applications would map to FPGAs. We also had a stated objective of determining how the FPGAs could be integrated into the system.

In California, the “Enhancing Simulation Performance on Clusters with Configurable Auxiliary Devices” project focused on communication challenges associated with integrating accelerators into modern HPC systems. This effort focused on both network-based approaches (i.e., connecting an FPGA to the system’s network fabric) and localized approaches (i.e., connecting an accelerator to the host’s I/O subsystem). Multiple applications were adapted to utilize the FPGA resources in order to gain a better understanding of what can be achieved with current generation technology.

Our results indicate that FPGAs *could* have a dramatic advantage (10$\times$ within the decade) in peak floating-point performance — if the FPGA vendors decide to focus on the HPC market. These advantages translate into an even greater advantage in sustained performance because the FPGA system can be engineered to deliver dramatically more memory bandwidth than conventional processors will ever deliver.

Unfortunately, there remain a number of challenges to deploying FPGAs that have not yet been solved. FPGAs are still remarkably difficult to program and are unlikely to execute fully general purpose code in the near future. Thus, FPGAs must be integrated into a system along with a conventional microprocessor. Our results suggest that this system integration is a particular challenge as FPGAs of the future will need a tight coupling with the processor through a high bandwidth interface. Indeed, the bandwidth requirements could become so great as to become unachievable. Finally, major reliability issues remain with single-event upsets (SEU) that preclude a large scale deployment of FPGAs in the near term.

Nomenclature

8B/10B A standard encoding scheme used to protect data in communication networks

BRAM Block Ram. A dual-ported bank of memory available within the Xilinx FPGA architecture

CRC Cyclic Redundancy check: a hash function used to detect errors in data transmissions

CT Computed Tomography

DSP48 A specialized block in Xilinx Virtex4 FPGAs designed to enhance DSP performance. The DSP48 is essentially a multiply-accumulate block with additional logic to cascade mutliple blocks together to create larger operations. A single block can do a 18x18 bit signed multiply followed by a 48-bit addition.

FFT Fast Fourier Transform

FPGA A field-programmable gate array is a hardware device that can be reprogrammed at the hardware level on a per application basis. FPGAs are programmed at a particularly fine grain (gates and flip-flops).

HDL Most FPGA designs are done using a hardware description language (HDL); however, many efforts are now considering high level language (HLL) compilers.

MGT Multi-Gigabit Transceiver (e.g., Rocket I/O)

MRI magnetic resonance imaging

PE processing element — a single computational block for a particular function

Reconfigurable Computing Reconfigurable computing refers to the practice of using programmable hardware — hardware that can be "reconfigured" — to implement computing algorithms.

Rocket I/O Multi-Gigabit transceivers in the Xilinx FPGA

RTI Ray-Triangle Intersection

SERDES Serialization/Deserialization — Process of converting between serial and parallel data representations

TCP Transmission Control Protocol - a widely used standard for reliably transmitting data over a network

Chapter 1

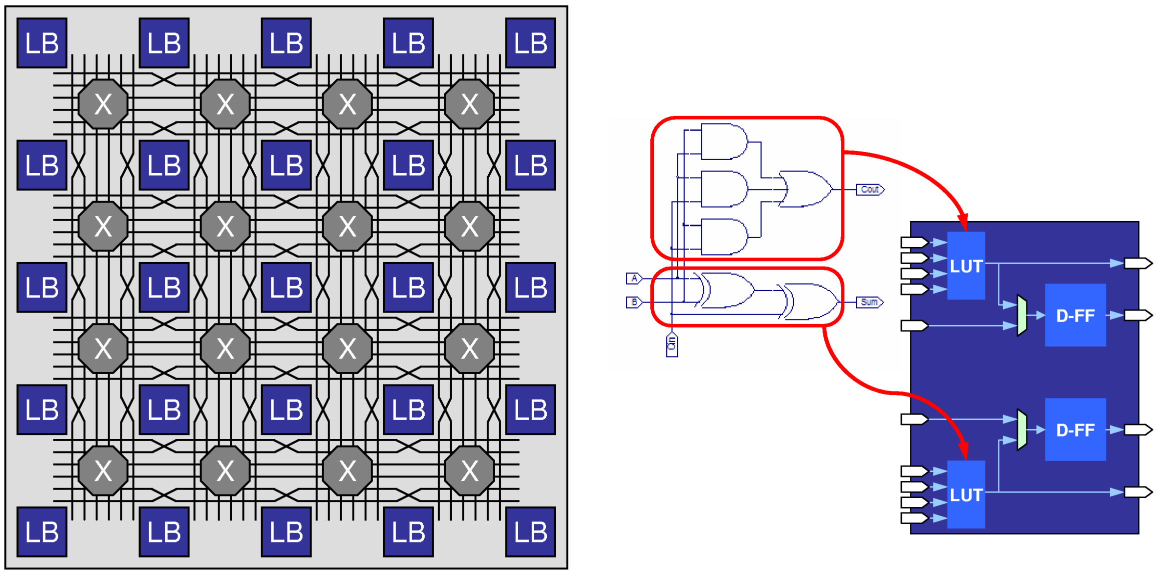
Introduction

The field of reconfigurable computing dates back almost 15 years. It focuses on the use of *programmable hardware* to implement computational algorithms. In contrast with traditional microprocessors, programmable hardware tends to run at lower clock rate, but exposes several orders of magnitude more parallelism. In contrast with ASICs (application specific integrated circuits), programmable hardware is significantly slower, but dramatically more flexible. Whereas ASICs are typically designed for exactly one purpose, programmable hardware can be reprogrammed (reconfigured) to meet the needs of whatever application is running. Thus, reconfigurable computing is often billed as delivering ASIC levels of performance with microprocessor levels of flexibility. In practice, reconfigurable computing typically uses field-programmable gate arrays (FPGAs) as the programmable hardware fabric.

FPGAs are reconfigurable hardware devices that can be programmed to emulate custom digital hardware that is specified by the end user. FPGA vendors such as Xilinx and Altera offer a variety of FPGA products that vary in terms of gate capacity (thousands to millions of user logic gates), power consumption (milliwatt to 20 watts), and cost ($20 to $5,000). While FPGAs are primarily utilized for prototyping designs before they are fabricated as ASICs, recent cost and size improvements have motivated industry to utilize FPGAs as ASIC replacements. These improvements have driven a renewed interest in utilizing FPGAs for reconfigurable computing research, as the larger and faster the FPGA, the more work that can be performed in a single chip.

One of the reasons why FPGA vendors are capable of rapidly improving their products is that FPGAs are tiled architectures. As illustrated at a high level in Figure 1.1, FPGAs utilize a two-dimensional array of programmable logic blocks (LBs) to implement the functionality of a user's design. An individual LB is comprised of one or more n-input lookup tables (LUTs), simple routing logic, and one or more memory elements. At design compile time, hardware synthesis tools map the logic functions in a design to truth tables that can be programmed into LUTs. The tools then place the LUTs in specific LBs and generate static routes through the FPGA to interconnect resources in the FPGA as required by the design.

In order to expand into new markets, FPGA vendors have recently begun to include special-purpose hardware units in their FPGA architectures. The "platform FPGAs" provide devel-



(a) **(b)**

Figure 1.1. (a) A high-level representation of an FPGA. (b) The mapping of a logic function to the LUTs of a LB.

opers with internal memory banks, integer arithmetic units, high-speed network transceivers, and even dedicated embedded microprocessors. For example, the largest FPGA in the Xilinx Virtex-II/Pro family features a megabyte of internal memory, twenty 3 Gb/s network transceivers, and two 400 MHz PowerPC 405 processors. These resources make current generation FPGAs more suitable for different kinds of applications and encourage researchers to utilize FPGAs in ways that were previously infeasible.

FPGAs have been demonstrated as great performers on DSP algorithms. They show order of magnitude (or more) benefits in every category from performance to performance/cost to performance/power. Indeed, FPGAs are beginning to dominate markets that were once controlled by ASICs and markets that were controlled by DSP processors. For example, many satellite programs use FPGAs. One big contrast between the DSP domain and the high performance computing (HPC) domain is the precision of the arithmetic. Where DSP and image processing applications typical use low precision numbers (narrow integers, fixed point, narrow floating-point), scientific applications using traditional HPC resources typically require double precision floating-point. In the past, research has indicated that FPGAs were inappropriate for such operations, but the work that spawned these two projects suggests that the latest FPGA devices can perform respectable amounts of floating-point arithmetic. Furthermore, a plethora of companies now offer FPGA based accelerators for HPC systems. Thus, two LDRD projects were created to explore whether FPGAs could be applied to traditional HPC and, if so, how.

Our research has demonstrated that FPGAs work quite well for double precision floating-point. In fact, some trends suggest that FPGAs could actually provide a significantly higher amount of peak floating-point performance than microprocessors in the near future. More remarkably, the investigation of algorithms on FPGAs has suggested that FPGAs could sustain a much higher percentage of that peak performance than microprocessors. In many cases, this is driven by market forces. Microprocessor vendors must deliver parts to a broad commercial market that does not have the level of bandwidth demands that many scientific applications have. Thus, for reasons of cost-effectiveness, microprocessors provide a relatively small amount of memory bandwidth. FPGAs, in contrast, target a market that needs abundant configurable I/O pins on the device. These I/O pins can be targeted to any purpose; thus, for the HPC market, we could choose to build a system where the FPGA used all of those pins for memory and had abundant memory bandwidth. In addition, although not shown here, other research has indicated that Sandia's scientific applications have more integer computations than floating-point computations. Microprocessors are balanced for equal amounts of integer and floating-point arithmetic, but FPGAs can dedicate resources in whatever balance is required. More importantly, integer computations are the thing that FPGAs do best; thus, this could give FPGAs a significant advantage.

As a means of complementing our investigation into floating-point use in FPGAs, we also investigated how much acceleration could be obtained in a practical sense from a real-world system. For this work we obtained a Cray XD1 system, which features an architecture that provides a tight coupling between host CPUs and FPGA accelerators. After examining the low-level characteristics of the XD1 and developing missing infrastructure, we adapted multiple computational kernels to the XD1's FPGAs and compared the performance difference between software and hardware implementations. These experiments revealed several observations. First, we were able to achieve speedups in all four of our examples even though our XD1 is only equipped with mid-sized FPGAs. While increasing the FPGA's capacity makes it easier to exploit parallelism, it is promising to know that clever architectural planning can still yield performance wins. Second, in multiple cases we found that the FPGA could easily out-pace the data rate of its connection to host memory. Thus, as good as an FPGA design may be, performance can ultimately be limited by the performance characteristics of the host in which it resides. Finally, adapting an algorithm to hardware (and getting it all to produce the right answers) is a time consuming process. Our application kernels and the corresponding host software took between three weeks and three months to develop and debug.

Having found a number of algorithms where FPGAs perform well, we also explored questions of system integration. This work had two facets: a practical implementation of network integration components and a simulation based study of an FPGA coupled directly to a microprocessor. In the network integration work we focused on the task of incorporating FPGA resources into an existing HPC system by connecting the FPGAs to the system's communication network. In order to achieve this goal, we investigated the communication capabilities of the Xilinx Virtex-II/Pro's Rocket I/O modules. We ultimately implemented a fully-functional network interface (NI) that is comprised of a TCP/IP Offload Engine (TOE) and a Gigabit Ethernet (GigE) unit. We found that the network-based approach to system integration was particularly useful in visualization applications (e.g., when an FPGA is used to generate data that is rendered by a cluster-driven, tiled display wall). The strength of this approach is that by relying on the network to bridge the gap between systems, FPGAs could be utilized in a flexible manner. However, it would be ill-advised for us to suggest that this approach is the right one for general HPC applications. The primary downside is that networks place a great deal of communication distance between CPUs and FPGAs. While some applications can tolerate this latency, it is a significant hindrance to application developers. In studying the interaction with a local microprocessor, we found that a broad class of operations had very little dependency on the latency of the connection, but a strong dependence on the bandwidth. In fact, the bandwidth requirements of future parts could reach so high as to make the coupling impractical. Perhaps more surprisingly, we found that the proposal by many to simply intercept existing library calls for operations such as matrix multiply or Fast Fourier Transform (FFT) would simply not work for the types of application usage scenarios at Sandia. Instead, a much better (non-blocking) interface would be required.

Despite many positive results, many challenges remain. While our studies have highlighted some important architectural characteristics, it will still be necessary to define a good architecture for integrating the FPGAs into a system. A good architecture will then need better ways to program the FPGA — application developers will not want to devote months learning to program hardware and weeks coding even the most simple of operation. In the current state of the art, the resulting programs would be locked to a single platform and it is unlikely that application developers would be willing to tolerate that lack of portability. Furthermore, current FPGA systems face distinct reliability challenges that must be addressed before they can be integrated into traditional HPC systems. Finally, the current cost of FPGA systems currently overshadows their impressive performance. The HPC market will need at least an order of magnitude improvement in a performance/cost metric over traditional systems to make it worth their effort.

Chapter 2

Related Work

This work is motivated by an extensive body of previous work in floating-point for FPGAs. A long series of work[33, 2, 5, 10, 22] has investigated the use of custom floating-point formats in FPGAs. There has also been some work in the translation of floating-point to fixed point[21] and the automatic optimization of the bit widths of floating-point formats[11]. In most cases, these formats are shown to be adequate for some applications, to require significantly less area to implement than IEEE formats[17], and to run significantly faster than IEEE formats. Most of these efforts demonstrate that such customized formats enable significant speedups for certain chosen applications. Unfortunately, many scientific applications depend on both the dynamic range and high precision of IEEE double-precision floating-point to maintain numerical stability. Thus, this work focuses on the IEEE standard. Indeed, some application developers within the DOE labs are beginning to discuss the need for greater precision than the standard IEEE formats, and such formats may be the topic of future work.

The earliest work on IEEE floating-point[7] focused on single precision and found that, although feasible, it was extremely slow. Later work[25] found that the performance was improving, but still relatively poor. Eventually, it was demonstrated[23] that while FPGAs were uncompetitive with CPUs in terms of peak FLOPs, they could provide competitive sustained floating-point performance. Since then, a variety of work[26, 2, 22, 40] has demonstrated the growing feasibility of IEEE compliant, single precision floating-point arithmetic and other floating-point formats of approximately that complexity. Indeed, some work[34] suggests that a collection of FPGAs can provide dramatically higher performance than a commodity processor.

Floating-Point Algorithms

Our earliest work[37, 38] demonstrated that FPGAs can indeed provide competitive floating-point performance with a potential for an order of magnitude win in the relatively near future This sparked a wave of research into the intersection of traditional high performance, scientific computing and FPGA based reconfigurable computing that paralleled our own work. Studies have indicated that FPGA can deliver high performance with the levels of precision needed by scientific computing[15, 13, 14, 12, 45, 46, 44, 6, 4, 47]. Numerous efforts have also begun to study double precision floating-point from the relatively simple matrix multiply[44] operations to the more complex LU decomposition[12] and sparse matrix-vector operations[47, 4].

The weakness in the majority of these studies, however, is that they do not consider either the API to deliver the performance to the application or the system architecture issues such as bandwidth and latency to the accelerator. Of these recent works, only a handful have discussed how the performance could be incorporated into an application. The three most notable examples are a molecular dynamics application[20, 32], a full CG solver[28], and a traffic simulation engine[36]. The major contribution of Sandia has been to consider the system level implications of such systems and the requirements for future systems.

Chapter 3

High Performance Floating-Point Modules

General Implications for Floating-Point Implementations

Floating-point arithmetic is fundamentally different from typical integer or fixed-point arithmetic. Where integer and fixed-point values are typically stored in two's complement, floating-point numbers are typically stored in signed-magnitude format. Floating-point numbers also add an exponent field to control the position of the decimal point in the value. The most widely used floating-point format is the IEEE 754 standard. As an example, the IEEE double precision floating-point format is shown in Figure 3.1. The mantissa (fraction part) is 52 bits, the exponent is 11 bits, and the sign is a single bit.

As the figure suggests, the exponent in the IEEE format is maintained in *biased* notation. For double precision, the bias is 1023 (approximately half the range), meaning an exponent of -1022 is stored as a 1. The second complication in the format is the use of an implied 1. An implied 1 means that the stored number is maintained in a normalized format such that there is a 1 immediately to the left of the decimal and the decimal is immediately to the left of the stored value. This allows the format to have an extra bit of precision without having to store it. Thus, the value can be extracted as shown in Equation 3.1.

$$(-1)^S \times 2^{exp-bias} \times 1.mantissa \tag{3.1}$$

There are also several special values define in the IEEE standard. These values are formed by using the reserved exponent values of all ones and all zeroes. The special values are zero, $\pm\infty$, and Not a Number (NaN), which is used as the result of meaningless operations (e.g. $\infty \times 0$). The reserved special values are summarized in Table 3.1. There is also a class

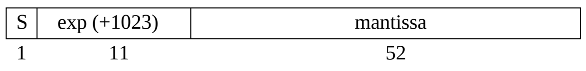


Figure 3.1. IEEE double precision floating-point format

Table 3.1. Special values in the IEEE 754 format

Special Value	Sign	Exponent	Mantissa
Zero	0/1	0	0
∞	0	MAX	0
$-\infty$	1	MAX	0
NaN	0/1	MAX	non-zero
Denormal	0/1	0	non-zero

of values known as denormals, which are represented by a zero exponent and non-zero mantissa.

Denormals are a special form of IEEE floating-point numbers that provide a small amount of extra precision as the result of an operation approaches underflow. Unlike most IEEE floating-point numbers, they do not include the implied one. Instead, they have an exponent of zero, keep the decimal immediately to the left of the stored value, and allow the first one to fall anywhere in the stored value. Floating-point hardware within a microprocessor typically implements denormals with an exception that then computes the value via software. However, in an FPGA-based implementation, to support full IEEE floating point we must generally add denormal support into the hardware itself. Thus, for denormal numbers, the value is extracted as in Equation 3.2.

$$(-1)^S \times 2^{1-bias} \times 0.mantissa \tag{3.2}$$

Floating-Point Unit Implementation

The library developed as part of this project includes modules for addition, multiplication, division and square root. Of these units, the most important are the adder and multiplier, as they are the most commonly used. In light of this, these units have had correspondingly more effort put into optimizing them and are discussed in more detail here.

General Considerations

The optimized floating-point units were adapted from units created for another research project. The original units were written in VHDL and formed the basis of the optimized units discussed here. There were two reasons to optimize these units: First, modern FPGAs are only now reaching a competitive state with CPUs in terms of floating-point performance, so the base floating-point units need to provide as much performance as possible. Second, the FPGA tools had a difficult time maintaining high clock frequencies when many units were used in a large design. In order to improve both of these, it was decided that

Table 3.2. Original Floating-point Units versus Optimized Units (size in slices, latency in cycles and speed in MHz)

Virtex2Pro (-6 speed grade)						
Precision	Adder			Multiplier		
	Size	Lat	Speed	Size	Lat	Speed
Single	495	13	195	592	16	176
Single - Optimized	328	10	268	345	11	267
Double	1090	14	143	1608	20	142
Double - Optimized	571	10	204	905	12	207

the units needed to be hand-mapped[1] and relationally placed to provide the best possible performance. The optimized units are written in JHDL[16], which is a structural design language that allows easy mapping and placing of logic.

The process of hand-mapping the units lead to further architectural enhancements. The enhancements allowed for further reductions in size and latency and an increase in operating frequency. In addition, the relational placement allows the units to maintain higher clock frequencies in large designs. Table 3.2 shows the size, speed and latency of the original and optimized adder and multiplier for both double and single precision. The values are taken for the Virtex2Pro (-6 speed grade), which was one of the fastest parts available at the time the original library was written.

The table shows that the optimizations provided a 1.8-1.9x improvement in area and a 1.4x increase in clock frequency for the double precision modules. This provided a total performance improvement of 2.7x for the adder and 2.6x for the multiplier.

Adder Implementation

The double precision adder consists of 9 operational stages. The general makeup of these stages can be seen in Figure 3.2. The fully pipelined version of the adder registers the data between each stage (indicated by dashed lines in the figure), and additionally registers the input before any computation is done, resulting in a latency of 10 clock cycles. The stages were designed to utilize all inputs to the 4-LUTs whenever possible. This results in both smaller, faster, and lower latency units.

The first two stages inspect the inputs and prepare them for the alignment step. This includes determining if either of the inputs is one of the IEEE special values, computing the alignment shift and swapping the smaller number into the aligment path. The larger exponent is also chosen as the interim exponent.

[1]Logic mapping consists of determining how the logic of the circuit maps to the FPGA structure, which consists mainly of 4-LUTs, but also has other auxillary logic that must be taken into account.

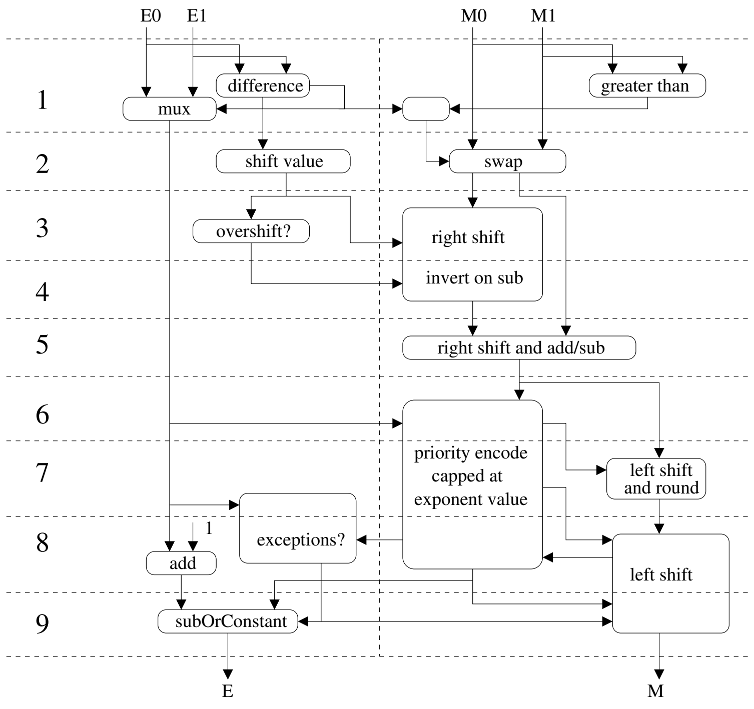


Figure 3.2. General computation layout for the floating-point adder

The third through fifth stages align the mantissas and perform the actual operation. Alignment is done through a series of shifting, conceptually implemented by stages which shift by different powers of 2, allowing any shift value from 0 to 53 to be generated. The majority of the shifting happens in stages 3 and 4, but the final shift by 1 is merged with the addition logic. Stages 6 through 9 are used to round the mantissa and renormalize it, if necessary. Again, in order to reduce circuit size, one stage of shifting is merged with the aritmetic logic which does the rounding. This happens in stage 7, while the rest of the shifting occurs in the final two stages. In addition to work on the mantissa, stages 7 through 9 also clean up the exponent and handle a variety of exception conditions.

Multiplier Implementation

The double precision multiplier is divided into 10 operational stages[2]. In addition, there is a fixed point multiply core which multiplies the two mantissas together. The total latency of the unit depends on the latency of the multiplier core. For Virtex2Pro, the multiplier core uses block multipliers and requires 4 cyles, resulting in a total latency of 12 cyles for the floating-point multiply. In Virtex4, the core uses DSP48 blocks, which include arithmetic logic in addition to the multiplier. These blocks reduce the required logic, but increase the latency to 7, bringing the total latency to 15.

Figure 3.3 shows the general breakdown of the stages for the multiplier. The mantissas are prepared for the multiplier core in the first three stages. This includes unpacking the numbers, checking for IEEE special values and normalizing a denormal input. The normalization logic includes an optimization which dramatically reduces the size of the unit: A denormal number is not fully normalized. Instead, the last three stages of shifting (4, 2, and 1) are removed to save logic. The result is that the backend shift path must be wider. However, the net result is a reduction of logic. The multiplier core begins operation in stage F4, and the backend stages (B1 through B4) round the result and prepare the mantissa for repacking into the IEEE format. The exponent path begins in stage F1 and runs through stage F6. Stages F4 through F6 operate in parallel with the multiplier core. The front-end stages add the two exponents together, subtract the bias and subtract the amount that the mantissa was shifted. In addition, the exponent is adjusted based on any IEEE special values. This result is then passed to the back end which will adjust it for the case of overflow, or in the case where the result of the core is greater than or equal to 2 and the mantissa must be shifted back into the acceptable range.

[2]There are two versions of the multiplier: one which supports denormal numbers and one which doesn't. The unit described here provides full support for denormal numbers.

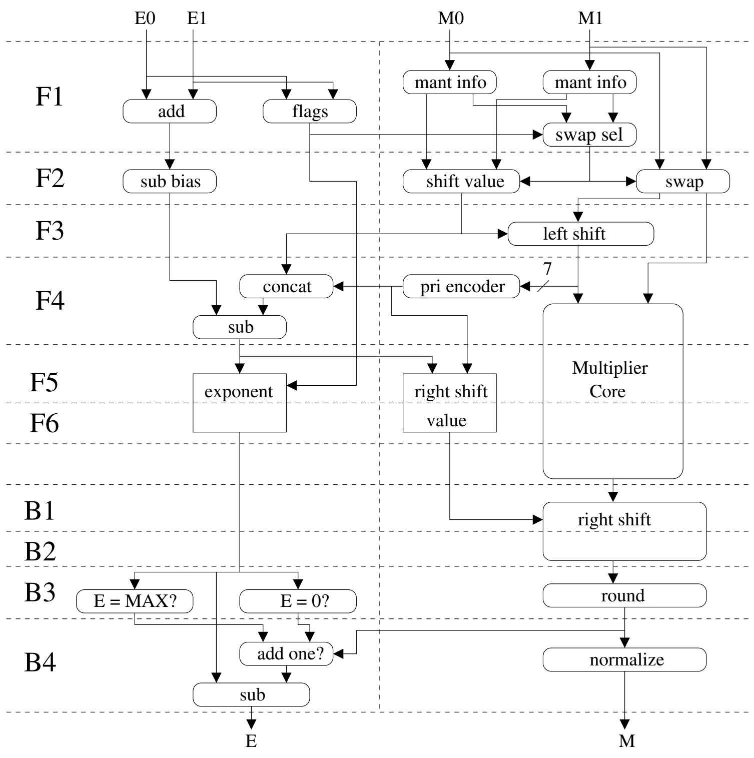


Figure 3.3. General computation layout for the floating-point multiplier

Characteristics

To access the quality of the Sandia floating-point modules, we have compared the performance characteristics of the floating-point units created in this project to those from Xilinx, Inc., which is currently the next best library available, in terms of performance.

The data shown in Table 3.3 gives the latency, size and speed characteristics of floating-point adders and multipliers found in the Sandia and Xilinx floating-point libraries. To make the comparison as accurate as possible, the units were configured to be as much alike as possible. This means that the Sandia units have removed denormal support and discounted the initial registers in both the size and latency fields. Also, the Xilinx double precision multiplier was created using the “medium usage” option so that it used the same number (9) of DSP blocks as the Sandia multiplier.

The Sandia floating-point units were mapped to Xilinx Virtex4 parts with a speed grade of -11. Sizes and speeds for each unit were obtained by placing a single unit in each FPGA and using the Xilinx place and route tools (ISE 8.1.03i) to determine size and maximum operating frequency. The numbers for the Xilinx library where found in the Xilinx Floating-Point Operator v2.0 datasheet. However, attempts to validate these numbers using the same methodology used for the Sandia units suggest that the Xilinx units are 20% larger than reported in the data sheets. This discrepancy arises because the datasheet numbers are determined setting the pack factor to one (-c 1). which constrains the tools to map and place the design into 1% of the FPGA area, if possible. This option is unlikely to be useful for a real application because of the clock rate penalty incurred. The table presents two numbers for the size of the Xilinx units: first, numbers directly from the data sheet (denoted Xilinx DS) and second, sizes from actual runs (denoted Xilinx) using the default pack factor. For double precision units, the Sandia adder provides a 1.7X size benefit while maintaining the same clock frequency and a 25% decrease in latency. The Sandia multiplier is about one-third the size of the Xilinx multiplier, operates at the same frequency and provides a 40% reduction in latency.

Table 3.3. Floating-point Unit Characteristics (size in slices, latency in cycles and speed in MHz)

Single Precision on Virtex4 (-11 speed grade)						
Library	Adder			Multiplier		
	Size	Lat	Speed	Size	Lat	Speed
Sandia	296	9	360	145	9	319
Xilinx	428	11	377	159	9	396
Xilinx DS	329	11	377	119	9	396

Double Precision on Virtex4 (-11 speed grade)						
Library	Adder			Multiplier		
	Size	Lat	Speed	Size	Lat	Speed
Sandia	507	9	275	384	10	274
Xilinx	860	12	271	1181	17	267
Xilinx DS	692	12	271	864	17	267

Chapter 4

Implications of Trends in FPGA Performance

As a precursor to the project in New Mexico, we studied trends in the performance of FPGAs and FPGA systems. We present the results of that study here for background, along with a retrospective on changes in those trends. To explore the potential of FPGAs to deliver their high peak performance to scientific applications, we also explored several important computational kernels. In this section, we focus on cycle accurate simulations of the performance of several kernels. We include a discussion of details such as memory bandwidth requirements and the parallelism required within the kernel. Leveraging data from our analysis of the trends in FPGA performance, the performance of the kernels was extrapolated to future systems and requirements, such as memory bandwidth, were extrapolated based on future performance levels.

Trends in FPGA Performance

Trends in the floating-point performance of FPGAs were analyzed immediately before the start of these FPGA projects[37]. The primary results from that work are provided here to illustrate the potential of FPGAs in the domain of scientific computing. Included after the 2003 analysis is a retrospective which discusses how these trends have held up over the last 3 years.

Peak floating-point performance for both microprocessors and FPGAs is analyzed over the period from 1997 to 2003 and extrapolated from there. For the microprocessor, performance is doubled every 18 months to represent the well known corollary of Moore's law. The trend line is forced through the 2003 data point for microprocessors. Since no such corollary has been established for FPGAs, the trend line is derived by starting with the 1997 data point, selecting a very conservative fit, and rounding the slope down.

Table 4.1 lists the parts chosen for the performance comparison. The commodity microprocessor with the highest peak performance was chosen for each of three years (regardless of the release date during the year). Since microprocessor performance trends are well known, only three data points were deemed necessary. Similarly, five FPGAs were chosen

Table 4.1. Parts used for performance comparison

Year	FPGA	CPU
1997	XC4085XLA-09	Pentium 266 MHz
1999	Virtex 1000-5	
2000	Virtex-E 3200-7	Athlon 1.2 GHz
2001	Virtex-II 6000-5	
2003	Virtex-II Pro 100-6	Pentium-4 3.2 GHz

for data points over the course of the same 6 years. An effort was made to choose the largest, fastest speed grade part with reasonable availability during that year[1]. For 2001, a stepping 0 Virtex-II 6000-5 device was chosen. Device stepping 1 was released early the next year and significantly improved the embedded multiplier performance. The part chosen for 1997 was chosen to be as representative as possible of the devices that would have been available; however, there was a constraint on which devices could be placed and routed for these experiments. The oldest tools that were available were Xilinx 4.2i, which do not support parts older than the XC4000XLA series. The XC4085XL-2 might have been a better choice, but the tools available would not target that device.

Admittedly, there are limitations to this type of analysis; however, the conservative assumptions that were made and the dramatic performance improvements projected should compensate for such limitations so that the "answer" is unchanged. For example, the order of magnitude performance advantage in 2009 may carry the same cost premium as current large devices. However, cheaper members of the same FPGA family will likely achieve a cost and performance advantage since FPGA performance is typically linear within a family, but cost is almost exponential. A second limitation is the lack of accounting for structures such as address generation and integer computation units. Such units are typically very simple in an FPGA. Finally, integration into a system is not considered here.

Addition

Figure 4.1 (a) indicates that FPGAs have significantly higher floating-point addition performance than commodity CPUs. This is a surprising revelation since FPGAs are generally considered to provide poor floating-point performance at best. More surprising still is the fact that FPGAs have been ahead for almost four years. The trend line for floating-point addition on FPGAs projects a growth rate of $4\times$ every two years. This trend line is diverging from the performance trend line for CPUs, which is only $2\times$ every 18 months. Notably, double precision addition performance on CPUs has been growing slower than the trend line from 1997 to 2003.

[1]These selections were made based on input from Chuck Cremer at Quatra Associates, Jason Moore at Xilinx, and personal memory.

FPGAs achieve this significant advantage over CPUs because their resources are configurable. If an application requires a very addition rich mixture of instructions, the FPGA can provide that. In contrast, commodity CPUs have a fixed allocation of resources that can be successful for a well mixed instruction stream, but has significant limitations in code that is dominated by one instruction.

Multiplication

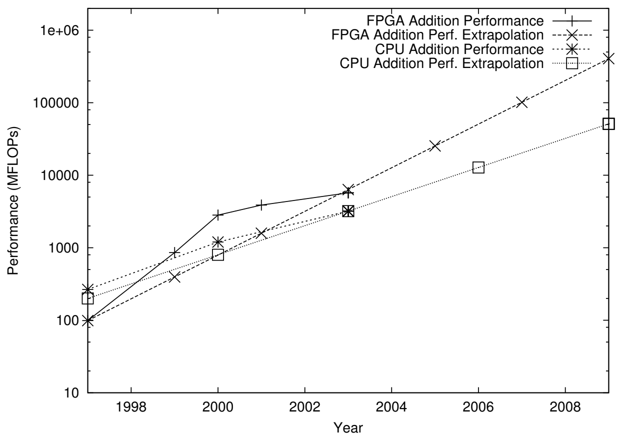
FPGAs have not dominated CPUs in floating-point multiplication performance for nearly as long they have in floating-point addition performance. Figure 4.1(b) indicates that FPGAs have only exceeded CPUs in multiplication performance since 2001 for double precision arithmetic. However, the trend lines are diverging more rapidly with multiplication performance growing at an average rate of 5$\times$ per year from 1997 to 2003. This is primarily because of the addition of 18×18 fixed multipliers in the Virtex2 series of parts. The use of these components to implement the multiply of the mantissas (9 for double precision, 4 for single precision) dramatically reduced the area required. This trend is likely to continue since architectural improvements (notably faster, wider, fixed multipliers) are likely to continue.

The CPU performance in Figure 4.1 (b) has grown slightly faster than the Moore's law trend line between 1997 and 2003. For double precision, this is primarily because of a change between 1997 and 2000 to allow multiplies to issue every cycle.

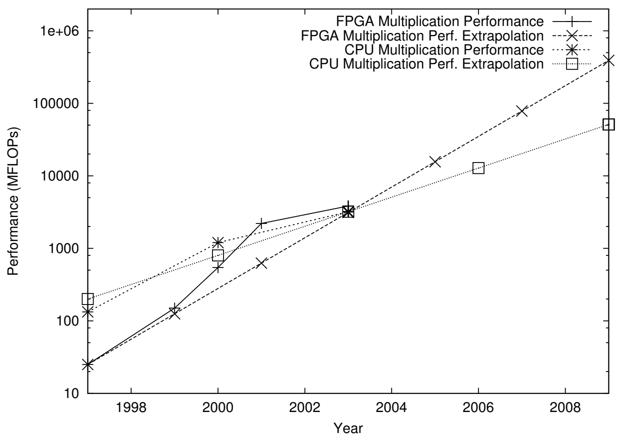
Division

As seen in Figures 4.1 (c), FPGAs have long exceeded CPUs in floating-point division performance — with one minor caveat. An XC4085XLA is not big enough (by a significant margin) to implement a double precision divider. Thus, the "performance" of a double precision divider on that part is the fraction of the divider it can implement times the estimated clock rate. This explains why the 4$\times$ trend line overestimates the performance of the 2003 part: components that can implement the operation are constrained to integer multiples of divide units. The XC4085XLA had a second artificial performance inflation because the mapper packed the CLBs much tighter to try (in vain) to make it fit. Thus, the area estimate is significantly smaller than it would otherwise be.

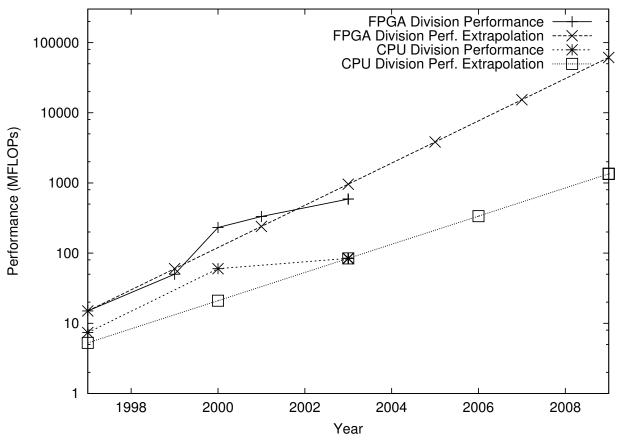
Commodity microprocessors are not well optimized for divide operations. This is a significant issue for some scientific applications[39]. Slow divides have first and second order effects: the division instructions are slow (and unpipelined) and these slow instructions clog the issue queue for modern microprocessors. This makes divide rich applications a good target for FPGA implementations.



(a)



(b)



(c)

Figure 4.1. Floating-point performance for: **(a)** double precision addition, **(b)** double precision multiplication, and **(c)** double precision division.

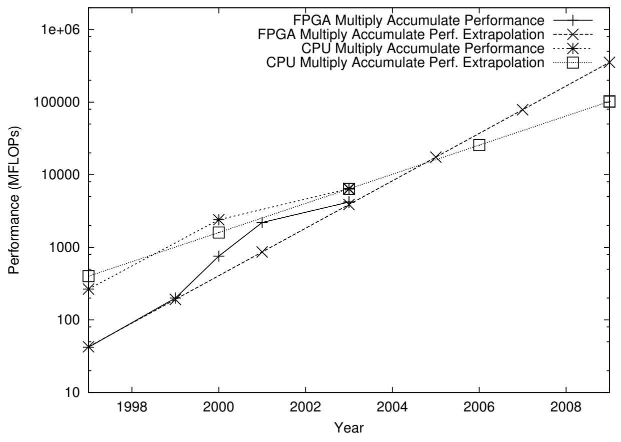


Figure 4.2. Floating-point double precision multiply accumulate performance

Multiply Accumulate

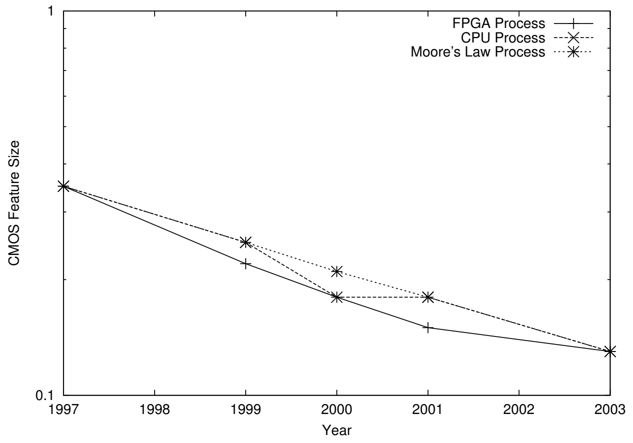
Multiply accumulate is somewhat different from the other operations considered in that it is a composite operation. More importantly, it is a composite operation that is fundamental to a number of matrix operations (including LINPACK). Figure 4.2 indicates that FPGAs are still somewhat slower than CPUs at performing this operation. FPGAs are, however, improving at a rate of 4.5$\times$ every two years. This improvement rate (effectively a composite of the performance improvements in addition and multiplication) yields a significant win for FPGAs by the end of the decade.

It would be easy to suggest that the comparison between FPGA and CPU in this case is not "fair" because the FPGA requires many concurrent multiply accumulates (in one multiply accumulate functional unit) to overcome the latency of the adder and achieve the projected performance; however, it should be noted that the Pentium-4 must alternate issuing two adds and two multiplies[2] with 4 cycle and 6 cycle latency, respectively, to achieve its peak performance [1]. With the small register set in the IA-32 ISA, this is not necessarily easier to exploit than the concurrency and parallelism available on the FPGA.

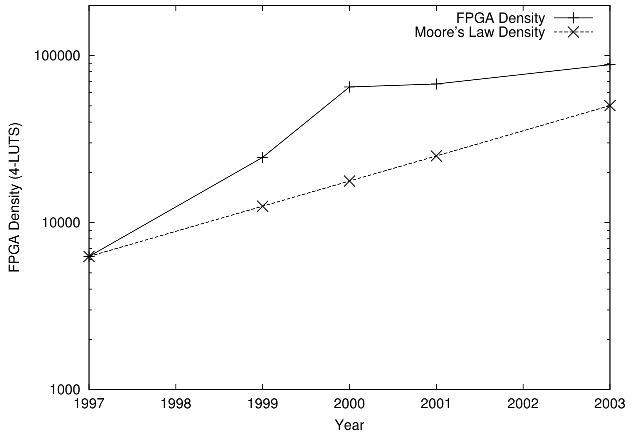
Analysis

A common theme among the performance graphs is the flattening of the performance trend line from 2000 to 2003. This is supported by the data in Figure 4.3, which clearly indicates a flattening in the growth of area, increase in clock rate, and feature size reduction. This

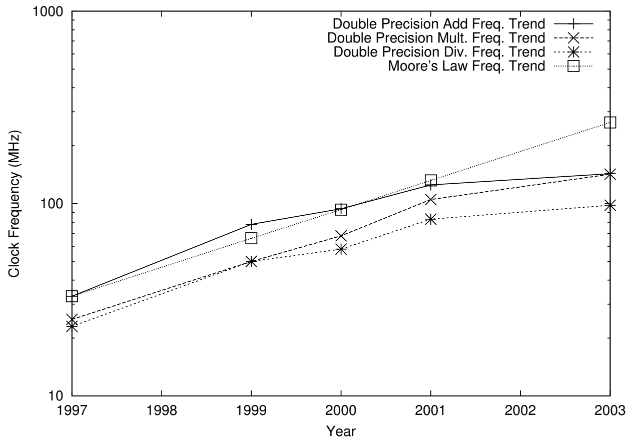
[2]The throughput of the SSE2 multiplier is one instruction per 2 cycles. Each instruction can do two multiplies.



(a)



(b)



(c)

Figure 4.3. Basic FPGA property trends including: **(a)** CMOS Feature size, **(b)** density in 4-LUTs, and **(c)** clock rate.

appears to bode ill for the projected performance trends; however, a closer look at Figure 4.3 indicates differently. In Figure 4.3(a), the trend in FPGA feature size broke sharply between 2001 and 2003, but it is still on the same overall trend line for the 6 year period as CPUs. Indeed, low cost FPGAs have already been introduced on the 90 nm process — well ahead of CPUs. High performance parts are expected to be introduced next year concurrently with CPUs based on 90 nm technology. Similarly, Figure 4.3(b) shows that the pace of FPGA density improvements has dropped sharply from 2000 to 2003, but overall density increases are still above the Moore's law projection. Even if a much larger device (the XC40125EX) was used as the 1997 baseline, the overall density improvement would remain slightly above the Moore's law projection.

Figure 4.3(c) seems to tell a slightly different story with regards to clock rate. The "Moore's law" trend line for clock rate provides a reference that clearly indicates that clock rate has not scaled as expected. However, this seeming discrepancy is relatively easy to explain. The device used as representative of 1997 technology for these experiments was the XC4085XLA-09. A more accurate part would have been the XC4085XL-2, but the Xilinx 4.2i tools that were available for these experiments would not process such a device. A XC4085XL-2 part is approximately 40% slower than the XC4085XLA-09 part used. Combining this with the significant performance increase that Virtex-II Pro parts should receive as the tool chain develops trends in clock rates that meet the expectations of Moore's law.

Improvement in addition and division performance are derived strictly from technology improvement; however, the 5$\times$ every two years performance growth of multipliers will be difficult to maintain indefinitely. Fortunately, only minor improvements are needed each generation to realize this gain. This should be readily achievable through 2009 (wider embedded multipliers, enhanced support for shifting, etc.).

Analysis Retrospective

FPGA vendors have not continued to deliver the level of performance improvements projected by the analysis here. While many would attribute this to a slowing in clock rate improvements delivered by Moore's Law, the reality is more closely related to choices made by the FPGA vendors. The primary *computational* market for FPGAs is driven by digital signal processing (DSP) style operations. DSP operations are typically narrow bit widths and, consequently, shallow pipelines. In contrast, floating-point uses both narrow and wide datapaths and has very deep pipelines. Similarly, FPGA vendors have chosen to create product mixes that target a set of domains that do not map well to floating-point applications. The primary concern is the lack of sufficient numbers of DSP48 blocks which limits the number of floating-point multipliers that can be built. If FPGA vendors chose to focus on the HPC market, they could continue to deliver improvements in clock rate for floating-point operations for a few more generations. This could be done both with basic architectural changes and with providing a better optimized set of resources. The latest Xilinx FPGA family, the Virtex5, shows some promise in the direction of architectural enhancements. The Virtex5 provides wider multiplier blocks (though not as wide as we would like to see), it also has attempted to increase clock frequency by moving from 4-input LUTs to 6-input LUTs, and by providing a coarser grained carry chain. However, it is unclear if Xilinx will provide a part with a good mix of these components (this only family currently released does not have a good mix of logic to multipliers).

The story with microprocessors is more complex. While microprocessors had been falling off of the trend of doubling performance every 18 months, recent developments, such as the move to 4 FLOPs per clock, have suggested that the microprocessor vendors are again looking to architectural changes for performance. These changes are likely driven by the perception of competition from the accelerator market, and have even led Intel to announce a prototype "teraFLOP on a chip". These types of advances could significantly erode many of the advantages an FPGA may have; however, the microprocessor market is *not* currently expanding memory bandwidth, so FPGAs could still deliver a significant advantage in *sustained* performance.

Kernels Explored in Simulation

We explored four kernels in cycle accurate hardware description language (HDL) level simulation. These included three basic linear algebra subroutines (BLAS): a matrix-matrix multiply (DGEMM), a matrix-vector multiply (DGEMV), and a dot product (DDOT). We also explored the fast Fourier transform (FFT). Here, we present results from DGEMM, DGEMV, and FFT as they represent three unique points in the spectrum.

Matrix multiply (DGEMM) is a completely computationally bound code in the sizes that most people study. Because it performs N^3 operations over N^2 data with numerous independent operations, it is highly amenable to sustaining high levels of performance on both FPGAs and microprocessors. In contrast, a matrix-vector multiply (DGEMV) requires as many memory operations as floating-point operations. This is much more representative of many codes at Sandia (an iterative solver, for example). Finally, the FFT operation has a much more complex control flow and more data dependencies than either DGEMM or DGEMV operations. Thus, the three kernels we present here highlight three unique facets of how FPGAs behave on floating-point operations.

Matrix Multiply

The standard matrix multiply (the DGEMM BLAS routine) is defined as:

$$\mathbf{C}_{ij} = \sum_{k=0}^{N-1} \mathbf{A}_{ik}\mathbf{B}_{kj} + \mathbf{C}ij \tag{4.1}$$

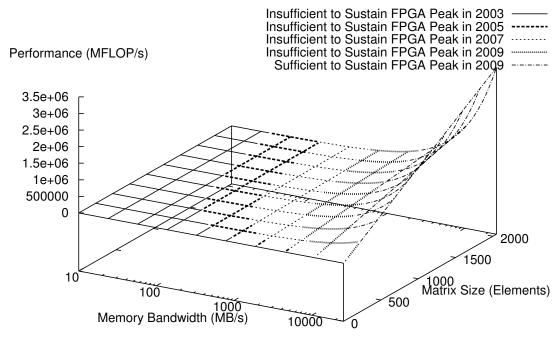


Figure 4.4. Maximum achievable performance versus memory bandwidth and matrix size

In the best case, this requires $4N^2$ memory accesses[3] and performs $2N^3$ floating-point operations. This yields $\frac{N}{2}$ floating-point operations for each element retrieved from memory. Achieving the best case, however, imposes the unrealistic requirement that two matrices be cached in the processor. Fortunately, proper use of caching in modern processors allows them to sustain a high percentage of peak with relatively low memory bandwidth. If each matrix only had to be retrieved from memory once, the maximum sustainable floating-point rate would be:

$$FLOPs = \frac{\frac{N}{2} \times BW}{8} \tag{4.2}$$

where BW is the memory bandwidth in bytes per second, N is the dimension of the matrix, and 8 bytes are required to store a floating-point number. This is graphed in Figure 4.4 on a log-log graph.

Typically, however, the processor cannot store all of the matrices involved. Instead, some form of blocking is used to divide the matrix into smaller pieces[42]. These smaller pieces are loaded into the processor, the computations are performed on them, and the partial results stored. For example, for a 64×64 matrix multiply, each matrix might be broken into 64 regions that are 8×8. A row of these blocks would then be multiplied by a column of these blocks to create an 8×8 block of the result. In the process, the partial result (an 8×8 block) would be updated 8 times (although typically in local storage or cache). The ultimate result is that the matrices are fetched several times more than would otherwise be necessary. For blocks of dimension S, this yields a factor of $\frac{N}{S}$ increase in accesses to the **A** and **B** matrices, leading to $2N^2 + \frac{2N^3}{S}$ memory accesses. For large matrices, this approaches

[3]This assumes square matrices and includes retrieving three matrices and storing one matrix.

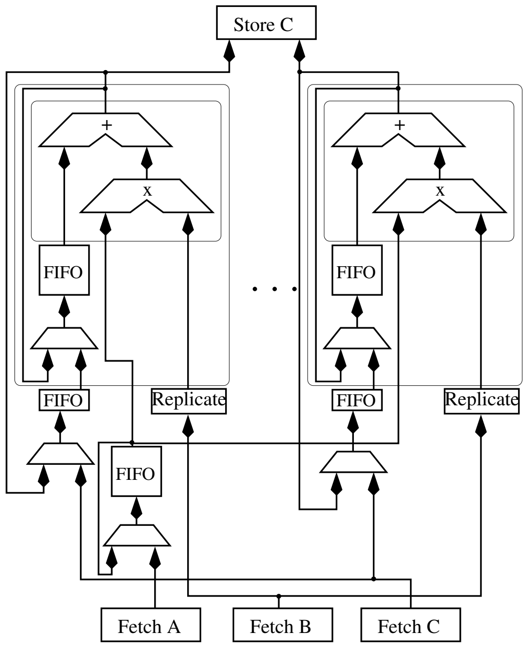


Figure 4.5. Matrix multiply implementation

a floating-point rate of:

$$FLOPs = \frac{S \times BW}{8} \tag{4.3}$$

FPGA Implementation

One way to view matrix multiplication is as a collection of *N* matrix-vector multiplications. As such, it exhibits significantly more parallelism. Unfortunately, it is impractical to store all of a matrix in most modern devices (with the possible exception of the Itanium chips with the largest caches) for reasonable matrix sizes. Thus, the implementation chosen (shown in Figure 4.5) resembles a collection of matrix-vector multiplications, but blocks the matrix to reduce total storage requirements.

To perform the matrix multiplication, a $S \times \frac{S}{m}$ section of a block of the matrix **B** in column major order is loaded into each of *m* MACC units. Simultaneously, the matching block of the matrix **C** is also loaded into the MACC unit in column major order. This requires a total of $6S^2$ elements of storage at 8 bytes per element. The corresponding block of **A** is loaded into a FIFO that is used to broadcast it to all MACC units $\frac{S}{m}$ times (also in column major order). Each MACC unit creates *S* replicas of each element of **B** to match the number of rows of **A** that will be multiplied with it. This provides the concurrency needed to hide all

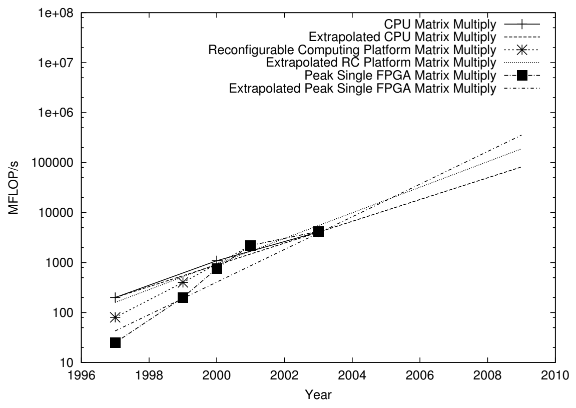


Figure 4.6. A comparison of double precision floating-point matrix multiplication performance on CPUs, FPGAs, and RC platforms

of the latency of the adder. As each element of **B** and **C** is used, it is discarded. When the MACC unit finishes, it produces an intermediate version of the result **C**. This intermediate is fed back to the input to be added to the multiplication of the next pair of blocks from **A** and **B**. When the final version of a block of **C** is produced, it is stored. Overall, this requires no more than $6S^2$ elements of storage at 8 bytes per element. This includes 2 copies of each matrix block — one to operate on and one to change it from row major to column major order.

Performance Comparisons

Matrix multiply is not typically memory bound. With relatively little caching, modern microprocessors can achieve a high percentage of peak performance with commodity memory attached. As seen in Figure 4.6, this enables microprocessors to maintain an edge over FPGAs and RC platforms up through 2003. For this graph, data points for the CPUs in 2000 and 2003 were measured and the data point for the CPU in 1997 was taken from [41]. The extrapolated trend line projects a $4.5\times$ growth in performance every 3 years.

Like microprocessors, the performance of matrix multiply on FPGAs is unconstrained by any architectural features. As such, the rate of growth ($4.5\times$ every two years) is the same as for the multiply-add performance projected in [37]. Recent reconfigurable computing platforms are also included in this comparison. These platforms often use multiple FPGAs and provide a "realistic" design point with a "realistic" memory bandwidth.[4] Most points are estimated, but an implementation on the Osiris board to validate the estimates has been simulated. Extrapolating the trend yields a performance growth rate of $3.25\times$ every two

[4]Realistic in that people buy it, but not realistic in a cost comparison with traditional processors.

years. This is lower than the FPGA growth rate because the number of FPGAs on RC platforms has been steadily decreasing. This growth rate may accelerate over the coming years (since less than one chip per board is unlikely).

Reconfigurable computing platforms also demonstrate the same characteristics (peak performance and performance growth) as matrix-vector multiply. Again, all but one of these points are currently estimated, but an implementation has validated one data point on an Osiris board.

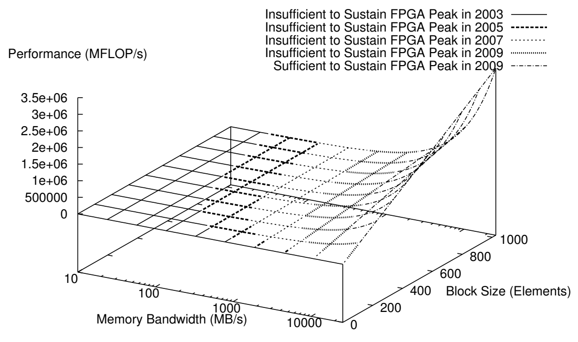
Memory Requirements

The key feature that distinguishes matrix multiplication from matrix-vector multiplication is that it requires much less memory bandwidth to maintain peak performance. This is an important factor since supercomputing applications of FPGAs are unlikely to provide the level of memory bandwidth that many RC platforms provide (due to the additional cost and negative reliability implications of several extra banks of memory). As such, it is important to examine the storage needs of FPGAs relative to their peak bandwidth and peak performance.

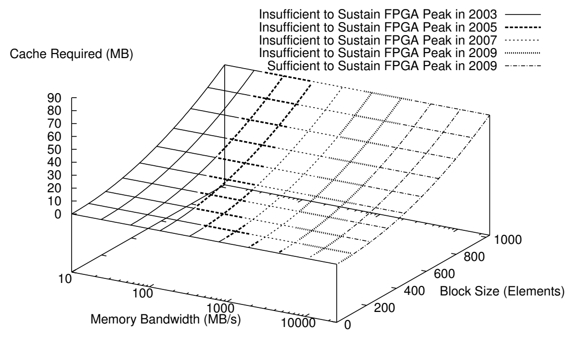
Figure 4.7(a) illustrates the high levels of performance achievable with relatively little memory bandwidth and sufficient internal storage. For example, the 4 GFLOP/s of peak floating-point capability available in 2003 can be sustained with only 512 MB/s of memory bandwidth and only 192 KB of internal storage in the FPGA. This easily falls within the limits of even the Virtex-II family. The important observation here is that internal memory can be traded for external memory bandwidth.

The flip side of this analysis is to consider the amount of internal memory needed for various memory bandwidth and block size combinations. Figure 4.7(b) compares the size of the internal cache needed to the amount of memory bandwidth needed and the block size. By 2009, FPGA platforms will need to provide several gigabytes per second of memory bandwidth or several megabytes of internal storage. For this algorithm, however, both the memory bandwidth and internal storage needed to maintain peak performance appear to be easily achievable.

An interesting revelation from this analysis is the significant gains in efficiency that FPGAs achieve. These gains appear to arise from the ability to explicitly manage internal storage and the high aggregate internal memory bandwidth. Commodity processors typically achieve only 80-90% of their peak performance on dense matrix multiplies while requiring hundreds of kilobytes of cache and gigabytes per second of memory bandwidth. By contrast, the FPGA only requires 192 KB of storage and 512 MB/s of external memory bandwidth. If this trend holds for other algorithms, it will be a significant advantage for FPGAs in the realm of high performance computing.



(a)



(b)

Figure 4.7. (a) Maximum achievable performance versus memory bandwidth and block size; (b) Memory needed in an FPGA to maintain peak performance versus memory bandwidth and block size

Matrix-Vector Multiply

The DGEMV BLAS routine is defined as:

$$\mathbf{w}_i = \sum_{j=0}^{N-1} \mathbf{A}_{ij}\mathbf{y}_j + \mathbf{z}_j \tag{4.4}$$

The lower bound on memory accesses is $N^2 + 3N$ (it must retrieve a matrix and two vectors and store one vector) and performs $2N^2$ floating-point operations. Achieving that requires that the vector be cached in the processor. In the limit, two floating-point operations are performed for each element retrieved from memory. DGEMV is also a memory limited operation on microprocessors. The maximum sustainable floating-point rate is:

$$FLOPs = \frac{2 \times BW}{8} \tag{4.5}$$

where BW is the memory bandwidth in bytes per second and 8 bytes are required to store a floating-point number. This is graphed in Figure 4.8 on a log-log graph.

FPGA Implementation

Matrix-vector multiplication has somewhat less inherent parallelism than matrix multiply. When the matrix dimension is greater than the depth of the addition pipeline, it is possible

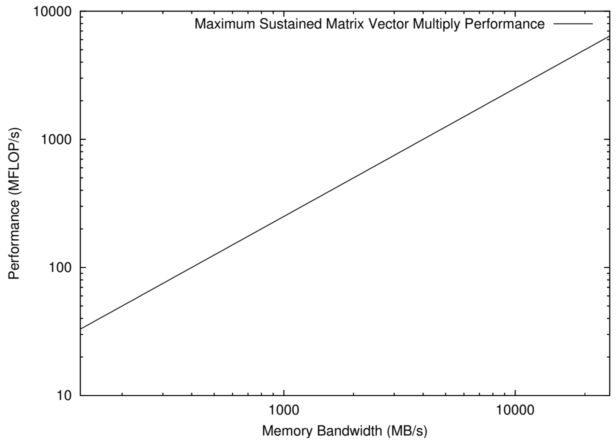


Figure 4.8. Maximum sustainable matrix vector multiply performance versus memory bandwidth

to use all of the floating-point capabilities of the FPGA without making special adaptations to the multiply accumulate unit. Instead, a standard MACC (shown in Figure 4.9) is used with additional storage and control outside of it as shown in Figure 4.10.

The design in Figure 4.10 assumes that the matrix, **A**, is stored in one logical memory with a bandwidth that will support some number of MACC units, m. Thus, the vectors **y** and **z** are loaded and broadcast to all of the MACC units while the matrix is fetched from memory and distributed to the MACC units. Before broadcasting to the MACC units, each element of **y** is replicated k times (where k is the pipeline depth of the adder). Effectively, there are k rows of the matrix being multiplied with the vector in each MACC unit. The vector **z** is used to initialize the summation unit. Thus, one MACC unit is fed the j^{th} element of k rows to match k copies of the j^{th} element of **y**. The matrix data can be fetched directly in this order from SRAM memories or can be fetched in larger pieces from SDRAM with re-ordering performed in the matrix fetch unit. Note that, once loaded, **y** is reused (through the feedback path) $\frac{N}{k \times m}$ times. This design does not apply the scalar multiples to **y** and **z** that are indicated in the standard, but such a change would be relatively simple.

The limitation of this approach is the need to store a vector for large values of N. The alternative is to divide the vector into parts of length L and perform the equivalent of $\frac{N}{L}$ smaller matrix-vector multiplies. The first would add the vector **z** to the result (as in Equation 4.4) and subsequent portions of the operation would add the intermediate result in place of **z**. This would increase the number of memory accesses to $N^2 + \frac{3N^2}{L}$. For FPGAs from 1999 and forward, an L value of 2000 is achievable. Thus, the N^2 term for the matrix access still dominates. This issue would only serve to decrease performance estimates for earlier FPGAs (thus skewing the trend line) and will not be considered further.

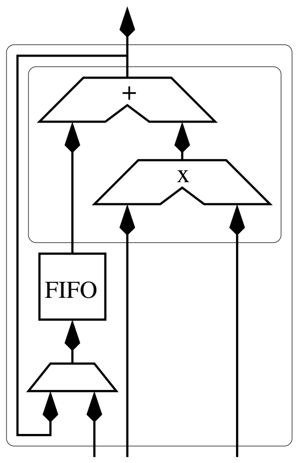


Figure 4.9. Multiply-accumulate unit

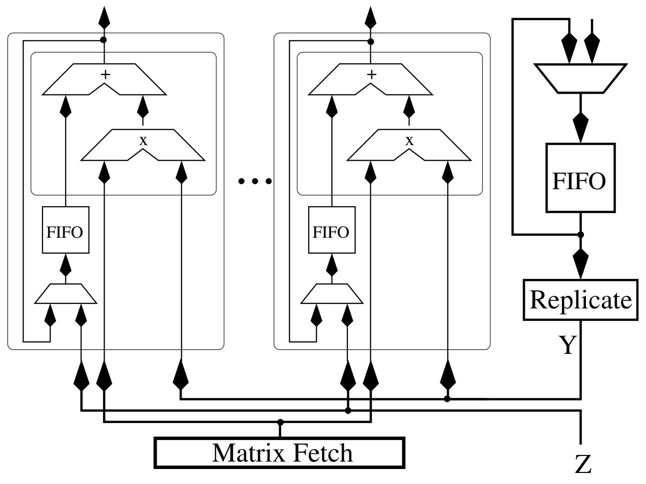


Figure 4.10. Matrix vector multiplication implementation

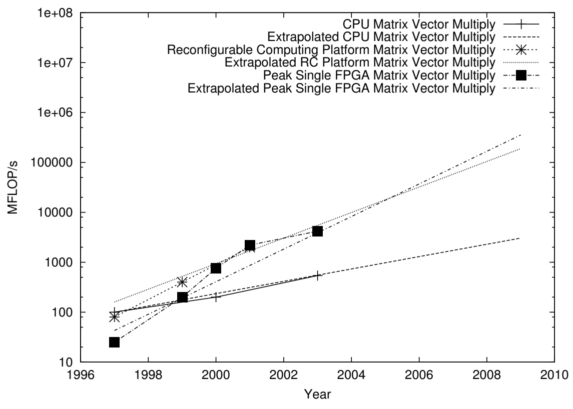


Figure 4.11. A comparison of double precision floating-point matrix-vector multiplication performance on CPUs, FPGAs, and RC platforms

Performance Comparisons

Unlike matrix multiply, matrix-vector multiplication is typically a memory bound operation; however, devices that provide enough memory bandwidth shift the limitation to the floating-point performance of the device. The performance of recent RC platforms is compared with recent commodity CPUs and the peak performance possible with a single FPGA in Figure 4.11. CPU performance was measured for the 2000 and 2003 data points with an N of 1000[5] and estimated for the 1997 data point based on the differences in memory bandwidth and architecture.[6]

Like `DGEMM`, peak `DGEMV` for the FPGAs is based on the maximum multiply accumulate performance of the FPGA. This is because the use of all of the FPGA pins for memory provides abundant bandwidth to achieve the peak performance. `DGEMV` also has sufficient independent parallelism to enable the FPGA to achieve its full peak performance on smaller matrix and vector sizes. Extrapolating this trend yields the same $4.5\times$ growth every two years as the dot product since the FPGA pin bandwidth never becomes the limiting factor.

Recent reconfigurable computing platforms are also included in this comparison. Most points are estimated, but an implementation on the Osiris board to validate the estimates has been simulated. Using the same assumptions used for the vector dot product yields a performance growth rate of $3.25\times$ every two years. This is lower than the FPGA growth rate because the number of FPGAs on RC platforms has been steadily decreasing. This growth rate may accelerate over the coming years (since less than one chip per board is

[5]Performance was constant across a wide range of values for N.

[6]This is a best case estimate assuming the older processor could sustain as much of the peak memory bandwidth as the newer processors.

unlikely). Nonetheless, since 1999, RC platforms have had both higher performance than commodity CPUs and a higher rate of growth. As a note, the performance for the 1997 RC platform data point is particularly low because the XC4000 series parts did not have sufficient internal storage to store a vector of any significant length. This would give the trend line an artificially low starting point and artificially steep slope[7], but the trend is established based on the performance the 1997 part would achieve if it had internal storage.

The final point for discussion is the amount of storage needed to achieve this performance. As noted, the XC4000 series had insufficient internal storage to achieve the full potential of the 1997 RC platform. The Virtex 1000 part, however, has enough storage for 1000 vector elements. The Virtex-2 6000 has enough storage for 10000 vector elements and the Virtex-2Pro series has even more storage. As long as the total storage in an FPGA is not decreased (an unlikely scenario), FPGAs already have abundant storage for matrix-vector multiplication.

Fast Fourier Transform (FFT)

The FFT is an optimized implementation of the Discrete Fourier Transform (DFT). The fundamental calculation of the N point DFT is described as:

$$Y[j] = \sum_{k=0}^{N_1-1} X[k] W_N^{jk}; \texttt{ where } W_N^{jk} = e^{\frac{-i2\pi jk}{N}} \tag{4.6}$$

When implementing the DFT as an FFT[35], there are many ways to structure the computation. The first decision to make — the radix of the operation — determines how many data items are combined in any given stage. Radix-2, where each stage of computation operates pairwise on the data set, was chosen for several reasons, including its simplicity. Radix-2 also yields the smallest butterfly unit, which allows for greater flexibility in studying the design space. Other radixes reduce the total number of operations, but increase the complexity and reduce the flexibility. Similarly, multiple-radix designs can increase flexibility.

Given a radix, there is still flexibility in formulating the computation. One formulation yields a basic computational kernel consisting of a single complex multiplication and two complex additions, as shown in Figure 4.12(a). This structure is commonly referred to as a butterfly because of the crossing dependencies. The radix-2 FFT is made up of $log_2(N)$ stages, where each stage computes $\frac{N}{2}$ butterflies.

There are two main ways to structure the stages, shown in Figure 4.13. A butterfly unit in a stage operates on a pair of results from the previous stage separated by an increasing (a)

[7]Although the embedded memories in newer FPGAs could be considered an architectural improvement (making this valid), similar improvements for matrix-vector multiplication are unlikely to occur.

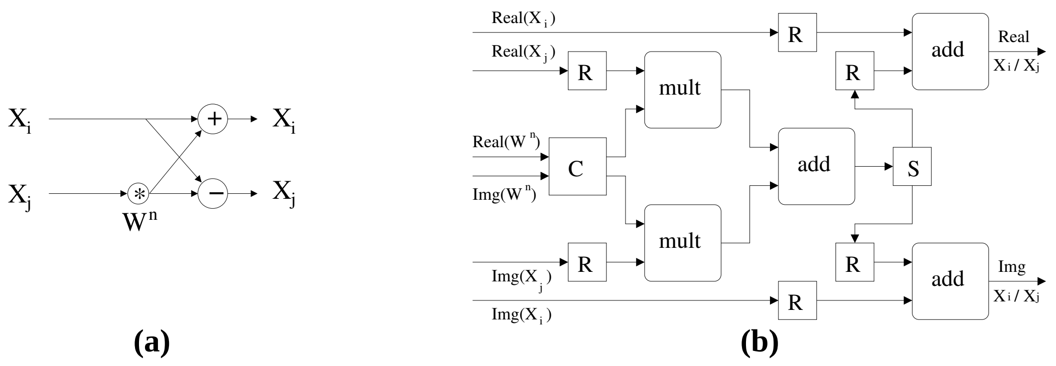


Figure 4.12. **(**a) Basic butterfly operation. **(b)** Basic butterfly datapath. The component S is a switch that directs inputs to alternate outputs. The components marked as R replicate the input once and C is a crossover to facilitate the complex multiply.

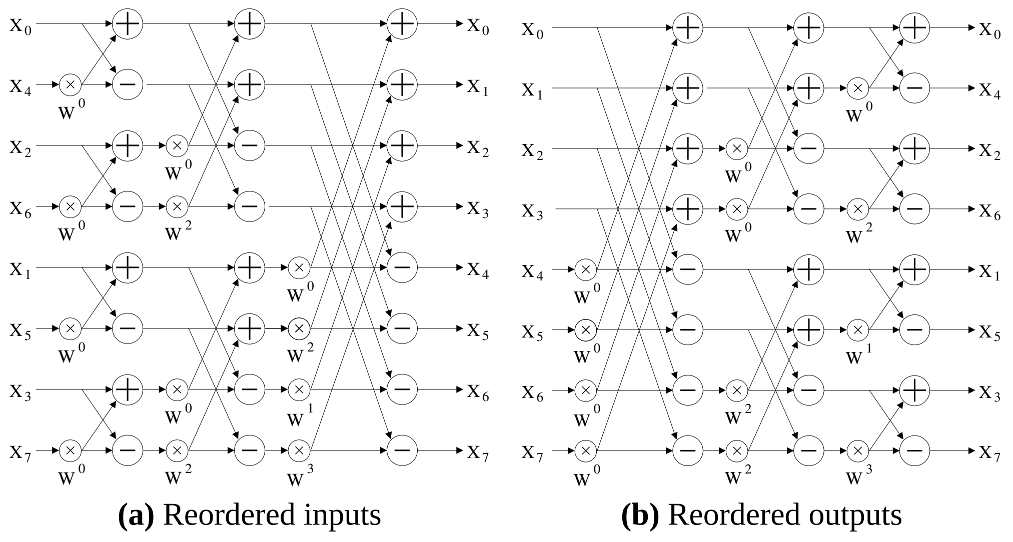


Figure 4.13. Two variations of 8-point, radix-2 FFTs

or decreasing (b) distance. Each of these structures requires a data reordering, either on the front- or back-end. We chose to reorder on the back-end as it allowed us to do the biggest butterflies first, providing more independent data sets with each progressive stage. Both approaches produce the same results, but in a different order.

The butterfly computation requires 4 real multiplications and 6 real additions. The hardware used in this study does these computations using 2 floating-point multipliers and 3 floating-point adders, as shown in Figure 4.12(b) On average, the hardware accepts one new complex number per cycle and generates one complex output per cycle. Each floating-point unit is used twice for each set of inputs, providing the total operations required for the butterfly. This design for the butterfly unit was chosen because it provides the greatest flexibility. It only requires a memory bandwidth of one complex data item per cycle, letting memory bandwidth scale more evenly. The bandwidth in and out of the unit also matches the bandwidth of on-chip dual-port block rams, allowing a single butterfly unit to be coupled with a single set of block rams on the chip.

In evaluating the performance of the FFT, the floating-point operation count that is typically used is $5Nlog_2(N)$; there are $log_2(N)$ stages that each contain $5N$ computations (four multiplies and six additions for each pair of data). These numbers only hold strictly for radix-2, though they are a good approximation for other radixes. As the radix increases, the number of stages goes down, but stage complexity increases.

The minimal data transfers to memory for the FFT is $2N$ elements with each element being 16 bytes for double precision complex numbers. This gives $32N$ bytes for a best case overall bytes per FLOP requirement of $\frac{32}{5log_2(N)}$. The actual bytes per FLOP requirement, however, will depend on how much of the data can be successfully cached on chip. This is discussed further in the discussion of each algorithm.

Parallel FFT Implementation

Parallelism in the FFT computation can be exploited in two ways: pipelined units, or parallelism in the stages (S), and parallel units, or parallelism (P) within a stage. The parallel design point explores the extreme where all of the parallelism is within a single stage as shown in Figure 4.14. In this mode, data is read from external memory, processed iteratively, and written to external memory. Each of the butterfly units operates on a different range of the data; each unit iterates (mostly) independently through all the stages of the computation. All of the butterfly units are used for the entire computation, but the overall throughput is constrained by external memory bandwidth. The number of cycles needed to compute an FFT using this scheme is:

$$T = \frac{2N}{BW} + BL + (\frac{N}{P} + BL)(log_2(N) - 2) \tag{4.7}$$

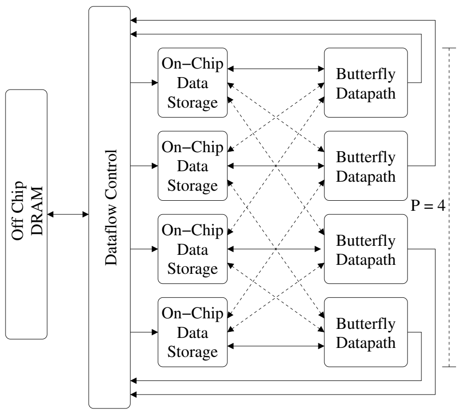


Figure 4.14. Architecture of the Parallel FFT

The first term of Equation 4.7 is the time to read and then write N items based on the memory bandwidth, BW, in terms of complex double precision floating-point items per cycle. The usable bandwidth is limited to the number of units, P. The second term is the latency of passing through the butterfly units during the read from memory. The third term is the time to perform the iterations — using P butterfly units of latency BL for $log_2(N) - 2$ iterations assuming that the first and last iteration are performed as part of reading and writing the data.

All of the FFT designs require internal storage for both data and "twiddle factors" (W_N^{jk}). In the parallel implementation, the data storage requires N double precision complex data items. This one storage area is reused by all iterations and must supply sufficient bandwidth to support the number of parallel processing units; thus, using off-chip memory is not practical. The total number of twiddle factor constants needed is equivalent to $\frac{N}{4}$ storage locations of 16 bytes (complex double-precision numbers).

Pipelined FFT Implementation

At the other extreme, one butterfly unit can be dedicated to each of the stages of the FFT in a pipelined fashion as illustrated in Figure 4.15. Data is read from memory and passed through a series of butterfly units before being written back to memory. Data delays and permutations are needed between each of the stages and between the pipelined FFT unit and DRAM memory. When the number of stages, S, that can be implemented in the FPGA is less than the number of stages needed by the FFT ($log_2(N)$), then $\frac{log_2(N)}{S}$ passes to memory are needed, with a the final pass being a subset, R, of the stages. For each pass to memory, data must be read and written in a particular permutation to optimize the delay and storage requirements in the pipeline (not described here). The number of cycles required to compute an FFT using a pipelined approach is:

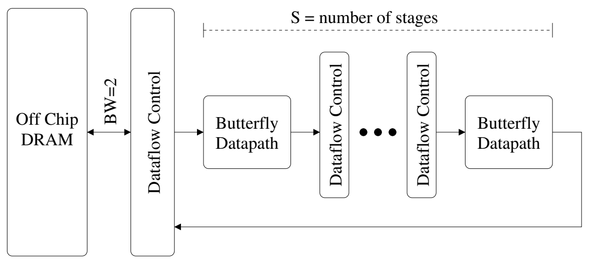


Figure 4.15. Architecture of the Pipelined FFT

$$T = P(S) \times \left\lfloor \frac{log_2(N)}{S} \right\rfloor + P(R) \tag{4.8}$$

$$P(J) = BL \times J + I(J) + \frac{2N}{BW} + (B-1) \times 2^J \tag{4.9}$$

$$I(K) = \sum_{i=0}^{K-1} B \times 2^i \approx B \times 2^K \tag{4.10}$$

$$R = log_2(N) \mod S \tag{4.11}$$

Each pass, $P(J)$, through J butterfly stages (each having a latency of BL) requires the time shown in Equation 4.9. Data dependencies between the stages introduce a delay that doubles at each stage and create a total inter-stage delay given by $I(K)$. The burst length of standard DRAM memories introduce a penalty associated with the burst length, B, to both the interstage delay and a back-end reordering time. The time to retrieve the data from memory and write it back is defined by $\frac{2N}{BW}$; however, a note is in order. The usable bandwidth is limited to one complex double precision floating-point number per cycle per direction due to the limited throughput of the butterfly unit. The final term represents the final pass through a subset of the stages, R, with the corresponding delays.

The pipelined FFT implementation has two significant disadvantages. Foremost, the high latency of the overall pipeline means that many of the butterfly units sit idle while the pipeline is initialized and while it is flushed; thus, if the overall computation is short relative to the pipeline delay, the sustained performance is low. In addition, if the number of FFT stages is not an even multiple of the number of hardware stages, many of the hardware resources sit idle on the final pass. The primary advantages of this implementation are the limited requirement for memory bandwidth and the limited requirement for internal memory for relatively short pipelines. In the prototype implementation, for example, a six stage pipeline is implemented, which more fully utilizes the Virtex-2Pro 100 and only requires a fraction of the memory needed by the parallel implementation.

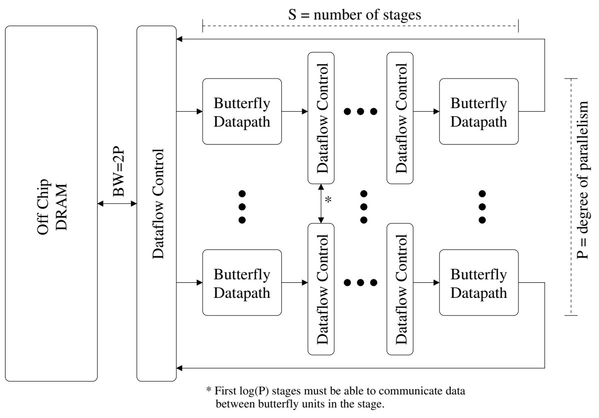

S = number of stages
Off Chip DRAM
BW=2P
Dataflow Control
Butterfly Datapath
Dataflow Control
*
P = degree of parallelism
* First log(P) stages must be able to communicate data between butterfly units in the stage.

Figure 4.16. Architecture of the Parallel-Pipelined FFT

The on-chip memory requirements for the pipelined version are almost entirely dependent on the number of stages. The reordering buffer between butterfly stages is $\frac{3\times 2^i}{2}$ elements, where i is the stage number. This gives a total of $\frac{3\times 2^S}{2}$ items of storage for the items being transformed. Due to the organization of the computation, the twiddle factors require $\frac{3N}{8}$ locations to store the values in a way that provides the bandwidth to provide the twiddle factors to the appropriate butterfly units.

Parallel-Pipelined FFT Implementation

Figure 4.16 is the cross between the two previous architectures. Data moves from external memory, through a set of parallel pipelines, and back to external memory. The first $log_2(P)$ stages must have additional data exchange circuits (for the first pass through the pipeline) as these stages have data dependencies between the pipelines. This approach leverages the ability of the pipelined architecture to reduce bandwidth demands and the ability of the parallel architecture to tolerate shorter input vectors (as well as a wider variety of vector lengths) than the pure pipelined approach. In contrast, the parallel-pipelined hybrid has a higher bandwidth demand than the purely pipelined approach and less tolerance of short vectors than the parallel approach.

The number of cycles to compute the FFT using this approach is the same as that for the pipelined approach. The difference is that it raises the amount of bandwidth that the pipeline can accept and shortens the number of pipeline stages. The overall memory requirements, however, are significantly increased in the near term. The data storage requirements fall to $P \times \frac{3\times 2^S}{2}$. Since the number of stages has been reduced by a factor of P, this is actually a factor of $2^P/P$ fewer items. For the twiddle factors, the storage grows to an upper limit of N storage locations (independent of the number of stages or the parallelism) to provide both the necessary storage and the necessary bandwidth.

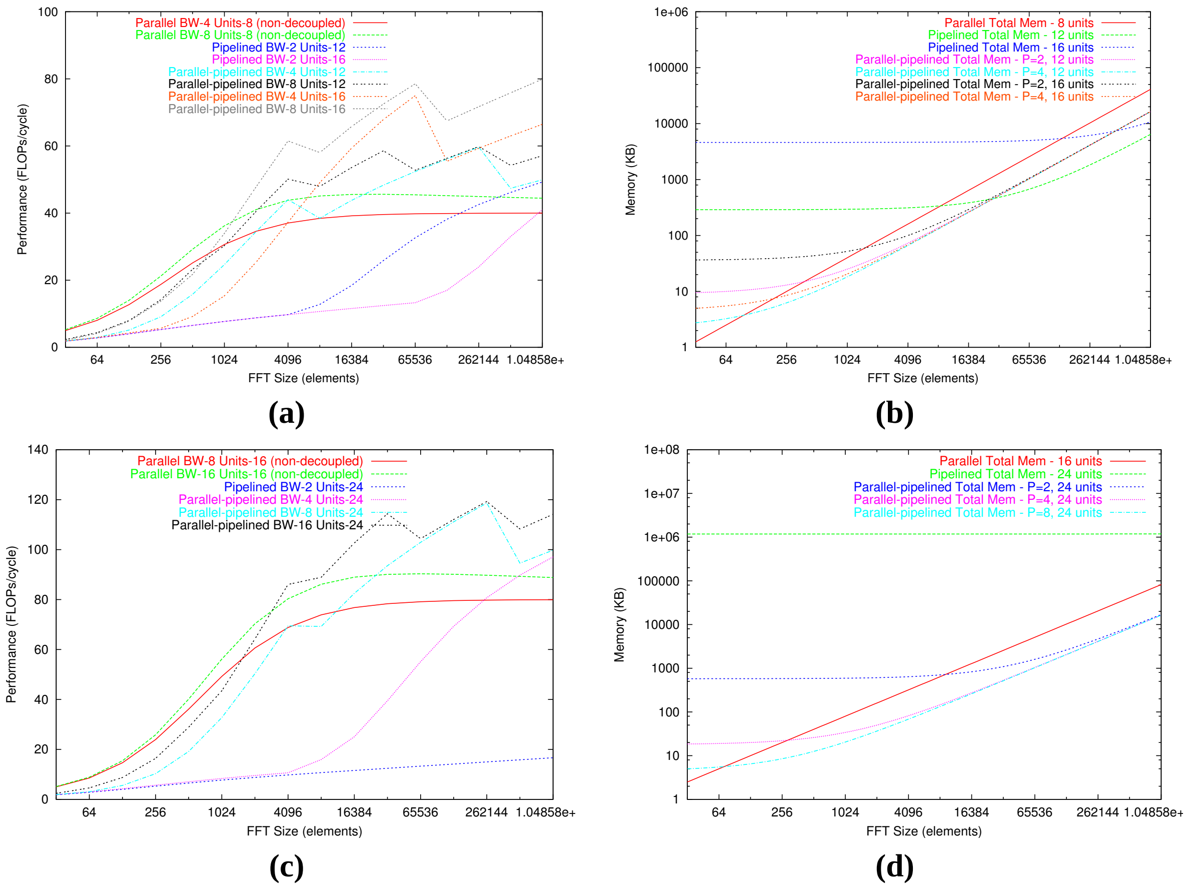


Figure 4.17. A comparison of the **(a,c)** performance and **(b,d)**memory requirement of FFT implementations

Comparison of Architectures

The “right” FFT architecture varies not just with the size of the FFT, but also with the size of the FPGA. This section explores the performance of each of the architectures given different memory bandwidths and the ability to fit varying numbers of units in the FPGA.

Figures 4.17(a) and (b) shows the estimated performance of several possible configurations for a modern FPGA. The most notable result is that the extreme depth of the pipeline combined with the inability to leverage additional memory bandwidth leave the pipelined implementation at a clear disadvantage to the parallel implementation until nearly 256K samples are processed. The only configurations presented for the parallel-pipelined implementation match the memory bandwidth to the number of parallel pipes, *P*. The parallel-pipelined implementations shown have an advantage over the parallel approach because of the depth of each parallel pipe. Finally, Table 4.2 contains the parallelism, *P*, and stages, *S*, for the parallel-pipelined implementations. Correlating Figures 4.17 with Table 4.2 in-

Table 4.2. Parallel pipeline shapes

BW	**Units**	**P**	**S**	**BW**	**Units**	**P**	**S**
4	12	2	6	4	16	2	8
4	24	2	12	4	32	2	16
8	12	4	3	8	16	4	4
8	24	4	6	8	32	4	8
16	24	8	3	16	32	8	4

Table 4.3. Memory requirements summary

Arch	**Min Memory**	**Max BW**
Parallel	$N+\frac{N}{4}$	P
Pipelined	$\frac{3\times2^S}{2}+\frac{3N}{8}+(B-1)\times2^S$	2
Hybrid	$P\times\frac{3\times2^S}{2}+N+(B-1)\times2^S$	2P

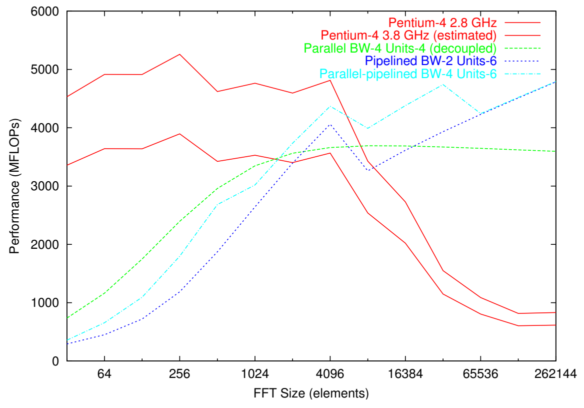
dicates that arrangements which are more "square" have better performance.

In terms of memory requirements, the maximum size of an FFT that can be processed is approximately 128K points. This levels out because more and more internal memory is required to provide the bandwidth needed for all of the butterfly units. FPGA memories tend to run at the same frequency as the logic; thus, more ports are needed to feed more units. This is similar to, but different from microprocessors where the number of units is small and, therefore, so is the number of ports from cache.

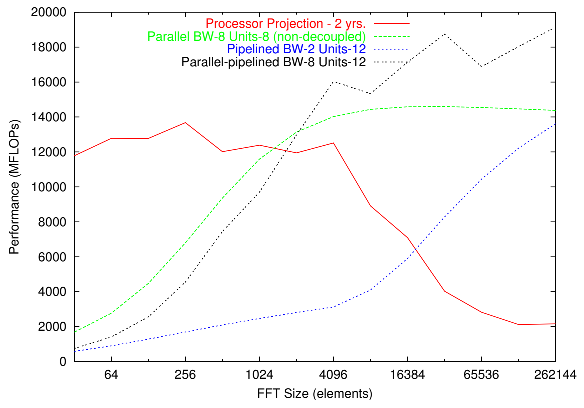
Moving out one generation yields 16 units for the parallel path and 24 units for the pipelined approaches. The results for these are shown in Figures 4.17(c) and (d). As can be seen from these figures, the purely pipelined approach no longer makes sense in terms of performance for reasonable values of N or in terms of memory. It is also unable to use the memory bandwidth that will be available in this time frame. The growth in memory for the three architectures is summarized in Table 4.3. Narrow values of P for the parallel-pipelined version have significant disadvantages as the number of units increases. At equivalent bandwidths, the parallel-pipelined approach does not outperform the parallel approach until N is over 2048.

FPGAs vs. CPUs

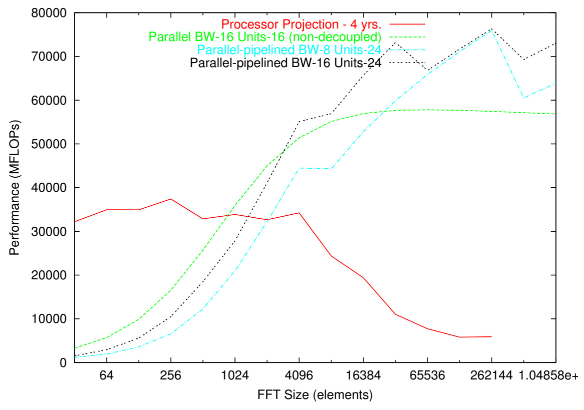
Figure 4.18 compares estimates of performance for FPGAs and CPUs over the next few years. The data for the 2.8 GHz Pentium-4 is from [9] using the Intel MKL. The 3.8 GHz Pentium-4 data was estimated using the ratio of the clock frequencies. Beyond that, CPUs are assumed to double in performance every 18 months, following the widely accepted



(a)



(b)



(c)

Figure 4.18. FPGAs vs. CPUs on current technology (2005) **(a)**, in two years (2007) **(b)**, and in four years (2009) **(c)**

corollary to Moore’s Law (starting with the 3.8 GHz Pentium-4). Initial versions of the prototypes on FPGAs indicate that the design will run at 160 MHz on a Virtex-2Pro 100-6 and FPGAs are assumed to double in clock frequency and area every two years following historical trends[37].

The size of a single FFT must currently approach 8K items for the FPGA to outperform a microprocessor and microprocessors currently have a dramatic advantage for small FFTs. Within two years, the FPGAs should have an advantage for FFTs as small as 1024 items, and within 4 years, that advantage is expected to be dramatic for larger sizes. Note that the far right hand side of the graph is not particularly fair since FPGAs do not have the internal memory to achieve that level of performance. Also, the elbow in the CPU curve will move to the right in future generations as the cache size grows; however, the comparison does not change as the CPU curve is still flat at 1024 elements while the FPGA performance is still growing.

Analysis Summary

The analysis of these three kernels indicates that FPGAs can sustain a very high percentage of peak performance over domains of interest. For cache oriented operations such as DGEMM, FPGAs are able to sustain a higher fraction of peak performance than many microprocessors while using less internal memory and less memory bandwidth. This is more a function of the programmability of the internal storage (rather than using it as a cache) than it is a special property of the FPGA itself. In contrast, DGEMV performs so well on FPGAs because FPGAs offer more memory bandwidth than microprocessors. This is not because FPGAs exploit a superior technology. Instead, it is a simple matter of economics: microprocessors cannot afford to deliver bandwidth that the majority of their consumers do not need. The FPGA market, in contrast, requires many programmable I/O pins that can be dedicated to any purpose. In the case of scientific computing, many of those would be dedicated to memory bandwidth. Finally, for FFT, FPGAs are somewhat constrained by their high floating-point unit latency for single, small FFTs. For large FFTs, however, FPGAs again demonstrate an advantage delivered by higher memory bandwidth.

Chapter 5

Implementations on a Real World System

A key step in predicting how a technology can be utilized in the future is evaluating how well it performs today. In order to gain a low-level understanding of current-generation reconfigurable computing issues, the California LDRD team purchased (with non-LDRD funds) a leading-edge HPC system with FPGA resources and focused on examining techniques by which applications could take advantage of the hardware. After investigating several commercial HPC platforms, we decided that the Cray XD1 provided an architecture that best met our interests. Following initial experiments with the XD1, we proceeded to adapt four different algorithms to make use of the XD1's FPGA hardware.

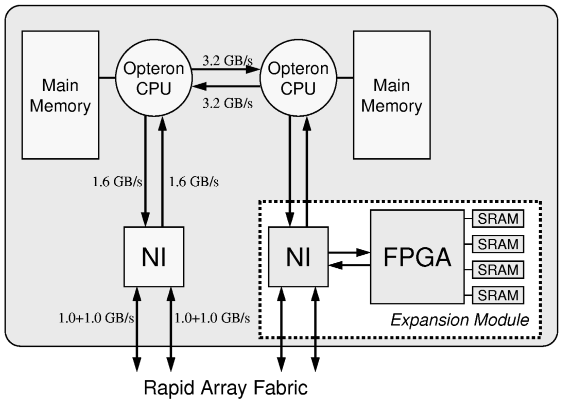
The Cray XD1

The Cray XD1 (Figure 5.1(a)) is an appealing arechitecture for reconfigurable computing research because it provides a dense multiprocessor system that features both host processors and FPGA accelerators. A single rack of an XD1 system houses six independent compute blades. As illustrated in Figure 5.1(b), each blade is equipped with two AMD Opteron processors, DDR memory, a hard disk, and a network interface (NI) that enables the blade to communicate with other blades through an InfiniBand-like network[1]. In order to maximize network performance, the NI is connected to the blade's CPUs through a HyperTransport link that is 1.6 GB/s in both directions.

Cray also produces an optional FPGA accelerator card that can be plugged into a proprietary HT connector on an XD1 compute blade. This accelerator card provides an additional NI for interactions with a secondary communication network and a user-programmable FPGA that can be utilized as a computational accelerator by applications running on the blade. In order to allow a sizable amount of data to be stored in close proximity to the FPGA, the accelerator card is equipped with four 4 MB banks of QDR2 memory that are connected directly to the FPGA. The fact that the memory is QDR2 enables an FPGA ap-

[1]While the XD1 utilizes InfiniBand hardware at the physical layer, Cray has implemented its own communication substrate called Rapid Array on top of the hardware. This communication software is incompatible with existing InfiniBand hardware/software.

(a)



(b)

Figure 5.1. (a) A single rack of a Cray XD1. (b) A high-level representation of the XD1 architecture.

plication to read 64-bits of data *and* write 64-bits of data to each memory bank every clock cycle, at speeds of up to 200 MHz.

Development Components

Cray provides two *closed-source* hardware components (i.e., cores) for simplifying the construction of FPGA accelerator hardware. The first of these cores is a HT communication engine for managing interactions with the host blade's memory system. This core handles low-level transactions on the HT bus and steps DDR signals down to a wider, single data rate (SDR) form that is easier for end users to work with. The HT core provides read and write interfaces for both FPGA-initiated requests and HT fabric-initiated requests. Unfortunately, this core does not hide all of the low-level details about HT from the user[2]. As a means of simplifying development, we have constructed our own DMA transfer engine that provides a continuous view of host memory for FPGA users. The second core provide by Cray implements a memory interface for accessing the accelerator card's external QDR2 memory. This core hides the low-level details of dual-ported DDR memory from the user and is straightforward to utilize. In addition to these cores, Cray provides a set of VHDL reference designs that demonstrate how the FPGA can be utilized by end applications in the XD1 environment.

In terms of RC software, Cray provides a basic library for controlling low-level interac-

[2]For example, the user is limited to a maximum of eight 64-bit words in a transfer, and transfers cannot cross a segment boundary of eight 64-bit words.

tions with a compute blade's local FPGA. This software allows users to program different hardware configurations into the FPGA as needed by the application (as opposed to automatically programming the FPGA on-demand). Unfortunately, the software does not implement a robust access control system for the FPGA resources. As a result, users are expected to work in a cooperative manner that time-shares use of the FPGA. This approach is fairly common among RC platforms, and is not a major concern in most research environments.

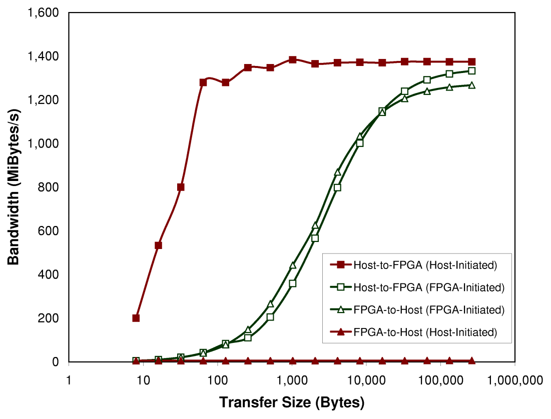
CPU-FPGA Data Transfers in the XD1

RC performance in an HPC platform often hinges on the system's ability to rapidly move data between a host application and an FPGA accelerator. Without a high-bandwidth, low-latency connection to the host CPU, an FPGA can only be utilized in coarse-grained applications. Fortunately, the XD1 architecture provides a full duplex connection between the CPU and accelerator that in theory is capable of supporting simultaneously transferring 1.6 GB/s (1.4 GB/s after overhead) in each direction.

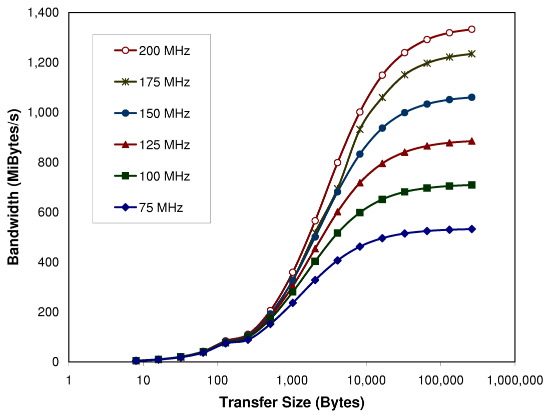
There are three ways that data can be moved between a host application and an FPGA accelerator in the XD1. First, a host application can read or write one of a limited number of FPGA application registers through a system call that is provided by the FPGA's device driver. While this interface is slow (a single write may take as long as 500 ns to execute), operations are guaranteed to take place in the order that they are issued. Second, an application can write/read data to/from the FPGA's address space using memory-mapped I/O techniques. This method provides exceptional write performance because write-combining allows individual I/O operations to be bundled into bursts. However, users must be aware of write consistency issues that arise from write combining. Finally, the FPGA can be instructed to issue DMA requests to physical memory. In order to facilitate this work, the FPGA's device driver allocates and pins a 2 MB block of contiguous memory for 1-copy transfers, and provides virtual-to-physical address translation functions in order to reference the memory.

We conducted two experiments to observe the low-level performance characteristics associated with transferring data between a CPU and an FPGA on an XD1 blade. In the first experiment we measured the bandwidth that could be obtained when transferring different data sizes between the host CPU and FPGA. We performed both read and write tests using both host-initiated transfers and FPGA-initiated transfers. As Figure 5.2(a) illustrates, performance is always best when push-based mechanisms are used to transfer data from one resource to another. Host-initiated writes yielded the best overall performance, achieving half the maximum bandwidth at message sizes of only 64 bytes.FPGA-initiated transfers had similar performance for larger message sizes. As expected, host-initiated reads performed abysmally.

The second experiment examined the impact of the FPGA's clock rate on data transfer performance. In this experiment, we measured the amount of time required for an FPGA to



(a)



(b)

Figure 5.2. (a) Read/Write performance for host-initiated and FPGA-initiated data transfers. (b) FPGA write performance for different FPGA clock speeds.

write different length messages to host memory. We then ran the benchmark at different FPGA clock speeds. As Figure 5.2(b) illustrates, reducing the clock rate of the FPGA decrease the rate at which the FPGA can write data to the host. This characteristic implies that it is important for developers to optimize their FPGA designs to use as fast of a clock as possible. While a design with a slower clock rate might be capable of performing more work per clock cycle, doing so may have the negative side effect of throttling the rate at which data can be exchanged between the CPU and FPGA, and thus limit overall performance.

Application Examples

In order to observe low-level performance aspects of the XD1, we adapted four different applications to utilize the XD1's FPGAs as computational accelerators. In all of these experiments, we assume the following:

- **Single CPU/Single FPGA:** While the XD1 has multiple compute resources, we focus on the situation where a host processor offloads a key operation to its local FPGA.
- **Speedups are Relative to the XD1's CPUs**: While faster CPUs are available, we report all speedups relative to the XD1's 2.2 GHz AMD Opteron processors. This choice creates a level playing field for benchmarks.

- **Single-Precision Floating Point:** Because our XD1 is only equipped with mid-sized (V2P50) FPGAs, we decided to implement all four algorithms using single-precision floating-point computations. This choice enabled us to consider larger computational kernels than would be possible if we utilized double-precision arithmetic. Additionally, an analysis of the applications we selected revealed that only one application (STIFF) truly suffered from utilizing reduced precision.

- **Transfers Times Included in Overhead:** For fairness to application developers, we perform all FPGA timings from the perspective of the software application. This viewpoint means that the time required to exchange data with the FPGA is included in the amount of time required for the FPGA to perform the work.

- **Transfer Times Include Ejection**: The 1.1 release of the XD1's software has a limited amount of memory for hosting data that is ejected by the FPGA. We include the extra step of transferring data from this memory to application memory (i.e., 1-copy) in the timing measurements in this chapter.

- **FPGA Build Tools:** In this work we utilize Xilinx's ISE 6.3 tool chain to build FPGA hardware. This tool chain invokes the Xilinx Synthesis Tool (XST) to perform front-end synthesis. While XST produces lower-quality results than other synthesis tools, it is widely available and has proved to be stable enough for our needs.

The four computational kernels that we have adapted to the XD1's FPGA hardware are the following: k-Nearest Neighbors (kNN), Stiffness Matrix generation (STIFF), Ray-Triangle Intersection (RTI), and Isosurfacing (ISO).

k-Nearest Neighbors (kNN)

In many pattern-matching and machine-learning applications it is necessary to categorize an input vector based on its similarities to different training vectors that have already been classified. One approach to performing this work is to apply the k-nearest neighbors (kNN) algorithm, which (1) computes the Euclidean distance between the input vector and each training vector, and then (2) locates the k training vectors that have the smallest distances to the input vector. Pseudo-code for this algorithm is presented in Figure 5.3.

```
for(i=0; i<NUM_TRAINING_VECTORS-1; i++) {
    sum = 0
    for(j=0; j<NUM_TERMS-1; j++) {
        diff = input[j] - training_vector[i][j]
        sum = sum + diff*diff
    }

    if(sum < best_results[k-1]) {
        store_in_sorted_list(best_results, sum)
    }
}
return best_results
```

Figure 5.3. The pseudo-code for the kNN algorithm.

While kNN produces high-quality results, researchers must often resort to inferior algorithms because kNN is an expensive operation to perform on large-scale data sets. Real-world problems often employ millions of training vectors with dozens of terms. As such, kNN can have a sizable runtime because it must compute the distance from every input vector to every training vector. Based on our analysis of the kNN algorithm, it is clear there are both opportunities and challenges involved in constructing a hardware accelerator for this algorithm.

Opportunities:

- **Many Parallel Operations:** The kNN operation performs a simple operation on a large number of inputs. Therefore it is possible to perform a sizable number of computations for this algorithm in parallel.

- **Static Training Data:** It is expected that the training data will be held constant for a long period of time while checking many different input vectors against the data. This characteristic enables us to store training data in memory close to the FPGA and reuse the data for multiple operations.

- **Few Inputs/Outputs:** Once the FPGA accelerator board is loaded with training data, the user only needs to inject a single input vector to begin useful work. Likewise, the algorithm only generates k results for each input vector. As such, the overhead of transferring data between the FPGA and host CPU is minimal compared to the work that is performed at the FPGA.

Challenges:

- **Data Throughput:** Performance in kNN is largely dependent on the rate at which training data can be brought into the system. It is imperative that the FPGA fetch as many data values from memory as possible (i.e., minimize memory stalls) and operate at as high of a clock rate as possible.
- **Adaptability:** Machine-learning/pattern-matching users expect that a kNN kernel will be capable of handling a variety of parameters (e.g., the number of terms in a vector or the number of vectors in the training set). Therefore, it is beneficial if an accelerator can support multiple parameters without requiring that the hardware be recompiled.
- **Pipeline Latency:** The kNN performs a running summation, which is nontrivial to implement in systems that utilize pipelined adders. Therefore it is necessary to consider techniques by which processing dependencies can be removed.

Our goal in adapting kNN to a hardware implementation was to store as many training vectors as possible in close proximity to an FPGA, and employ an array of distance-computation processing elements (PEs) in the FPGA to exploit the natural concurrency of the algorithm. In order to maximize memory bandwidth, each PE operates in a streaming manner, processing each term in a training vector sequentially. We assume that all training vectors are the same length (specified by software at run time) and that individual vector terms are single-precision floating-point values (32-bits). Based on the fact that the XD1's FPGA can fetch 256-bits of data per clock cycle from the on-card QDR2 memory banks, it is possible to place eight PEs in parallel in the FPGA design.

A high-level representation of the hardware is depicted in Figure 5.4. As this illustration indicates, each PE is comprised of three floating-point units, a k-value sorting unit, and a feedback buffer for maintaining intermediate results. All PEs operate in lock step. Each PE computes the difference between an input vector term and a training vector term, and then adds the square of the difference to a running sum. The sorting unit maintains a list of the k smallest distances observed for an input vector (as well as the corresponding training vector identifiers). Once the PEs have stepped through all training vectors, the k results of each PE are merged using a final k-value sorting unit. The k best best results are then transferred to the host's memory as output.

The fact that the floating-point units are pipelined forced us to rearrange the flow of execution in the PEs in order to implement the running summation in an efficient manner. Rather than process all of the terms of a vector before moving on to the next vector, we decided to process a term for all vectors before moving on to the next term. While this approach requires that the intermediate results for each vector must be buffered in the PE,

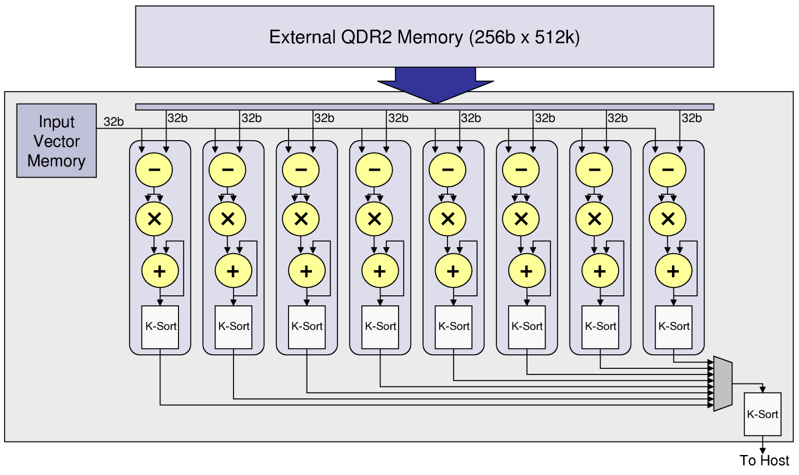


Figure 5.4. Architecture of the kNN Hardware Accelerator

the running-sum dependency is removed and the PE can stream through input values very efficiently. The PEs are also equipped with multi-pass functionality to handle the case where the number of training vectors stored in the PE's section of QDR2 memory exceeds the number of intermediate values that can be buffered in the PE. The modified version of the algorithm is presented in Figure 5.5.

```
00: for(i=0; i<NUM_TERMS-1; i++) {
01:
02:     for(j=0; j<NUM_TRAINING_VECTORS_IN_SET-1; j++){
03:         diff    = input[i] - training_vector[i][j]
04:         sums[j] = sums[j] + diff*diff
05:     }
06: }
07:
08: sort(sums)
09: return sums[0:k-1]
```

Figure 5.5. The pseudo-code for the modified kNN algorithm.

A number of experiments were conducted on the XD1 to observe the performance of the hardware accelerator for various scenarios. In these tests we adjusted the number of terms in a vector and the number of training vectors loaded in the FPGA's on-card memory. We then measured the amount of time required for an application to perform the kNN operation first in software and then using the hardware accelerator. The results are reported in terms of speedup when utilizing the hardware accelerator.

The results of the experiments are presented in Figure 5.6 for vectors with 2-60 terms. These particular experiments yielded hardware speedups ranging from 1.5x to 6.5x. In

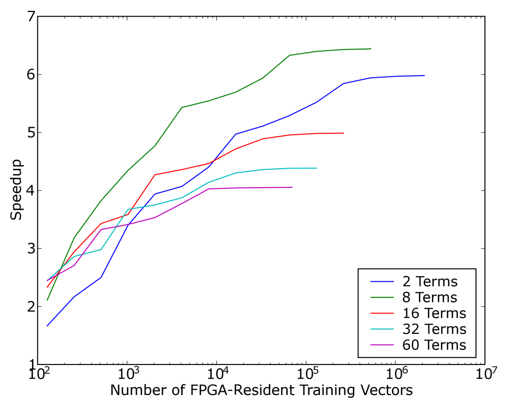


Figure 5.6. Performance gain of FPGA accelerator for kNN.

terms of training vector data set sizes, performance improved as the number of training vectors stored in on-card memory was increased. This effect can be attributed to the fact that CPU-FPGA communication overhead is less of a factor in larger jobs than it is in smaller jobs. In terms of vector length, the best performance was observed when the algorithm utilized vectors with 8-12 terms. Increasing the number of terms degrades performance for two reasons. First, the FPGA's on-card memory has a fixed capacity for housing training data. Therefore, increasing the number of terms in the vectors decreases the number of vectors that can be stored close to the FPGA, which in turn limits the size of the job that can be performed without reloading the memory. Second, the current implementation of the kNN accelerator requires a few idles cycles to take place between the processing of each term. Thus, increasing the number of terms increases the total number of idle cycles that take place in a job. These idle cycles present an opportunity for improvement in future work.

Additional experiments were conducted to assess the complexity of converting the kNN architecture to utilize data formats other than 32-bit floating-point values. We replaced the floating-point units with 32-bit integer pipelined units and recompiled the design. In addition to requiring approximately half the resources of the floating-point design, the integer design achieved speedups of up to 8x over software. Based on these characteristics, a second row of PEs could be added to the design to process a second input vector in parallel with the first. While this approach would require the user to submit multiple jobs at a time, it is conceivable that performance would scale with the number of rows that could be instantiated in the FPGA.

Stiffness Matrix (STIFF)

In many finite element (FE) applications, it is necessary to compute the stiffness matrix for each node in a mesh for every time step in the simulation. A high-level representation of an algorithm that can be used to generate the stiffness matrix for a single node is presented in Figure 5.7. While this algorithm is a straightforward computation, the fact that a simulation may operate for tens of thousands of time steps and utilize meshes with millions of nodes can result in a significant workload for the application. Therefore, it is beneficial to consider techniques by which this work can be performed more efficiently.

An examination of the stiffness matrix computation reveals that there are both opportunities and challenges to be considered when implementing the algorithm in an FPGA accelerator.

Opportunities:

- **Many Parallel Operations:** The algorithm employs a large number of independent operations that can be executed in parallel. This characteristic provides the hardware designer with many possibilities for optimization.
- **Meta-Operations:** The algorithm performs the same set of computations on a variety of data inputs. As such it is possible to group a set of computations together as a "meta-operation" and then reuse a set of floating-point units over time to implement different parts of the algorithm.
- **Few Inputs/Outputs:** While the algorithm maintains a fair amount of data internally, there are few input and output data values. This characteristic is desirable because it minimizes the amount of data that is exchanged between the host CPU and the FPGA.
- **Many Constants:** The algorithm utilizes a large number of data constants throughout the algorithm. These constants can be stored efficiently in on-chip ROMs that are placed in close proximity to the floating-point units. These simplifications to the hardware data path can result in improvements to the hardware's overall clock rate.

Challenges:

- **Many Operations:** The sheer number of operations performed in the algorithm makes it impractical to instantiate a unique floating-point unit for every computation in the algorithm. Therefore, we must consider design techniques that reuse a fixed number of floating-point units to perform different operations over time.
- **Multiple Sections:** At a course-grained level the algorithm has two computational sections that are performed sequentially. While we can pipeline the work through the two sections, the second section has more operations and thus a different processing rate than the first.

```
dNdr[4][3][3] = dNdr_constants    //Constants
dNds[4][3][3] = dNds_constants    //Constants
NU[2]         = NU_constants      //Material properties constants
OM[2]         = OM_constants      //Material properties constants
qw[3]         = QW_constants      //Material properties constants
detJ[3][3]    = zeros(3,3)        //Determinant for each Jacobian matrix
J[3][3][2][2] = zeros(3,3,2,2)    //9 Jacobian matrices
G[8][3][3]    = zeros(8,3,3)      //8 Gloms matrices
k[8][8]       = zeros(8,8)        //1 result matrix

// Initialize the 9 Jacobian matrices (one for each quadrature point)
for(a=0;a<3;a++)
  for(b=0;b<3;b++) {

    for(p=0;p<2;p++)
      for(q=0;q<2;q++)
        for(i=0;i<4;i++)
          if(p==0) J[a][b][p][q] += dNdr[i][a][b] * coords[i][q];
          else     J[a][b][p][q] += dNds[i][a][b] * coords[i][q];

    detJ[a][b] = cross_multiply(J[a][b]);
  }

// Compute Matrix Gloms
for(a=0;a<3;a++)
  for(b=0;b<3;b++)
    for(i=0;i<8;i++)
      G[i][a][b] = J[index1]*dNdr[index2] + J[index3]*dNds[index4];

// Compute Stiffness Matrix
for(a=0;a<3;a++)
  for(b=0;b<3;b++)
    for(i=0;i<7;i++)
      for(j=i;j<7;j++)
        k[i][j] += G[index1]*G[index2]*NU[index3] +
                   G[index4]*G[index5]*OM[index6]*qw[a]*qw[b]/detJ[a][b];

//Return k[0:7][0:7]
```

Figure 5.7. Pseudo-code for the stiffness matrix computation at a single mesh point. Matrix indices are omitted in this listing for brevity.

- **Long Loops:** The second section of the algorithm performs 36 meta-operations over 9 iterations. The large number of meta-operations in an iteration discourages simple loop unrolling techniques.

- **Feedback:** The second section of the algorithm has a feedback loop that sums the result of each meta-operation with the previous iteration's result. This feedback can be challenging to implement efficiently given that the floating-point units are pipelined.

- **Single-Use Operations:** While most operations in the algorithm can be mapped to a small set of meta-operators, the determinant computation (line 20) is only performed once per Jacobian matrix. As such, it is likely that the hardware performing these operations will be idle for many clock cycles in an iteration.

Multiple implementation strategies were considered for the FPGA design. Estimating that only 50 floating-point units could be instantiated in the XD1's V2P50 FPGA, we decided that it would be impractical to unroll the loops or merge them into one continuous computational block. The availability of multiple computations that could be grouped into meta-operations motivated us to consider techniques by which a set of hardware units could be reused to perform a series of similar meta-operations. In particular we grouped the calculations into four meta-operations: the Jacobian (lines 17-18), the determinant (line 20), the modification matrices (line 27), and the stiffness matrix (lines 34-35). We then partitioned the design into a front-end unit and a back-end unit. While both units perform 9 iterations per job, the front-end unit performs 8 operations per iteration while the back-end performs 36 operations[3]. As such, we constructed a hardware design that utilizes one front-end unit and multiple back-end units to improve throughput, as depicted in Figure 5.8. Once the front-end has processed a job, it passes the work on to the next available back-end. This assignment is always sequential due to the deterministic nature of the operations. Based on the workload, a balanced design has 4-5 back-end units.

A hardware accelerator design employing single-precision floating-point units was implemented for the V2P50 FPGA on the Cray XD1. Through experimentation with the hardware compilation tools we determined that the V2P50 has the capacity to house a design with one front-end and five back-end units. The design tools reported that the hardware was capable of operating at a maximum frequency of 190 MHz. This speed is the highest of all four kernels that were constructed in this project. The high speed of the design can be attributed to the fact that the large number of data constants used by the algorithm enabled the hardware data path to be simplified.

A software library was constructed in order to allow applications to take advantage of the STIFF hardware accelerator. This library provides communication mechanisms for exchanging data with the FPGA, as well as a software implementation of the algorithm for comparison purposes. After basic experiments were performed to test the validity of the accelerator, we optimized the software library to provide better communication performance

[3]While the output is an 8x8 matrix, the result data is in the form of an upper-triangle matrix. Thus, many operations can be eliminated.

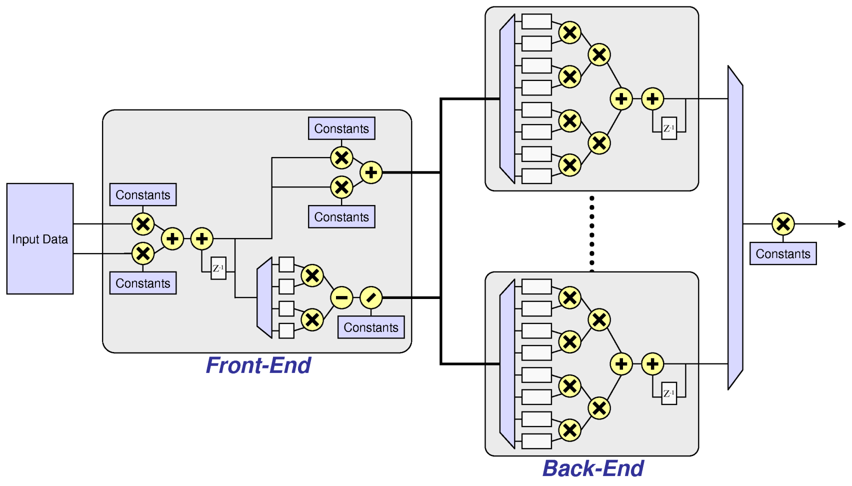


Figure 5.8. Architecture of the STIFF Hardware Accelerator

with the FPGA. Specifically, we constructed a second function call for injecting a batch of jobs at a time. This function copies all of the data for a batch of jobs to the FPGA and then posts a single update to the FPGA's control register for the work. This approach allows the expensive notification operation (500 ns) to be amortized across multiple jobs and results in better performance.

Performance measurements were conducted to observe the speedup of the accelerator when compared to a software implementation of the algorithm. The performance measurements captured the amount of time required by the end application to perform a large number of stiffness computations. For the design with five back-ends, we observed that the STIFF accelerator provided a 4x gain in performance over the software implementation. Utilizing more than five back-ends with only a single front-end did not yield additional performance gains.

Ray-Triangle Intersection (RTI)

Photon mapping[19] is a global illumination technique employed in visualization to render high-quality images of three-dimensional scenes. Photon mapping applications inject millions of photons of light into a scene and then track each photon's path as it reflects off objects in the scene. For each photon reflection the application must determine which object a photon will collide with next and where the intersection takes place. This operation can be accomplished by (1) performing ray-triangle intersection (RTI) computations between a photon and all of the triangles the photon can possibly collide with and then (2) selecting the intersection point that is closest to the photon's current location. Given the sheer number of photons and objects in a scene, it is important for RTI computations to take place as rapidly as possible.

```
vertex   T0xyz, T0xyz, T2xyz          // Triangle Vertices
vertex   PHOxyz, PHDxyz               // Photon Origin and Direction


S0xyz = T1xyz - T0xyz
S1xyz = T2xyz - T0xyz

C0xyz = CROSS PRODUCT( PHDxyz , S1xyz )
DET   = DOT PRODUCT( S0xyz , C0xyz)

if (DET != 0) {
    INV_D  = 1 / DET
    S2xyz  = PHOxyz - T0xyz
    U      = DOT PRODUCT ( S2xyz , C0xyz ) * INV_D

    if (0.0 < U < 1.0) {
        C1xyz = CROSS PRODUCT( S2xyz , S0xyz )
        V     = DOT PRODUCT( PHDxyz , C1xyz ) * INV_D

        if (V > 0) AND (U + V > 1.0) {
            T = DOT PRODUCT ( S1 , C1 ) * INV_D
            return HIT at TUV
        }
    }
}
return NO_HIT
```

Figure 5.9. Pseudo-code for the Möller-Trumbore Ray-Triangle Intersection (RTI) Algorithm

The Möller-Trumbore[27] algorithm is a well-known method for efficiently performing the RTI computation. A high-level representation of the algorithm is presented in Figure 5.9. Given a ray and a triangle, the algorithm produces a TUV coordinate if an intersection occurs, where T is the distance from the ray's origin to the triangle and UV is the two-dimensional offset of the intersection within the triangle. With the exclusion of one division operation (line 11), the algorithm only requires additions, multiplications, and comparison operations.

Due to the fact that photon mapping applications can spend minutes to hours rendering a single image, it is worthwhile to investigate how an FPGA accelerator could be utilized to improve performance. In order to hide the overhead of exchanging data with an accelerator, we assume that the photon mapping application operates on a bundle of photons at a time. As presented in Figure 5.10, for each photon in the bundle, the accelerator must perform an RTI operation with every triangle to locate the nearest triangle that is in the photon's path. In addition to a TUV intersection point, the accelerator identifies the triangle that each photon collides with.

```
00: for(p=0; p<NUM_PHOTONS-1; p++){
01:
02:     T_best = infinity
03:
04:     for(id=0; id<NUM_TRIANGLES=1; id++){
05:         (is_hit, T,U,V) = compute_rti(photons[p], triangles[id])
06:         if((is_hit) and (T < T_best)) {
07:           (T_best,U_best,V_best) = (T,U,V)
08:           ID_best = id
09:         }
10:     }
11:     photon_results[p] = (ID_best, T_best, U_best, V_best)
12: }
```

Figure 5.10. Pseudo-code for RTI Work in a Photon Mapping Application.

Through an initial analysis of the algorithm, we determined that the opportunities and challenges for this work include the following.

Opportunities:

- **Multiple Computations:** The Möller-Trumbore algorithm performs a moderate number of floating-point operations in a predictable manner. As such, it is straightforward to rearrange the algorithm into a form where it can be implemented as a continuous, deep pipeline of operations.
- **Static Scene Data:** It is safe to assume that scene geometry changes infrequently in the photon mapping application. Therefore, scene data can be loaded into the FPGA's on-card memory at the beginning of a new video frame and then utilized in work involving many different bundles of photons.
- **A Large Number of Jobs:** The "light propagation" phase of a photon mapping algorithm sends hundreds of thousands of rays (or more) into a scene, each of which may interact with up to 20 or 50 surfaces. The resulting workload provides an opportunity to hide data transfer overhead through batch processing.

- **Multiple Iterations per Job:** A substantial number of RTI computations must be performed for each job because each photon in a job must be interested with each triangle in the job. This large volume of work provides us with opportunities to hide the computational latency of the RTI pipeline and achieve higher utilization of the floating-point units.

Challenges:

- **High Resource Utilization:** With over 50 floating-point operations, the RTI algorithm requires at least a medium-sized FPGA in order to be possible. While the XD1's V2P50 is just large enough for the design, it is important to note that the maximum clock rate for a design generally decreases as an FPGA reaches capacity.

- **High Complexity:** The use of a deep computational pipeline with a high utilization rate implies that a large number of data values are in-flight at any given time. As such, there is a significant amount of complexity associated with the hardware, which in turn affects the development and debug time.

Based on these observations, we devised a strategy for implementing an FPGA accelerator for photon mapping applications. Our goal was to implement the RTI as a deep pipeline of floating-point units, and then provide wrapper hardware that minimizes the amount of time the pipeline is idle. In terms of memory resources, we decided to house photon data in on-chip memory and triangle data in external on-card memory. This decision was motivated by the fact that a scene may be comprised of millions of triangles of data and that it is useful to store as many triangles near the FPGA as possible. The downside of this choice is that the XD1's external memory is not wide enough to supply a triangle's three vertices (288 bits) in a single fetch operation. Thus, in order to keep the pipeline busy, we had to reverse the order of the loops in Figure 5.10. The new system fetches a triangle and then performs the RTI computation between the triangle and all photons. A high-level representation of the accelerator architecture that we constructed is presented in Figure 5.11.

At the heart of the accelerator is a deep pipeline of floating-point units that implement the RTI computation. This pipeline is comprised of three sections. First, an intersect unit is used to perform the vector computations required in the algorithm. This unit performs three vector subtracts, two cross products, and four dot products for each RTI computation. In order to minimize the number of pipeline stages in this unit, we defer the algorithm's division operation (line 11 in Figure 5.9) until the third section of the pipeline. Second, a compare unit determines whether an intersection is better than a previously discovered result for the photon in question. Because the intersect unit does not perform division, the compare unit operates using a numerator-denominator representation of the numbers. Information about the nearest intersection point for each photon is stored in a feedback buffer connected to the comparison unit. Finally, in the third section of the pipeline, floating-point division units are utilized to convert the numerator-denominator representations of the results to their final TUV values.

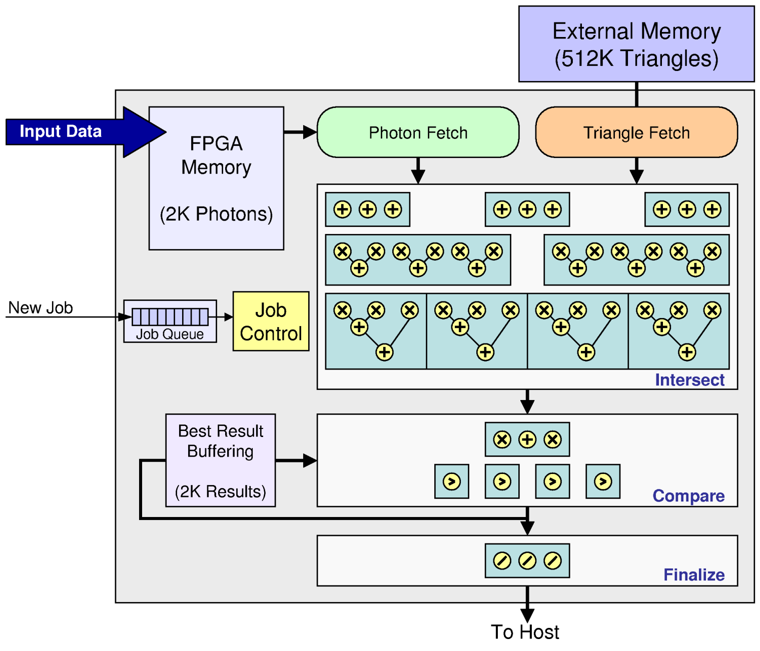


Figure 5.11. Architecture of the Ray-Triangle Intersection hardware accelerator

The hardware wrapper around the RTI pipeline is designed to maximize the rate at which work is issued to the pipeline. In terms of memory resources, photons are issued from an internal memory bank while triangles are issued from external memory. Rather than implement the photon memory bank as a hardware FIFO queue, we opted for an open memory interface controlled by the host application. When injecting a bundle of new photons into the memory, the host application determines where the bundle should be written and then provides a start and a stop address for the bundle to the FPGA. This approach simplifies control hardware in the FPGA and allows for more flexibility in the host application. For the triangle memory, we opted for a simple compression scheme that allows vertex coordinates to be reused by different triangles. In this scheme the first three memory banks of the FPGA's on-card memory house vertex data, with XYZ data being striped across the three banks. The remaining memory bank holds triangle definitions, where each of a triangle's three vertices is represented by an offset into the vertex memory. Based on the 16 MB QDR2 capacity of our XD1, this system facilitates up to 1M unique vertices or up to 512K triangles. The triangle memory interface is also double buffered to enable the next triangle to be fetched from memory while the RTI pipeline processes the current triangle.

Application software was constructed to validate the accuracy and benchmark the speed of the accelerator. This software loads a large number of triangles into the FPGA's QDR2 memory and then measures the amount of time required to process jobs with different numbers of photons and triangles. Results are then compared to those generated by a software module that implements the same functionality. In our benchmarks, we allowed the number of photons in a job to range from 4 to 1023, and the number of triangles to range from 8 to 4096. Two different views of the speedup observed in the same experiment are presented in Figure 5.12.

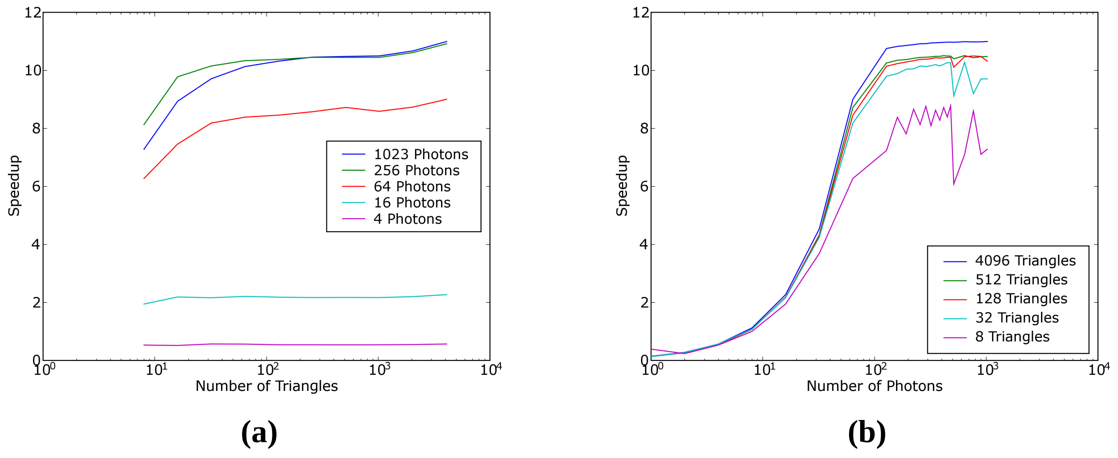


Figure 5.12. Speedup of the RTI Hardware Accelerator over Software

From this experiment we first observe that the speedup of the hardware accelerator improved when more photons and/or triangles are processed in a job. From (a) we observe that for a fixed number of photons, the majority of the speedup can be obtained using only a small number of triangles (approximately 32-64). This property is beneficial because it allows software developers to break jobs into smaller regions of geometry and still obtain a substantial performance gain. In terms of photons, we observe from (b) that the majority of speedup for a fixed number of triangles is obtained when jobs with more than 64 photons are utilized. This pivot point can be attributed to the depth of the compare section of the pipeline (i.e., the feedback buffer requires that the first few results of the first iteration be written before the second iteration begins). The fact that substantial performance gains are possible with a small number of photons is important, because it implies that host software does not have to gather large bundles of photons together in order to make use of the accelerator.

In additional experiments we increased the maximum number of photons in a bundle to 4095 and were able to obtain a 12x speedup over software. While additional optimizations could be applied, we believe that in order to obtain more performance, it is necessary to move to a larger capacity FPGA that could house multiple instances of the photon mapping accelerator.

Isosurfacing (ISO)

Many scientific applications generate a large volume of output data for every time step in the simulation. In order to better explore these results, researchers often utilize post-processing visualization techniques that highlight relevant features in the data and represent regions of interest in a more insightful, graphical form. One such visualization technique that is frequently used to interrogate a volume of data is isosurfacing.

Isosurfacing algorithms approximate the set of all points in a region of space where some field of interest is a constant. For example, isosurfaces of computed tomography (CT) or magnetic resonance imaging (MRI) scans (Figure 5.13) approximate surfaces of constant density (CT) or magnetic dipole alignment (MRI). Typically, isosurfacing algorithms approximate this set of points with a set of triangles.

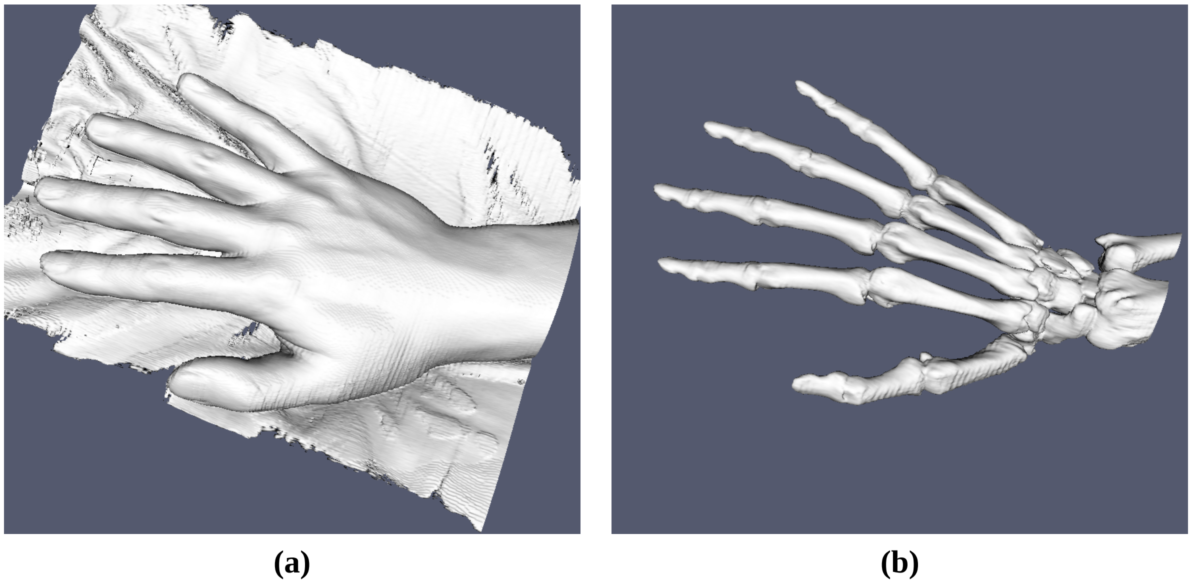


Figure 5.13. In this 512x512x210 data set, we obtained (a) exterior and (b) interior views of a hand by isosurfacing with different threshold values. *MRI data courtesy of Serge VAN SINT JAN, Ph.D., Associate Professor, Université Libre de Bruxelles.*

Medical images and many physical simulations (especially computational fluid dynamics) store fields over a regular, three-dimensional grid of points. The grid of points can also be treated as a grid of rectangular prisms (or voxels), with each hexahedral prism defined by 8 adjacent points. Isosurfacing can be a time consuming operation when applied to large data sets because it is necessary to threshold and analyze every voxel in the grid. Additionally, a single data set may yield tens of millions of triangles of isosurface geometry. As such we have constructed an FPGA computational kernel that performs isosurfacing in a manner that is similar to the well-known Marching Cubes[24] algorithm.

Marching Cubes steps through every 8-point cube of data in the input volume and analyzes each cube individually to determine if and where the isosurface intersects the cube. An intersection occurs when a cube's points are neither all above nor all below the threshold value. When an intersection occurs, the algorithm locates the points on the cube's edges where the crossings take place and then generates a series of triangles within the cube that best approximate the isosurface within the cube. The opportunities and challenges in adapting this algorithm include the following:

Opportunities:

- **Many Parallel Operations:** The algorithm performs the same operation on a large number of independent data cubes. Therefore, there is a large amount of parallelism that can be exploited in the algorithm.
- **Minimal Floating Point:** The majority of the work performed in the algorithm can be executed with integer operations (e.g., thresholding and portions of the triangle vertex generation). This trait enables us to consider using a large number of slim processing elements in parallel.
- **Two Sections:** The algorithm can be divided into a front-end section that performs thresholding and a back-end section that generates triangles. By placing sufficient buffering between the two halves, it is possible for the (faster) front-end section to be decoupled from the (slower) back-end section.

Challenges:

- **Input Data Structure:** Before any hardware can be designed, it is necessary to determine the shape of the input data that will be passed to the FPGA. The XD1's QDR2 memory is sufficient for housing a full, three-dimensional data set for moderately-sized applications. However, doing so may not be beneficial, since each data point is accessed at most 8 different times by the algorithm. A more attractive approach is to break the volume into smaller, more manageable slices.
- **Large State:** Each cube that is processed by the algorithm has a sizable amount of state information that must be retained until the cube's execution is complete (e.g., the cube's 8 data points and XYZ coordinates are used in the final triangle generation phase). The sheer size of this state information can create a strain on FPGA memory resources when a large number of PEs are instantiated. Therefore, it is necessary to consider techniques by which state information is represented in a compact manner.
- **Synchronization:** While having a large number of PEs is beneficial for performance, the FPGA design must provide the proper arbitration mechanisms to allow the PEs to share common resources such as the back-end triangle generation unit.

Our first major decision in adapting the isosurfacing algorithm to hardware involved selecting the manner in which input data is processed. In order to provide better flexibility and more possibilities for overlapping computation with communication, we decided to have the FPGA process an MxNx2 sheet of data at a time as opposed to working on the entire cube of data. This data set structure is beneficial for multiple reasons. First, it allows larger dimension problems to be processed because the entire data set does not need to fit in memory at the same time. Second, the host can load the next sheet of data into memory while the FPGA is processing the previous sheet. Finally, the control logic for the design is simplified since the FPGA has one less dimension to traverse while stepping through the data set. As illustrated in Figure 5.14, the FPGA breaks the MxNx2 sheet into K PxNx2, where P is the number of PEs available in the array. N is currently set at a maximum of 1024 to simplify buffering requirements, but could easily be increased if needed.

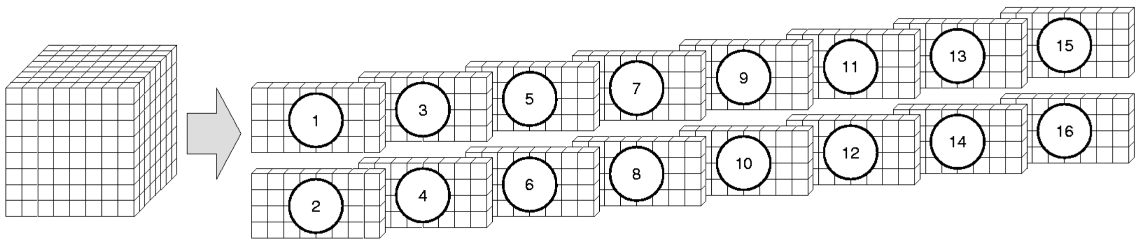


Figure 5.14. Rather than processing an entire cube at a time, the isosurfacing accelerator processes slices of data.

An isosurfacing hardware accelerator was constructed for the XD1's FPGAs. The hardware design utilizes an array of PEs to perform the front-end threshold operations and a back-end triangle generation unit to produce isosurface geometry. A high-level representation of the architecture is illustrated in Figure 5.15.

The number of PEs in the front-end portion of the array is equivalent to the width of the memory interface divided by the width of a data sample. Based on the observation that most scientific instruments that capture volumetric data generate low-precision samples, we optimized the design to utilize 16-bit integer data values. This assumption enables 16 data values to be fetched from the XD1's 256-bit wide QDR2 memory each clock cycle. Adjacent samples are striped across the QDR2 memory banks in order to allow data for 15 cubes to be fetched each clock cycle. After the first column of cubes for a slice is fetched, a new column of cube data is ready for processing every other clock cycle. In order for two adjacent slices to be processed back-to-back (e.g., slices 1 and 2 in Figure 5.14), the last row of data is buffered within the FPGA. This extra row of data enables 16 cube rows to be processed at a time after the first slice is completed. Thus, the front-end unit employs a total of 16 PEs.

A buffering system is utilized between the front and back ends of the design in order to allow the PE units to operate independent of the back end. Once a PE locates a cube that intersects the isosurface, it stores information about the cube in a local FIFO. In addition to

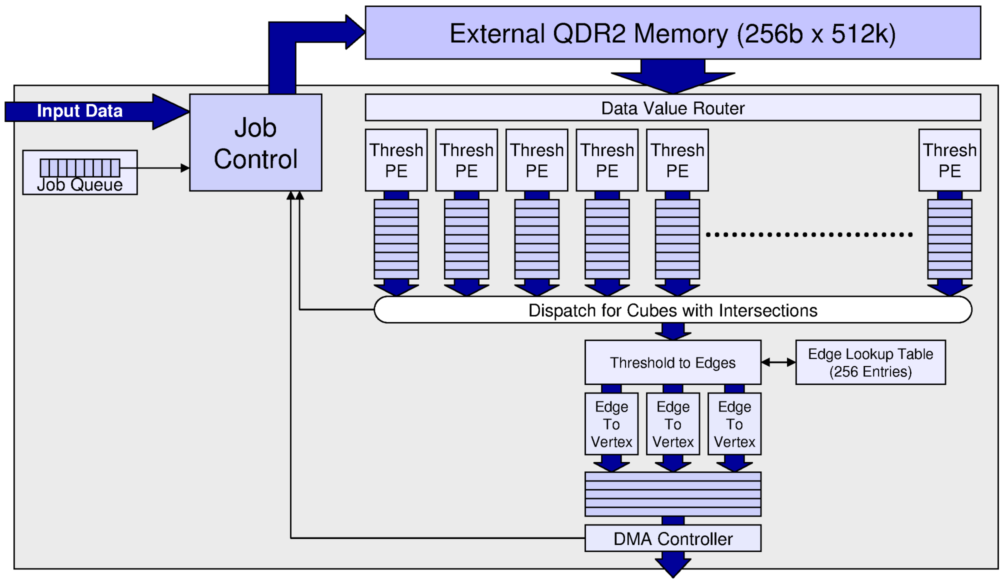


Figure 5.15. Architecture of the isosurfacing accelerator.

being 1024 entries deep, each FIFO is wide enough to store all of a cube's state information (over 100-bits) in a single clock cycle. This bandwidth allows back-pressure control in the front-end to be managed in a simple manner: once any FIFO in the array nears capacity, an alert is issued to the job control unit to stall the issuing of new cubes until the FIFO has enough space to house additional results. Work is issued to the back-end unit through a dispatch unit. This unit polls the PE FIFOs and issues work in a prioritized, round-robin manner. Due to the large number of PEs in the front end, the dispatch unit utilizes a double-buffering technique that reduces the latency in locating new work to issue to the back end.

The back-end section of the architecture transforms state information about a cube into triangle geometry that approximates the isosurface within the cube. In order to accomplish this task, a threshold-to-edges unit utilizes a 256-entry lookup table that specifies the number and orientation of the triangles that should be generated to represent the isosurface for a particular cube's conditions. Based on the lookup table, the threshold-to-edges unit can issue up to five different triangles as output. Three edge-to-vertex units are utilized in parallel to generate vertex co-ordinates, one triangle per clock. This operation requires multiple integer-to-floating-point conversion operations and two floating-point operations per vertex. The end results are stored in a FIFO and then transferred to host memory using a customized DMA engine.

A number of experiments were conducted on the Cray XD1 to observe the performance characteristics of the isosurfacing (ISO) hardware. In our first experiment, we supplied a volume of random input data to the system and compared the performance to an application that performed the work entirely in software. In these tests we consistently observed that the hardware accelerator provided a 4x speedup over software for data dimensions up to 512x512x512. However, it should be noted that random input data is effectively the worst-case scenario for isosurface applications, because the random data does not contain smooth, continuous surfaces and therefore a substantial amount of geometry will be generated. In contrast, we also conducted experiments with null data sets that yielded no isosurface geometry. For the FPGA accelerator, the null data set is the best case because the front-end hardware can rapidly parse through the data without having to be delayed by the back-end triangle generation unit. In fact, we observed performance gains of up to 80x over software in these experiments. Performance for real data sets naturally lies somewhere in between. Our next step in this work is to focus on increasing the clock speed of the design and examine the performance of the system with scientific data sets.

Application Summary

The characteristics for each of the hardware accelerator designs is summarized in Table 5.1.

Table 5.1. Characteristics of the four XD1 accelerator designs

Design	kNN	STIFF	RTI	ISO
Speedup	8x	4x	12x	4x-80x
PEs	8	5	1	16
Clock Rate	164 MHz	190 MHz	179 MHz	140 MHz
External Memory Purpose	Training Vectors	-	Triangle Data	Input Data
FP ADD Units	16	12	24	3
FP MUL Units	8	34	26	0
FP DIV Units	0	1	3	3
Total FP Units	24	47	53	6
Slices (% of V2P50)	19,366 (82%)	17,331 (73%)	20,659 (87%)	18,329 (77%)
RAMB16s (% of V2P50)	40 (17%)	76 (32%)	92 (39%)	199 (85%)
MULT18x18s (% of V2P50)	32 (13%)	148 (63%)	104 (14%)	33 (14%)

XD1 Observations

Our work with the XD1 has provided us with a substantial amount of insight into the low-level details of RC in current-generation systems. In regards to the XD1, we make the following observations:

- **Overall Performance Wins:** Even though our XD1 system was only equipped with mid-sized FPGAs, we were still able to achieve a performance gain in all four applications. While larger capacity FPGAs make it easier to exploit parallelism in an algorithm, it is encouraging to see that speedups are possible with smaller (and lower costing) FPGAs.

- **Reuse vs. Deep Pipelines:** While the STIFF and RTI designs utilize roughly the same number of floating-point units, the RTI design resulted in more of a performance win, even though it operates at a lower clock rate. As this case implies, performance is based on more than just the number floating-point units and the clock speed of a design. The STIFF design is an underachiever because it reuses floating-point units to perform different computations and individual units are not utilized 100% of the time. In contrast, the floating-point units in the RTI design are arranged in a deep pipeline that results in extremely high utilization once data begins to flow. In addition to performance advantages, deep pipelines are more straightforward to implement and require less control logic.

- **Simple Data Transfer APIs:** In our initial work, we cautiously constructed FIFO queues for managing data transfers into and out of the FPGA. While queuing is necessary for controlling the flow of data through the accelerator, we later moved to simpler implementations that required all buffer reservations for a job be completed before the job could be issued to the FPGA. This optimization simplifies the amount of bookkeeping that the FPGA must perform and reduces the amount of state information that must be shared between the CPU and FPGA.

- **Host-CPU Bandwidth Limits Performance:** The XD1's architecture provides substantial bandwidth between CPU and FPGA resources[4]. However, as we observed in the ISO design it is still possible to saturate this connection and incur performance penalties. Designers must anticipate these bottlenecks and partition their algorithms into software and hardware components accordingly.

- **Exploiting On-Card Memory Bandwidth:** The kNN and ISO designs both took advantage of the fact that the XD1 provides wide, on-card memory for FPGA applications. Because this memory can provide more bandwidth than the host CPU can obtain from main memory, it is possible for computational kernels to exhibit significant wins when on-card memory is used wisely. The challenge in this work is finding an application with the right memory access patterns, and implementing hardware that is capable of issuing memory transactions every clock cycle.

[4]In fact, the only commercial system we know of with higher bandwidth is the SRC-7.

- **Lengthy Development Time:** Each of these designs required between three weeks (STIFF) and three months (RTI) to develop and debug. While we built in-house tools to automate portions of the work[5], the entire development process is lengthy and complex. While we expected hardware development to be time consuming, we found that an unusually large portion of our development time involved writing the interface software for the host applications and adjusting it to achieve acceptable performance. In hindsight, we should have anticipated this overhead, as debugging the software interface is where a design first comes to life and all unexpected problems first start to materialize.

- **XD1 as a Research Vehicle:** In polite terms, the XD1 was a fascinating research vehicle, but alas, a far cry from a production-quality ride. As we expected, the XD1 gave us a flexible platform for low-level experiments that would simply not be possible in other systems. That said, our XD1 exhibited multiple hardware failures (e.g., multiple cooked hard drives and occasional overheating in the FPGAs) and many software problems (e.g., lost home directories). If it were not for the dedication of the original XD1 development team (OctigaBay) we would have been lost in Cray's antiquated, coin-operated bureaucracy.

Earlier this year Cray announced that it was discontinuing the XD1 product line. While some of the authors of this report were unmoved by this news, the XD1's demise is another example of a computer industry truism: well-thought out architectures do not always lead to profitable business ventures. As such, we feel it is important to reiterate the importance of supporting commercial efforts that are based on open standards and provide open source code.

[5]For example, we built a tool to analyze an algorithm's data flow graph (DFG) and generate VHDL hardware for the pipeline of operations.

Chapter 6

System Simulation

One of the challenges with FPGA based application acceleration is the issue of system architecture. Two things are clear: FPGAs will be getting much faster over the next few years, and FPGAs will still have to be complemented by a conventional microprocessor to support general purpose applications. The coupling between the FPGA and the conventional microprocessor will be a key issue for the success of these systems. To explore this issue, we considered an approach proposed by many building accelerator based systems: intercepting common library calls (e.g. BLAS and FFT calls) and executing them on the accelerator. We focused on the DGEMM and FFT calls in the way that the scientific applications in use at Sandia currently use those calls: a streaming series of small calls.

Motivation and Methodology

There has been a recent surge in work on new system architectures supporting FPGAs (and accelerators in general). Each system has different implications for the aggregate bandwidth and latency of the connection to the processor (Table 6.1[1]). While a large body of work has explored the *potential* of these systems for scientific computing, little work has discussed the context of real applications. Research tends to explore the power of FPGAs to address large dense matrix algorithms (DGEMMs) and large FFTs, but such work seldom comments on any applications that actually use such operations (mostly because such applications are rare). More importantly, few analyses have looked at the salient properties of the architectures to determine their relative importance.

Motivating Applications

Generally speaking, scientific applications do not leverage large dense matrix operations. Indeed, we know of no supercomputer applications at Sandia that leverage them. Small dense matrix operations are a different story. For example, a large number of small, double precision, complex 1D Fast Fourier Transforms (FFTs) is used in parallel molecular

[1]HyperTransport (HT) can achieve over twice the listed rate, but it is constrained by FPGA I/O capabilities.

Table 6.1. Typical system parameters

System Type	Latency	Bandwidth
PCI-X	1000 ns	1000 MB/s
PCI-Express 8X	800 ns	3.6 GB/s
HyperTransport (HT)	250 ns	3.2 GB/s
Memory	250 ns	3.6 GB/s

dynamics codes (e.g., LAMMPS[29, 30]) as part of a distributed 3D FFT. The typical approach to a distributed 3D FFT is to perform three iterations of N^2 FFTs of size N with data transpositions between the iterations. The typical size of a call is on the order of 128 to 256 and N^2 consecutive, independent calls to the function are made. Similarly, forecasting codes can use thousands of small (size 256 to 1500), consecutive, independent 1D FFT calls[43]. FFTs are a significant, although not dominant, contributor to overall run-time for these applications.

Another example is the dense matrix multiply operation (DGEMM). While none of Sandia's applications use DGEMM operations for large operations (dimensions of hundreds or more), many of them do small, dense, double precisions matrix multiplies. For example, MPQC[18] is a quantum chemistry code that uses thousands of small, independent matrix multiplies ranging from 15×15 to 50×50. Similar patterns occur in sparse matrix multiplies that are used in Mondo SCF (another quantum chemistry code). Finally, modern solver techniques are moving to locally dense, globally sparse matrices with multiple right-hand sides. This also leads to a large number of independent DGEMM operations that tend to be approximately 16×16. It is anticipated that the DGEMM calls will become as much as 90% of the execution time as other aspects of the codes are tuned.

Methodology

This effort used two hardware platforms and a system level simulator to run the same benchmarks. The first hardware platform was the Cray XD1, which was used as both the FPGA platform and as a processor platform to measure the performance of the Opteron. The second hardware platform was a Pentium-4 Xeon workstation that was used as an example of the *best available* processor performance, since the Pentium-4 is known to have a higher peak floating-point rate on dense matrix operations than the Opteron.

Cray XD1

Figure 6.1 depicts an XD1 compute blade containing two AMD Opteron processors. One CPU connects directly to a network interface (NI) chip using a HyperTransport (HT) link. This NI uses a Xilinx Virtex2Pro FPGA that limits the CPU to network interface link to

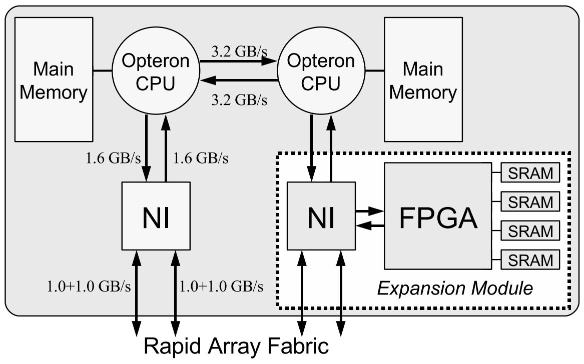


Figure 6.1. The XD1 architecture

1.6 GB/s per direction (1.4 GB/s after overhead). The user-programmable FPGA is on an expansion board connected to an HT interface on the second processor. As Figure 6.1 illustrates, the user FPGA is connected to a second NI through a simplified version of HT. This interface enables the FPGA to read and write the hosts memory, as well as respond to memory requests issued by the host processor. The XD1 utilized for this work is populated with Virtex2Pro50-7 expansion boards and Opteron 248 processors, which are clocked at 2.2GHz and have a peak of 4.4 GFLOPs.

The FPGA design flow used Xilinx's ISE 6.3.03 tool chain including the Xilinx Synthesis Tool (XST) for VHDL synthesis. The XD1 system runs the 1.1 release of Cray's system software with a modified Linux 2.4.21 kernel. The 1.1 release of the XD1 system software can only pin one 2MB block of contiguous host memory for sharing data with the FPGA. This typically requires a data copy, but the more recent 1.3 release of XD1 system software uses the Graphics Address Resolution Table (GART) to address up to 1 GB of host memory from the FPGA and eliminates the need for a copy. Our tests emulate this behavior by not performing the final copy in the 1.1 release.

Pentium-4 Xeon Workstation

A Pentium-4 Xeon workstation was used as one of the processor baselines for comparison. The workstation has two 3.2 GHz Pentium-4 Xeon processors with EM64T technology. It is equipped with 8 GB of RAM and is running the RedHat Enterprise 4 WS Linux distribution. The compiler used was GCC 3.4.5, but the compiler only compiled the outer loop. Matrix operations were measured using FFTW version 3.1[8] and the Intel Math Kernel Library (MKL) version 8.1. FFTW was faster for the FFT portion of the test, and so it was reported in the results section.

```
choose_operation_size();
buffer=allocate_buffer();
begin_timer();
for (i = 0; i < N; i++)
    do_op(buffer);
end_timer();
```

(a)

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
begin_timer();
for (i = 0; i < N; i++)
    do_op(buffer[i]);
end_timer();
```

(b)

Figure 6.2. The traditional (a) and modified (b) benchmarks

Structural Simulation Toolkit

To explore configurations beyond the XD1, we used the Structural Simulation Toolkit (SST). The Structural Simulation Toolkit is built around Enkidu, a hybrid simulation framework that optimizes for the common case in architectural simulation by providing low-overhead synchronous time-stepping to handle most functionality. For less frequent communication between components, an asynchronous event mechanism is provided. SST integrates the SimpleScalar (v3.0) toolkit's sim-out-order processor model[3] to model conventional processors. For these experiments, an execution-based front end supporting PowerPC Mach-O binaries was used.

Benchmarks

Two benchmarks were used to test these concepts and provide a comparison point against traditional microprocessors. These simple benchmarks were designed to capture the way applications work; thus, they differ slightly from a traditional benchmark. A comparison is shown in Figure 6.2 — the difference is subtle, but critical. Where a typical benchmark does a large number of iterations over a single buffer, real applications tend to do a large number of iterations over *different* buffers. Thus, the benchmarks reflect that.

Approach

Virtually every major HPC system shipped today requires standard libraries to be available for the Basic Linear Algebra Subroutines (BLAS) and FFT. Some vendors have proposed using hardware accelerators to intercept these calls and, thus, provide improved performance. The problem, however, is in the traditional semantics of a blocking subroutine call. Contrast the examples in Figure 6.3. The blocking calls are perfectly suitable for either execution on a host microprocessor or for performing large routines on a compute accel-

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
for (i = 0; i < N; i++)
    do_op(buffer[i]);
```

(a)

```
choose_operation_size();
for (i = 0; i < N; i++)
    buffer[i] = allocate_buffer();
for (i = 0; i < N; i++)
    start_op(buffer[i], request[i]);
wait_all(request, status);
```

(b)

Figure 6.3. **(a)**Blocking vs. **(b)**Non-blocking approaches

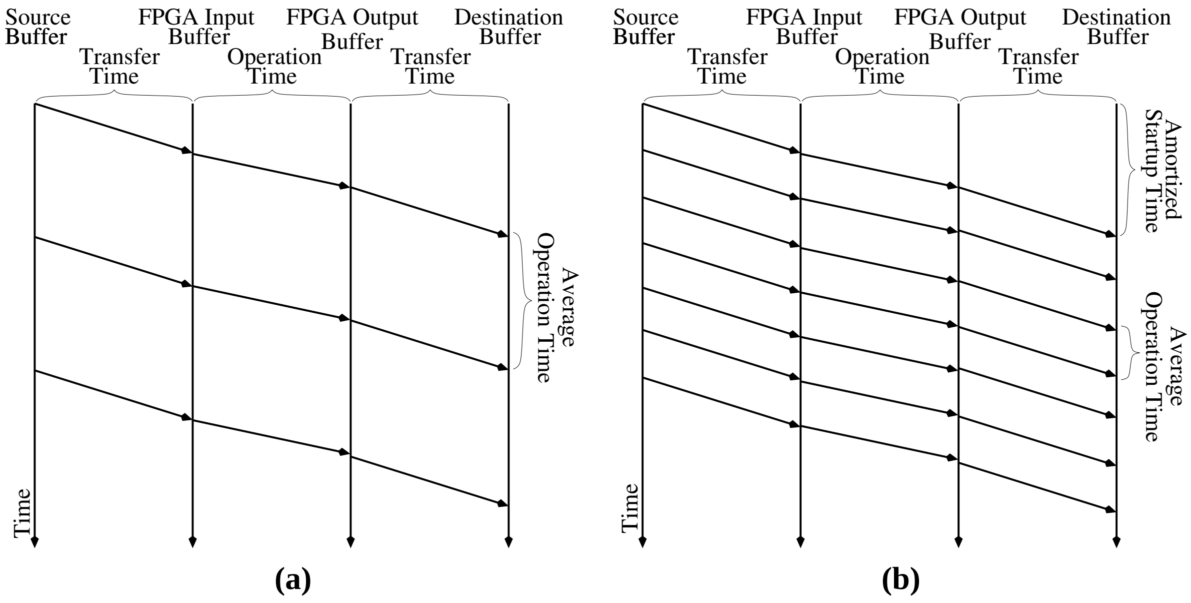


Figure 6.4. Timeline for **(a)** blocking and **(b)** non-blocking operations

erator; however, when there is a large number of small operations to do, the nonblocking calls expose more parallelism to the system and enable the system to pipeline these operations. The net effects can be seen in the timeline in Figure 6.4, where the nonblocking calls can leverage the double buffering on the accelerator to overlap communication between the host processor and accelerator with computation that is occurring on the accelerator. While this is conceptually straightforward, this paper aims to quantify the impact in the context of a specific technology to motivate the development of appropriate APIs.

To target the way Sandia's applications use DGEMM and FFT calls, we focus on implementations that can accelerate large numbers of small calls. Thus, rather than one large unit in the FPGA, we have several smaller units that can each handle the processing for a call. These are integrated with the host using double buffering techniques so that data can be transferred between the host and FPGA while the computation is occurring. Figure 6.5

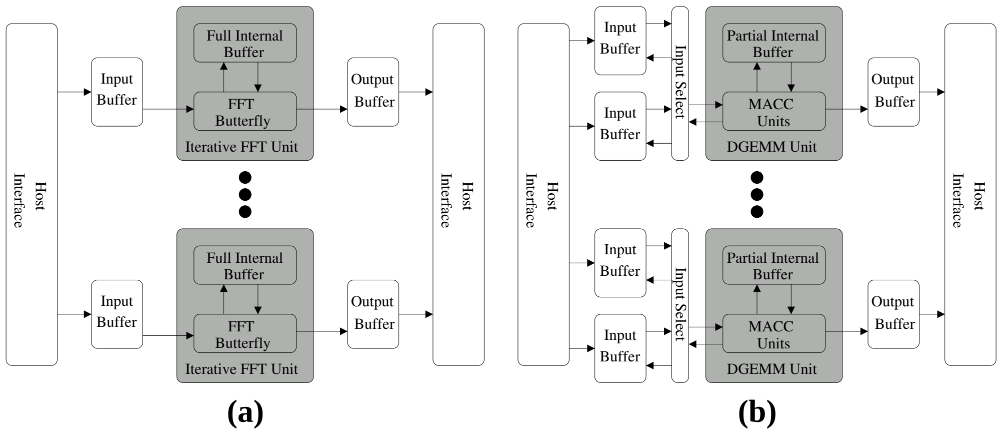


Figure 6.5. Buffer approaches for **(a)** FFT and **(b)** DGEMM.

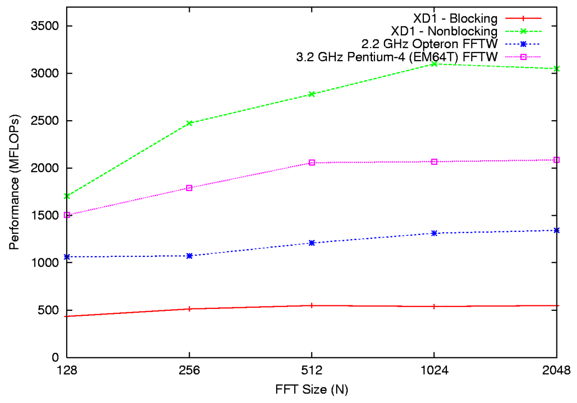
illustrates this concept for the FFT and DGEMM operations.

The FFT and DGEMM operations in Figure 6.5 take slightly different approaches. While the FFT needs *both* ports of a dual-ported RAM during the core operation, the DGEMM operation does not. Therefore, the FFT uses full sized input, output, and internal buffers. This is sufficient to provide double buffering for the FFT. The DGEMM operation, on the other hand, allows the system to manage the double buffering. An input selection mechanism makes this transparent to the unit. In practice, this is implemented by splitting the top and bottom of a single dual-ported RAM bank into independent buffers and selecting based on the high order address bit.

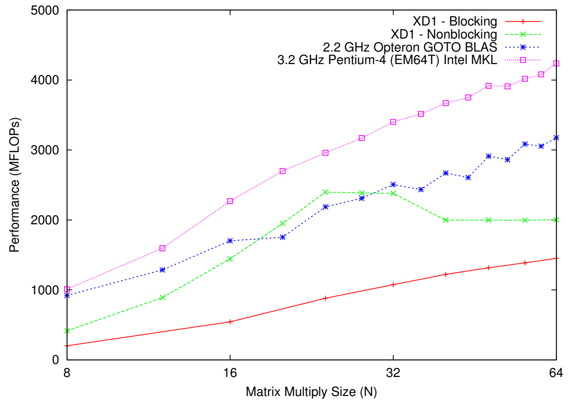
Measurements from the XD1

To validate our analysis and simulations and to provide a concrete comparison point between microprocessors and FPGAs, we took measurements from the Cray XD1 and from commodity microprocessors. Figure 6.6(a) compares the FFT performance of the FPGA on the Cray XD1 to the Opteron on the same platform and a recent Pentium-4 Xeon processor. The first observation about the graph is that it is very different from the "standard" benchmark graphs for the processors used, because our benchmarks performed 1000 operations over 1000 different buffers where traditional benchmarks perform 1000 operations over the same buffer.

The next important note is that the FPGA on the Cray XD1 can achieve a 60% to 125% performance improvement, depending on problem size, if a nonblocking API is used. Using a blocking API stands in stark contrast with 60% *lower* performance than the microprocessor. Results from the Pentium-4 Xeon indicate that this 2.5 year old FPGA that is only half

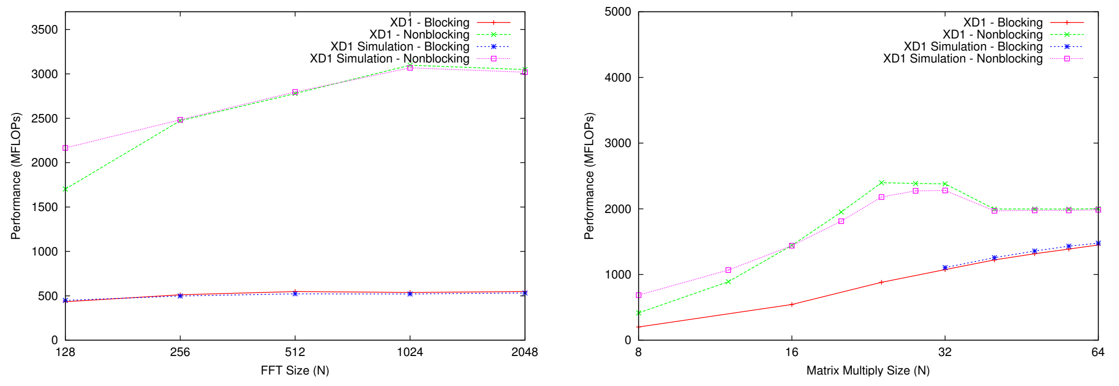


(a)



(b)

Figure 6.6. Comparison of FPGA performance to processor performance for **(a)** FFT and **(b)** DGEMM as measured on the XD1



(a) **(b)**

Figure 6.7. Validation of the simulation models for **(a)** FFT and **(b)** DGEMM

as big as the largest FPGA from that era can outperform one of the fastest current single core processors — if, and only if, it uses a nonblocking interface.

Figure 6.6(b) presents results from DGEMM and contrasts with the FFT results. Modern microprocessors are highly optimized to perform operations like DGEMM; thus, the FPGA loses by as much as a factor of 2, even when it uses a nonblocking interface. Given that it is a relatively old, relatively small FPGA, this is not surprising.

Simulation Results

Like microprocessors, FPGAs have reaped many benefits from Moore's Law. In fact, recent FPGA performance gains have outstripped the performance gains of microprocessors To explore the near-term potential of FPGAs, we leveraged a hybrid discrete-event/cycle-driven simulator. The initial simulations were used to validate the simulator. Figure 6.7 indicates that the simulator captures most of the salient points of the system. For the FFT, the only point that is not in almost perfect agreement (within 1%) between the simulator and the XD1 implementation is the 128 point non-blocking FFT. The simulation is more representative of what can be achieved, but there is approximately 500 ns of overhead in the XD1 implementation that is associated with buffer management that we are trying to eliminate.

The DGEMM results are slightly less tightly correlated. For many cases, the simulation results are within 1% of the measured values. From an *N* of 20 to 32, the results are within 5%. Measurements indicate that at least a 1% difference can be attributed to the inaccuracies in the actual clock rate on the XD1. Below an *N* of 16, the operations use smaller transfers than the 128 point FFT; thus, the differences occur for the same reason

— unexplained overhead in the interaction with the host that is being debugged. Blocking simulations were only run over the range of N from 32 to 64, since those are the only points where the blocking implementation fully utilizes the resources. For these runs, the simulations are within 3% of the measured values. Overall, we believe the simulations provide sufficient predictive capabilities when the predicted differences are large.

The second set of simulations focused on the bandwidth and latency requirements for FPGAs now and as FPGAs increase in performance. These simulations used nonblocking operations and swept latencies from a minimum of 50 ns to a maximum of 1000 ns. With nonblocking operations, latency had no impact on the performance and those results are omitted for brevity. FPGA sizes are modeled based on the Virtex2Pro50-7 on the Cray XD1, the Virtex2Pro100-6 available on SRC systems, one Moore's Law doubling (40 MACC units, 250 MHz, roughly a Virtex4-FX140) and two Moore's Law doublings (80 MACC units, 380 MHz)[2]. Results presented in Figure 6.8 assume a latency between the processor and FPGA of 250 ns.

For currently available systems (Figures 6.8 (a) and (c)), the choice of device matters. Larger devices deliver more performance with bandwidths that are readily achievable. It is also clear that PCI-X levels of bandwidth (500 MB/s/direction or 1000 MB/s total) are not sufficient for the domains of interest discussed earlier, and, generally speaking, modern FPGA systems need more bandwidth. Even the bandwidth of HT (1400 MB/s/direction sustained) is not quite sufficient.

A more interesting story arises from Figures 6.8 (b) and (d). All but one pair of lines completely overlaps with the generation after it all the way to 5 GB/s/direction. Two parts with a $3\times$ difference in performance are both bandwidth constrained to the same performance in the domain of interest. A more aggressive design point could take the bandwidth to the FPGA up to 10 GB/s/direction or even 20 GB/s/direction as technologies like HT-3 come online. While the graphs show the potential for strikingly high levels of performance, the I/O connections to FPGAs will have to improve dramatically to leverage any of those potential gains.

While the bandwidth requirements seem extreme, there are bright points. Foremost, it is unlikely that the largest FPGAs (represented by the "two doublings" line) will ever be cost-effective for HPC applications. With the performance of FPGAs growing faster than that of microprocessors, this would indicate that a smaller, cheaper FPGA could deliver all of the performance that the bandwidth will support. Also, with growth in computing power comes growth in the amount and types of science that can be done. This tends to increase the size of operations such that 16×16 DGEMMs and 128 point FFTs are likely to become 32×32 DGEMMs and 512 point FFTs in another generation. These trends will somewhat reduce the bandwidth needed to achieve higher levels of FPGA performance.

The final set of simulations explored the impact of a traditional blocking API on FPGA

[2]These numbers assume that a generation gives $2\times$ the transistor count, but somewhat under $2\times$ the clock rate. Clock rate growth is difficult to forecast for floating-point on FPGAs as it is virtually independent of technology in the near term and depends on FPGA features introduced.

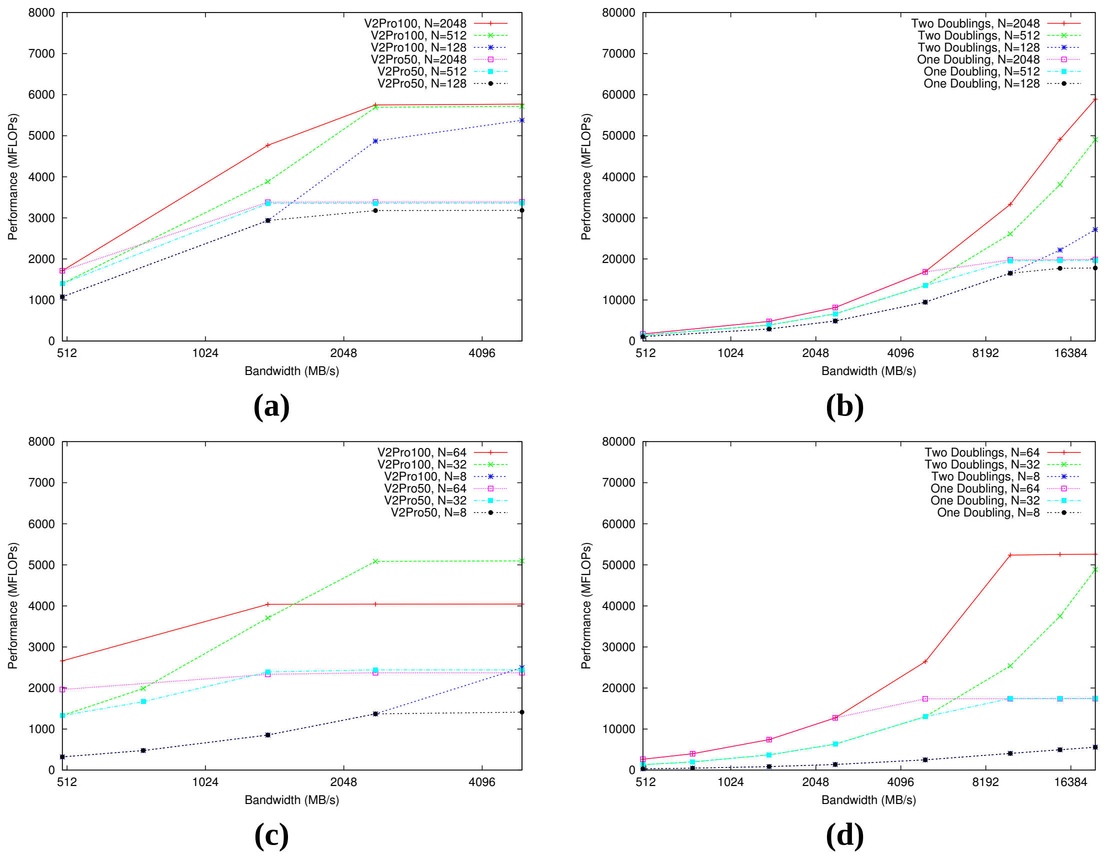


Figure 6.8. Simulated impact of bandwidths as FPGA capabilities scale for FFT ((**a**) and (**b**)) and DGEMM ((**c**) and (**d**))

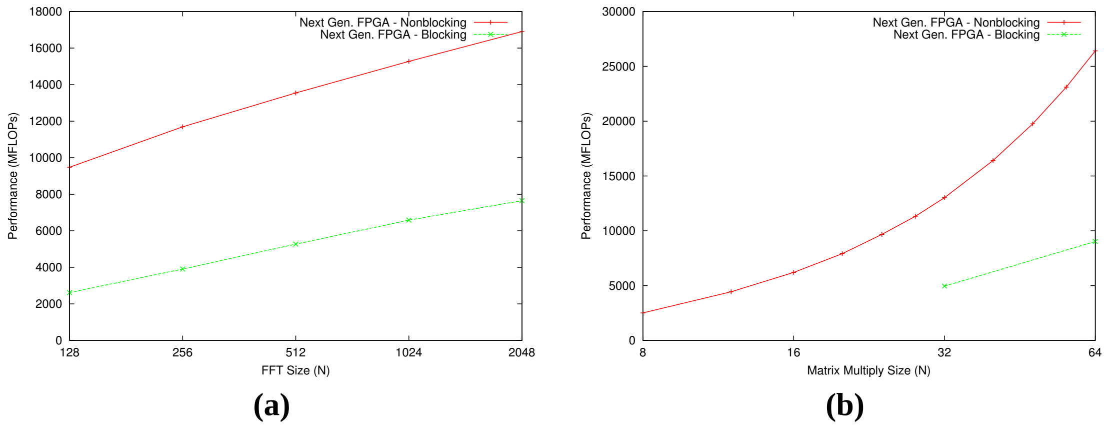


Figure 6.9. Simulated impact of blocking operations for **(a)** FFT and **(b)** DGEMM

performance at two generations beyond the Virtex2Pro100. The latency between the processor and the FPGA was set to an optimistic 50 ns (to offer every advantage to the blocking approach) and bandwidth was set to 5 GB/s/direction to match what should be achievable soon with HT or PCI-Express. Assuming that at least 1000 independent operations are required yields the graphs in Figure 6.9. The blocking calls are $2\times$ to $3\times$ slower than their nonblocking counterparts. More importantly, the blocking graph for the matrix multiply *starts* at 32×32 because this is the first point at which a matrix multiply could fully utilize the device.

Summary

It is important to recognize the mismatch between the way FPGA researchers assess FPGA capabilities and the way applications would use the devices. Accelerator proponents like to claim that they can accelerate standard libraries, but many applications do not call these libraries in a way that can be exploited; thus, a new, non-blocking API is needed to expose the parallelism to the hardware. Indeed, the issue of API impacts performance as much or more than the issue of architectures with as much as a $3\times$ loss in performance if the wrong API is used.

That said, architecture is critically important. The performance that the Xilinx Virtex4-FX140 should deliever will far outstrip the bandwidth that is readily available to it. With 5 GB/s/direction of bandwidth, this part (which should be available soon) would greatly outstrip the capabilities of a microprocessor. And, modern interfaces can deliver almost that level of bandwidth, but FPGA I/Os often limit the performance of those interfaces (e.g. HT). This challenge multiplies with the next generation, where those parts cannot be distinguished from current parts without delivering drastically more bandwidth.

As a final note, while these results were obtained with an FPGA focus, many of them are generally applicable to accelerator technologies. Accelerator proponents typically offer to accelerate DGEMM and FFT operations by attaching an accelerator to a commodity microprocessor and using standard BLAS calls. These results indicate that this may work in the near term for large operations, but in the longer term, the traditional blocking BLAS calls will hide the parallelism that is required. Furthermore, for many of the DGEMM and FFT operations used by scientific applications, the API is already a major barrier and the system architecture (particularly bandwidth) will be an enormous barrier in the near future. This is not meant to say that accelerators are infeasible, only that traditional library calls will not be a productive way to use them.

Chapter 7

Challenges to Deploying FPGAs

Even if FPGAs prove to have dramatic performance advantages, they still face many challenges to deployment. These range from algorithms to run on the new platform to the cost and reliability of the platform itself.

Algorithms

Much as new algorithms and parallelization strategies were needed with the rise of the massively parallel processing (MPP) paradigm, new algorithms will be needed to leverage the unique capabilities of FPGAs while interacting with microprocessors as peers. This means partitioning the application differently than any system in existence today. As an example, consider the typical node envisioned by many, where each node in an MPP consists of a microprocessor and an FPGA. The FPGA could exist in a completely subordinate state, where the microprocessor makes calls that are then executed on the FPGA. The microprocessor *could* block while these calls execute. This type of model, however, is what led Amdahl to make his initial (and pessimistic) observations about the potentials of parallel computing. A far better approach would be to refactor the algorithms to promote the FPGA to the status of a full peer that concurrently ran a separate part of the algorithm.

Programmability

One of the weakest points for FPGAs has always been the challenges of programming the devices. The ultimate problem is that traditional sequential languages (C, C++, Fortran) do not express the kinds of parallelism that are inherent in a hardware device. Thus, legacy code does not, and cannot, readily compile to modern FPGAs and deliver high performance.

At the other end of the spectrum from traditional sequential languages are hardware description languages (HDLs). An HDL does an exemplary job of expressing hardware level parallelism and the associated control flow. The only problem is that the vast majority of applications developers do not know and do not want to know how to program in an HDL.

Furthermore, development in an HDL is time consuming and tedious and would greatly extend application development time.

A compromise position adds extensions to sequential languages (typically to C) that enable the programmer to express parallelism that he otherwise could not. Unfortunately, the current state of the art in this domain still sacrifices significant performance (at least $2\times$) while still forcing the developer to significantly change his code (e.g. pointers are not typically supported). Thus, even in the best case, where these tools mature to have comparable performance to an HDL, the user is going to need an extremely compelling reason to rewrite the application in the new language.

Portability

When application developers create a new application, they expect it to run everywhere. One of the fundamental enablers of the widespread adoption of MPPs was the rise of the MPI standard. With MPI, a single program would run on any MPP (and, now, commodity clusters) with minimal porting effort. FPGAs are a long way from this goal. Most FPGA programming environments are provided by a hardware vendor (at significant cost) and are completely bound to a single platform. Some of the higher level languages can be retargeted to a small selection of different platforms; however, the architecture of those platforms currently vary so dramatically that it can be very challenging to obtain performance portability.

Architecture

Most systems still treat the FPGA as a subordinate accelerator with a *subroutine* mentality. This leads to a mentality where either the FPGA or the hosting processor is executing a single serial stream. The FPGA needs to become a full peer. Furthermore, as our results have indicated, that FPGA needs to have remarkable levels of bandwidth to the host processor to obtain data to process. Fortunately, as algorithms improve, the demand for bandwidth could be reduced.

Cost

FPGA based accelerators are phenomenally expensive by HPC standards. Given an HPC node that costs $5,000, adding a $10,000 FPGA can be quite prohibitive. This seriously impacts the performance/cost advantage of the FPGA. In the future, if the performance advantage of FPGAs becomes sufficiently compelling, it will be possible to use a lower cost FPGA instead. This will be absolutely critical for the success of FPGAs in HPC.

Reliability

An FPGA system that delivers an order of magnitude improvement over the best current MPP will have many more parts than the MPP. Many of those parts will be FPGA devices that are sensitive to soft-errors and are hard to detect errors in (see [31]). The corresponding decrease in system reliability will be unacceptable for major HPC procurements.

Chapter 8

Conclusions

FPGAs have remarkable potential for impacting the future of high performance computing. Projections suggest that FPGAs could yield an order of magnitude performance improvement over microprocessors. More importantly, many algorithms demonstrate a dramatically higher *sustained percentage* of peak performance than microprocessors. Together, the potential for improvement can approach two orders of magnitude.

Unfortunately, FPGAs are not yet ready for production usage in HPC systems and face major hurdles to becoming ready for production use. Issues of programming and portability are currently major barriers for applications developers, but there are organizations such as OpenFPGA (www.openfpga.org) that are trying to address those barriers. Studies at other national labs[31] have indicated that FPGAs are sufficiently succeptible to terrestrial radiation to prevent their use in a large HPC system; however, device manufacturers are working hard to address this problem. Finally, FPGAs still face a major barrier in terms of cost, but the issue of cost can easily be addressed by using smaller devices if the performance advantage is sufficiently compelling.

The net result is that the future of FPGAs depends heavily on the successes of industry and academia in addressing many key challenges. FPGAs have numerous architectural and market advantages that make them a compelling technology *if* they can be deployed in a reliable system at a reasonable cost.

References

[1] *IA-32 Intel Architecture Optimization: Reference Manual*. Intel Corporation, USA, 2003. Order Number:248966-009.

[2] Pavle Belanovic and Miriam Leeser. A library of parameterized floating-point modules and their use. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2002.

[3] Doug Burger and Todd Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[4] Michael deLorimier and Andr DeHon. Floating point sparse matrix-vector multiply for FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2005.

[5] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in fpga based DSPs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2002.

[6] Yong Dou, S. Vassiliadis, G.K. Kuzmanov, and G.N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2005.

[7] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI*, 2(3):365–367, 1994.

[8] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

[9] Matteo Frigo and Steven G. Johnson. Fft benchmark results, January 2005. From webpage at http://http://www.fftw.org/speed/p4-2.8GHz-new/.

[10] Altaf Abdul Gaar, Wayne Luk, Peter Y.K. Cheung, Nabeel Shirazi, and James Hwang. Automating customisation of floating-point designs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, 2002.

[11] Altaf Abdul Gaar, Oscar Mencer, Wayne Luk, Peter Y.K. Cheung, and Nabeel Shirazi. Floating point bitwidth analysis via automatic differentiation. In *Proceedings of the International Conference on Field Programmable Technology*, Hong Kong, 2002.

[12] Gokul Govindu, Seonil Choi, Viktor K. Prasanna, Vikash Daga, Sridhar Gangadharpalli, and V. Sridhar. A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*, Santa Fe, NM, April 2004.

[13] Gokul Govindu, Ling Zhuo, Seonil Choi, Padma Gundala, and Viktor K Prasanna. Area and power performance analysis of a floating-point based application on FPGAs. In *Proceedings of the Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003)*, September 2003.

[14] Gokul Govindu, Ling Zhuo, Seonil Choi, Padma Gundala, and Viktor K Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop (RAW)*, Santa Fe, NM, April 2004.

[15] K. Scott Hemmert and Keith D. Underwood. An analysis of the double-precision floating-point fft on fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2005.

[16] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting. A CAD suite for high-performance FPGA design. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–24, Napa, CA, April 1999. IEEE Computer Society, IEEE.

[17] IEEE Standards Board. IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std. 754-1985, The Institute of Electrical and Electronics Engineers, New York, 1985.

[18] Curtis Janssen. Personal communications.

[19] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[20] Volodymyr Kindratenko and David Pointer. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2006.

[21] M. P. Leong, M. Y. Yeung, C. K. Yeung, C. W. Fu, P. A. Heng, and P. H. W. Leong. Automatic floating to fixed point translation and its application to post-rendering 3d warping. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 240–248, Napa Valley, CA, April 1999.

[22] Jian Liang, Russel Tessier, and Oskar Mencer. Floating point unit generation and evaluation for fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 185–194, Napa Valley, CA, April 2003.

[23] Walter B. Ligon, Scott P. McMillan, Greg Monn, Fred Stivers, Kevin Schoonover, and Keith D. Underwood. A re-evaluation of the praticality of floating-point on FPGAs.

In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, Napa Valley, CA, April 1998.

[24] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, volume 21, pages 163–169, New York, NY, USA, July 1987. ACM Press.

[25] Loucas Louca, Todd A. Cook, and William H. Johnson. Implementation of IEEE single precision floating point addition and multiplication on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116, 1996.

[26] Zhen Luo and Magaret Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, 49(3):208–218, 2000.

[27] Tomas Mller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *journal of graphics tools*, 2(1):21–28, 1997.

[28] Gerald R. Morris, Victor K. Prassana, and Richard D. Anderson. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2006.

[29] Steven J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal Computation Physics*, 117:1–19, 1995.

[30] Steven J. Plimpton, R. Pollock, and M. Stevens. Particle-mesh ewald and rRESPA for parallel molecular dynamics. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[31] Heather Quinn and Paul Graham. Terrestrial-based radiation: A cautionary tale. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2005.

[32] Ronald Scrofano, Maya Gokhale, Frans Trouw, and Viktor K. Prasanna. A hardware/software approach to molecular dynamics on reconfigurable computers. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2006.

[33] Nabeel Shirazi, Al Walters, and Peter Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 155–162, 1995.

[34] William D. Smith and Austars R. Schnore. Towards and RCC-based accelerator for computational fluid dynamics applications. In *Proceedings of The International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03)*, pages 226–232, 2003.

[35] Winthrop W. Smith and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms*. Wiley-IEEE Press, New York, 1995.

[36] Justin L. Tripp, Henning S. Mortveit, Anders A. Hansson, and Maya Gokhale. Metropolitan road traffic simulation on fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2005.

[37] Keith D. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2004.

[38] Keith D. Underwood and K. Scott Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 2004.

[39] Jeffrey S. Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *Proceedings of the 2002 Conference on Supercomputing*, November 2002.

[40] Xiaojung Wang and Brent E. Nelson. Tradeoffs of designing floating-point division and square root on Virtex FPGAs. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 195–203, Napa Valley, CA, April 2003.

[41] R. Clint Whaley. Automatically tuned linear algebra software (ATLAS), January 2004. From webpage at http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF00/Whaley.pdf.

[42] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[43] D. L. Williamson, J. B. Drake, J. J. Hack, R. Jakob, and P. N. Swarztrauber. A standard test set for numerical approximations to the shallow water equations in spherical geometry. *J. Comput. Phys.*, 102:211–224, 1992.

[44] Ling Zhuo and Viktor K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on fpgas. In *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, April 2004.

[45] Ling Zhuo and Viktor K. Prasanna. Design tradeoffs for blas operations on reconfigurable hardware. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, Oslo, Norway, June 2005.

[46] Ling Zhuo and Viktor K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *Proceedings of the ACM/IEEE SC2005 Conference*, November 2005.

[47] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2005.

DISTRIBUTION:

1	MS 0139	Arthur Hale, 1900
1	MS 0316	Scott Hutchinson, 1437
1	MS 0321	Jennifer Nelson, 1430
1	MS 0370	James Strickland, 1433
1	MS 0376	Ted Blacker, 1421
1	MS 0378	Randy Summers, 1431
1	MS 0384	Arthur Ratzel, 1500
1	MS 0630	Kenneth Washington, 4600
1	MS 0672	Lyndon Pierson, 5616
1	MS 0801	Robert Leland, 4300
1	MS 0822	David Rogers, 1424
1	MS 0970	Anthony Medina, 5700
1	MS 1205	Sidney Gutierrez, 5900
1	MS 1316	Mark D. Rintoul, 1412
1	MS 1316	Steve Plimpton, 1412
1	MS 1318	William Camp, 1410
1	MS 1318	David Womble, 1410
1	MS 1318	Suzanne Rountree, 1415
1	MS 1318	Andrew Salinger, 1416
1	MS 1318	Bruce Hendrickson, 1415
1	MS 1319	K. Scott Hemmert, 1423
1	MS 1319	Keith Underwood, 1423
1	MS 1319	James Ang, 1422
1	MS 1319	Neil Pundit, 1423
1	MS 1320	Scott Collis, 1414
1	MS 1320	Mike Heroux, 1414

1	MS	1322	Sudip Dosanjh, 1420
1	MS	1322	James Tomkins, 1420
1	MS	1322	John Aidun, 1435
1	MS	9051	Philippe Pebay, 8351
1	MS	9151	Leonard Napolitano, 8900
1	MS	9151	Howard Hirano, 8960
1	MS	9152	Craig Ulmer, 8963
1	MS	9152	David C. Thompson, 8963
1	MS	9152	Jerrold Friesen, 8963
1	MS	9158	Curtis Janssen, 8961
1	MS	9158	Robert Armstrong, 8961
1	MS	9158	Matthew Leininger, 8961
1	MS	9158	Helgi Adalsteinsson, 8961
1	MS	9159	Heidi Ammerlahn, 8962
1	MS	9159	Michael Hardwick, 8964
2	MS	9018	Central Technical Files, 8944
2	MS	0899	Technical Library, 4536
1	MS	0123	D. Chavez, LDRD Office, 1011

Sandia National Laboratories