NUMERICAL SOLUTIONS OF DIFFERENTIAL EQUATIONS ON FPGA-ENHANCED COMPUTERS

A Dissertation

by

CHUAN HE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2007

Major Subject: Electrical Engineering

NUMERICAL SOLUTIONS OF DIFFERENTIAL EQUATIONS ON FPGA-ENHANCED COMPUTERS

A Dissertation

by

CHUAN HE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Co-Chairs of Committee,	Mi Lu
	Wei Zhao
Committee Members,	Guan Qin
	Gwan Choi
	Jim Ji
Head of Department,	Costas N. Georghiades

May 2007

Major Subject: Electrical Engineering

ABSTRACT

Numerical Solutions of Differential Equations on
FPGA-Enhanced Computers. (May 2007)
Chuan He, B.S., Shandong University;
M.S., Beijing University of Aeronautics and Astronautics
Co-Chairs of Advisory Committee: Dr. Mi Lu
Dr. Wei Zhao

Conventionally, to speed up scientific or engineering (S&E) computation programs on general-purpose computers, one may elect to use faster CPUs, more memory, systems with more efficient (though complicated) architecture, better software compilers, or even coding with assembly languages. With the emergence of Field Programmable Gate Array (FPGA) based Reconfigurable Computing (RC) technology, numerical scientists and engineers now have another option using FPGA devices as core components to address their computational problems. The hardware-programmable, low-cost, but powerful "FPGA-enhanced computer" has now become an attractive approach for many S&E applications.

A new computer architecture model for FPGA-enhanced computer systems and its detailed hardware implementation are proposed for accelerating the solutions of computationally demanding and data intensive numerical PDE problems. New FPGA-optimized algorithms/methods for rapid executions of representative numerical methods such as Finite Difference Methods (FDM) and Finite Element Methods (FEM) are designed, analyzed, and implemented on it. Linear wave equations based on seismic data processing applications are adopted as the targeting PDE problems to demonstrate the effectiveness of this new computer model. Their sustained computational performances are compared with pure software programs operating on commodity CPU-based general-purpose computers. Quantitative analysis is performed from a hierarchical set of aspects as customized/extraordinary computer arithmetic or function units,

compact but flexible system architecture and memory hierarchy, and hardware-optimized numerical algorithms or methods that may be inappropriate for conventional general-purpose computers. The preferable property of in-system hardware reconfigurability of the new system is emphasized aiming at effectively accelerating the execution of complex multi-stage numerical applications. Methodologies for accelerating the targeting PDE problems as well as other numerical PDE problems, such as heat equations and Laplace equations utilizing programmable hardware resources are concluded, which imply the broad usage of the proposed FPGA-enhanced computers.

DEDICATION

To my wonderful and loving wife

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

1. INTRODUCTION

Numerical Evaluation of scientific or engineering problems governed by Partial Differential Equations (PDEs) numerically is in general computationally-demanding and data intensive. In typical numerical methods such as Finite Difference Methods (FDM), Finite Element Methods (FEM), or Finite Volume Methods (FVM), etc., PDEs are discretized in space to bring them into finite-dimensional subspace and solved by standard linear algebra subroutines. Spatial discretizations for realistic Scientific and Engineering (S&E) problems could easily result in millions, even billions, of discrete grid points, which correspond to large linear system equations with the same number of unknowns. If the problem was time-dependent, in order to simulate the transient behavior of the problem, we may need to solve the linear system equations for hundreds to thousands of discrete time steps. Furthermore, if the problem was nonlinear, we have to resort to iterative methods to guarantee the convergence of the numerical solutions, which means solving multiple linear system equations in each time evolution step. Typical applications of numerical PDE problems include but are not limited to Computational Fluid Dynamics (CFD), computational physics, computational chemistry, weather forecast/climate modeling, seismic data processing/reservoir simulation, etc.

The last two decades have seen rapid improvements in performance and complexity of digital computers. Without any doubt, the most convenient computing resources for solving numerical PDE problems are commodity computers. With the continuing renovation of Very Large-Scale Integration (VLSI) semiconductor manufacturing technology, modern commodity CPUs now consist of hundreds of millions of transistors and work at internal clock rates up to several GHz. Their low price and flexible program-controlled execution mode attain commodity CPU-based general-purpose computers feasible for almost all kinds of applications. However, because a large portion of silicon area inside CPUs is committed to sophisticated control

---

This dissertation follows the style of Microprocessors and Microsystems.

logics, the transistor utilization and energy efficiency of such devices are generally poor. Moreover, although the nominal speed of commodity CPUs are skyrocketing, in reality, only a small fraction of their peak performance could be delivered for our computationally-intensive and data-demanding problems. Solving such problems may easily take people hours, days, weeks, even months. Some large-scale problems continue to be unsolvable in an acceptable period of time even on today's fastest supercomputer platforms.

An alternative is to build special-purpose computer systems for specific problems at hand using Application-Specific Integrated Circuits (ASIC) as the core components. Compared with commodity CPU-based general-purpose computers, a majority of in-chip transistors could be devoted to useful computations/operations so that such application-specific systems could achieve much higher computational performance (100X ~ 1000X) with better transistor utilization as well as energy efficiency. However, this approach presents problems such as lack of flexibility, long developing period, and its high Non-Recurrent Engineering (NRE) costs. If the high cost could not be amortized with mass product, this approach would be expensive.

The emergence of Field Programmable Gate Array (FPGA) devices gives people another option to construct a new class of FPGA-based computers. FPGA is one type of "off-the-shelf" digital logic devices, the same as commodity CPUs. Inside an FPGA chip, there are numerous regularly-distributed island-like programmable hardware resources such as logic slices, SRAM blocks, hardwired multiplier blocks, or even processor blocks. Design engineers can configure/program these hardware resources at runtime to perform a variety of basic digital logics such as AND, OR, NOT, FLIP-FLOP etc. Multiple similar or different programmable slices can cooperate to implement complex arithmetic or functionalities with the help of surrounding programmable interconnection paths. This so-called In-System-Programmability (ISP) consumes a considerable silicon area and causes FPGA-based hardware implementation working at a much slower speed than ASIC chips. However, FPGA devices have the potential to accommodate tens, even hundreds, of similar or different arithmetic/function units so that the aggregate

computational performance may still be much higher than what is provided by a commodity CPU. Furthermore, users can utilize the same FPGA-based system to accelerate different problems with the help of the delightful ISP property. From this point of view, FPGA-based computer works as “middleware” between the pure hardware-based approaches (ASICs) and the pure software-based approach (commodity CPUs). It has the potential to provide users with high computational power and while maintaining acceptable flexibility.

This thesis will explore the feasibility of utilizing FPGA resources to accelerate computationally-demanding and data intensive numerical solutions of PDE problems. The research work can be divided into three main parts: the first part proposes a new hardware architecture model as “FPGA-enhanced Reconfigurable Computers”, which takes distinguished features of numerical PDE problems into account so that significant performance improvement could be expected. It begins with an introduction of the motivation for FPGA-enhanced computers, related work, and other background information. Then it discusses the system architecture of this new computer model and its detailed implementation as a single workstation as well as a parallel cluster system.

The second and the third parts of this thesis discuss conceivable methods to accelerate FDM and FEM on the proposed FPGA-enhanced computer platform. Here, I select linear wave equations based on seismic data processing applications as the targeting PDE problems. A hierarchical set of aspects as customized/extraordinary computer arithmetic or function units, compact but efficient system structure and memory hierarchy, and FPGA-optimized software algorithms/numerical methods are proposed and analyzed together with detailed implementations. A great variety of experiments comparing sustained computational performance of these numerical methods running on FPGA-enhanced computers with commodity CPU-based general-purpose computers are carried on to show the superiority of this new computer system. All results can also be applied to accelerate the solutions of other numerical PDE problems such as heat equations, Laplace equations, thereby implying the broad use of the proposed FPGA-enhanced computers.

2. BACKGROUND AND RELATED WORK

2.1 Application Background: Seismic Data Processing

Seismic reflection survey is the most widely used geophysical exploration technique in the petroleum industry and plays a key role in locating underground oil and gas reservoirs for more than sixty years. The basic equipment for the field survey is a source producing impulsive seismic waves, an array of geophones receiving underground echoes, and a multi-channel wave signal displaying and recording system. This method is quite simple in concept: The Earth is simply modeled as stratified medium with material properties such as velocity, density, anisotropy, etc. Impulsive seismic waves are excited on the ground and propagate downward into the Earth. When they encounter interfaces of rock layers, the waves will be reflected back and recorded by an array of geophones deployed on the ground. The elapsed time and amplitudes of reflections could be used to determine underground rock layers' depths and attitudes. The main purpose of seismic data processing is to reduce noises embedded in the reflected seismic signals and convert them into interpretable images of subsurface structures [1].

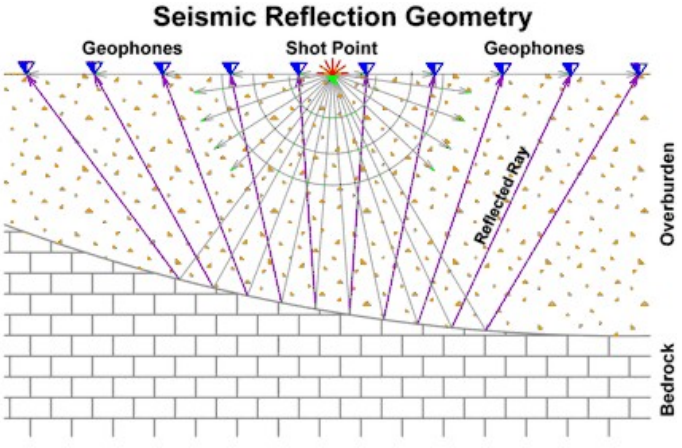

Figure 1. Demonstration of Seismic Reflection Survey

Two main procedures dominate seismic data processing flow: seismic modeling and seismic migration. The mathematical model of energy propagating inside the Earth is acoustic wave equations (2.1) or elastic wave equations (2.2), which can be classified into hyperbolic PDEs.

$$\frac{\partial^2 P(x,y,z,t)}{\partial t^2} - \rho(x,y,z)v^2(x,y,z)\nabla \bullet \left(\frac{1}{\rho(x,y,z)} \nabla P(x,y,z,t) \right) = F \tag{2.1}$$

$$\rho \partial_t v_i = \partial_i \sigma_{ii} + \partial_j \sigma_{ij} + \partial_k \sigma_{ik} \tag{2.2a}$$

$$\partial_t \sigma_{ii} = \lambda \left(\partial_i v_i + \partial_j v_j + \partial_j v_j \right) + 2\mu \partial_i v_i \tag{2.2b}$$

$$\partial_t \sigma_{ij} = \mu \left(\partial_i v_j + \partial_j v_i \right) \tag{2.2c}$$

Seismic modeling is a forwarding problem that simulates the scattering field arising when an impulsive source excites an underground region with known physical properties. Seismic migration is an inverse problem that estimates these physical properties using measured data as initial or boundary conditions. In conventional seismic data processing flow, modeling and migration are two intimately related procedures. They perform iteratively with one's output acting as input to the other: Process start from an initial estimate of underground parameters. Migration methods are then used to collapse diffractions produced by underground scattering points or faults and move dipping reflectors to their physical subsurface locations. After abstracting parameters by analyzing the migrated underground image, modeling procedures are employed to produce a so-called ground "seismogram". By comparing the ground seismogram with the measured dataset, it is possible to indicate where the previous estimations of underground parameters are inaccurate and revise the estimations correspondingly. The process will repeat until the difference between two consecutive iterations is sufficiently small. Mathematically, seismic migration is not a well-posed problem, i.e., boundary conditions provided by the measured dataset are not enough to produce a unique solution. So in general, three to five of such iterations are necessary to obtain an acceptable migrated underground image [2].

Seismic modeling/migration is now the central and culminating step of seismic data processing, and may easily devour 70 to 90 percent of CPU time of the entire

process flow. They are all time-consuming procedures: Even for the simplest acoustic cases, the workload for solving large-scale 3D wave propagation problems could easily exceed the capability of most contemporary computer systems. Although the computational power of commodity computers keeps doubling every 18 months following Moore's Law, by utilizing more and more innovative migration methods, the total elapsed time for processing a 3D middle-scale region is always kept in one week to one month for almost three decades. Old migration methods such as phase-shift or Kirchhoff migration are computationally efficient and affordable for most customers. However, their imaging resolution is not good enough to depict clearly complex underground structures with lateral velocity variation or steep dips. New methods such as frequency-space migration or Reverse Time Migration (RTM) that directly solve acoustic/elastic wave equations would provide infinitely improved accuracy. However, their intensive computational workloads for 3D full-volume imaging are hard to undertake, even for institutes that able to afford the high costs of operating and maintaining supercomputers or large PC-cluster systems. In reality, finishing a designated seismic processing task in a reasonable time period is still much more important than obtaining a better result using improved modeling/migration methods [3].

2.2 Numerical Solutions of PDEs on High-Performance Computing (HPC) Facilities

In the past decade, numerical computing efforts have grown rapidly with performance improvements of general-purpose computers and parallel computing environments. Although the nominal frequency of commodity CPUs is skyrocketing, the real utilization of the peak performance for our targeting problems are in general very poor. The main reason for this inefficiency is that a large portion of transistors in today's commodity CPUs are utilized for control logics or to provide flexible data flow, which attains the prevalent Von Neumann computer architecture model not well-suited for most numerical computing applications. Consequently, many scientific/engineering problems are extremely time-consuming or even unsolvable on contemporary general-

purpose computers, especially when large 2D or 3D geometrical regions are designated as computational domain. It seems that there will always be an urgent demand for more powerful and faster computer systems.

Sustained Floating-Point (FP) performance, which is often represented in terms of Megaflops or Gigaflops, is a key factor in measuring the computational performance of a computer system. Numerical computing applications are generally data intensive as well as computationally demanding. Correspondingly, numerical algorithms/methods always exhibit low FP-operation to memory-access ratio, require considerable memory space for intermediate results, and tend to perform irregular indirect addressing. These intrinsic properties inevitably result in poor caching behavior on general-purpose computers due to their complex system architecture and memory hierarchy. A huge gap always exists between the theoretical peak FP performance of a commodity CPU and the realistic running speed of a program. Pure software methods, from high-level parallelism on PC-Cluster system [4] to low-level memory and disk optimization [5] or even instruction-reordering [6] are exhausted to accelerate the execution of numerical applications. However, even with careful hand optimization, only a few numerical subroutines such as dense matrix multiply or Fast Fourier Transformation (FFT) can achieve 80~90 percent of a CPU's peak performance; For most others, 20~30 percent is very common; with some CPU utilization even reaching as low as 10 percent or even less than 5 percent in realistic applications [7]. Consequently, many large numerical simulation tasks cannot be executed routinely except in institutes that can afford high costs of operating and maintaining supercomputers or large PC-cluster systems.

2.3 Application-Specific Computer Systems

Application-specific computers are computer systems customized for particular uses, rather than as general-purpose computers. Application-specific computing is an active R&D area, both in academia and in industry. Traditionally, it aims at building for each algorithm/application a specific hardware device – the Application-Specific Integrated

Circuit (ASIC) chip, to achieve better implementation in terms of hardware resources, performance, cost, and energy requirements. While general-purpose computing deals with sequential algorithms or concurrent sequential processes, application-specific computers deal with parallel algorithms running in a physical space-time domain. From the technological point of view, it is not a difficult task to design an application-specific ASIC chip and use it as the core component to construct a fully-customized special-purpose computer system with much higher computational performance than general-purpose computers. However, this approach encountered several practical barriers such as high NRE costs, especially for today's VLSI manufacturing technology; long development period, which tends to devour most of the performance advantages; and the most important reason: lack of flexibility.

One widely-cited successful example of special-purpose computers is the 17-year-old GRAPE (short for GRAvity PipE) project managed by a group of computer scientists and astrophysicists at the University of Tokyo [8] [9]. The success of this project rests on the ability to solve a special problem that was hard to solve on general-purpose computers. This project had resulted in the GRAPE family of special-purpose computers. GRAPE was built as a Newtonian force accelerator in the form of an attached acceleration card working in a way similar to graphic accelerators. When simulating a typical large-scale gravitational N-body problem, almost the entire original program is unchanged and still run on the host computer, while only the gravitational force calculations in the most innermost loop are replaced by a function call to the special-purpose hardware. Compared with commodity CPUs that use only a small fraction of their transistors in arithmetic computations, GRAPE essentially utilized almost all available transistors inside the ASIC chip instantiating arithmetic units. In other words, the key trick of GRAPE to outperform general-purpose computers is to optimize the utilization of transistors for a specific application. The single board version of GRAPE-6, which was developed after 2000, achieved the peak performance at 500G flops when coupled to a conventional PC workstation as front end. Such a simple combination could bring about astonishing computational power, equivalent to 1000 contemporary personal

computers, to the desk of an individual astrophysicist. Although the prospect of this product line is fascinating, GRAPE had proven to be too far from success in business: only tens of different versions of GRAPE systems were sold (or donated) to major astrophysical institutes around the world.

2.4 FPGA and Existing FPGA-Based Computers

2.4.1 FPGA and FPGA-Based Reconfigurable Computing

Field Programmable Gate Array (FPGA) is one tpye of “Commercial-Off-The-Shelf” (COTS) digital logic devices that contains numerous island-like programmable hardware resources surrounded by programmable interconnection paths. Design engineers can configure/program such devices at runtime to perform a variety of tasks from basic digital logics to complex function and arithmetic units. FPGAs were noticed in the beginning for their advantages of allowing implementation of customized logic functions and to be reconfigured on-the-fly, thus removing most of the NRE costs from new digital system designs. For these reasons, they have been widely utilized as glue logic, or to build prototyping systems.

As we introduced above, for small designs and/or low production volumes, ASIC becomes a less attractive solution because of its high NRE costs. We predict that this trend would be even more distinct in the future because this cost would be even higher with the evolution of semiconductor manufacturing technology. The emergence of FPGA devices shed new light on these applications. Xilinx, one of the major manufacturers of FPGA devices, even defines its products as hardware-programmable high density ASIC with the time to market and cost advantages over standard ASIC products.

As the cost per gate of FPGAs declines, embedded and high-performance systems designers are being presented with new opportunities for accelerating their applications using FPGA-based hardware platforms. These COTS semiconductor devices are far more flexible than commodity CPUs in that users are not bounded by a given instruction

set anymore. By designing all instructions/dataflow explicitly from the bottom up and programming the chips, users now have a dataflow machine without necessity to decode instruction flow on-the-fly. Such FPGA-based systems combine the general-purpose computing models with the hardware oriented application-specific computing model into a new computing model as reconfigurable computing. It adds to ASICs the flexibility inherent in the programming of Von Neumann computers, while at the same time, maintaining the hardware efficiency inherent in ASICs. Software and hardware are no longer viewed as two completely separated concepts. Traditional software-related topics such as languages, compilers, libraries, etc. are also key components of programmable hardware designs. Furthermore, the re-programmability of these devices allows users to execute different hardware algorithms/procedures on a single device in turn, just as large software packages run on conventional computers. These new computing platforms effectively bridge the performance gap between traditional software programmable microprocessor-based systems and application-specific platforms based on ASICs.

2.4.2 Hardware Architecture of Existing FPGA-Based Computers

With the initial emergence of FPGA-based computing platforms in the 1990s, for the first time in computer engineering society, scientists and engineers had an option to customize their own low-cost but powerful “FPGA-based computers” for specific problems at hand. Such application-specific systems had been widely used to speedup almost all kinds of fixed-point based Digital Signal Processing (DSP) applications. Such problems are in general computation-bounded, so their executions could be significantly accelerated by taking advantage of FPGAs’ high computational potential. As listed in Table 1, some institutes/companies designed and built stand-alone high-performance “FPGA-based application-specific supercomputers” as substitutes to conventional general-purpose computers. Although all of these systems achieved impressive performance improvements over contemporary supercomputers for their targeting

applications, most of them were built for demonstration purpose only, so are in general too expensive to be affordable for most potential users in academy or industry.

Table 1. FPGA-Based Reconfigurable Supercomputers

Systems	Manufacturer	Year	Number of FPGAs	Applications	Hardware Costs
BEE-2 [10]	UC Berkeley	2005	5 per Blade 200 per Rack	Radio astronomy	$ 500K
Starbridge [11]	NASA	2005	11 per system	Aeronautics & astronautics	$ 150K
DN8000K10 [12]	DINI Group	2005	16 per board	Digital circuit emulation	-

Although FPGAs have the potential to provide much higher computing power than commodity CPUs, they tend to be inefficient for certain types of operations such as branch, condition, and variable-length loops so are unsuitable for accelerating control-dominating applications. A natural choice is to couple them with commodity CPUs so that the execution of software could be accelerated by mapping the most computationally-intensive instructions into FPGAs and leaving all other subroutines still running on CPUs. From now on, I will use “FPGA-enhanced general-purpose computers” or simply “FPGA-enhanced computers” to refer to this class of computer systems. By far, there are three popular ways to introduce FPGA resources into a computer system: as adds-on card attached to standard peripheral interface such as PCI or VME [13] [14], as a coprocessor unit coupled with CPU directly [15], or as a heterogeneous processing node connected to computers’ system/memory bus [16]. Generally speaking, the tighter FPGA is coupled with CPU in a computer system, the more accelerations it could achieve due to lower administration overhead and wider communication bandwidth.

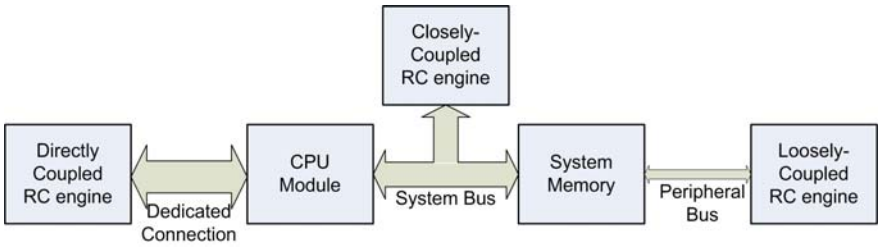

Figure 2. Coupling FPGAs with Commodity CPUs

The traditional and also the cheapest way of introducing FPGA resources into commodity computers is to attach FPGA-based PCI or VME card to peripheral bus of a computer. Almost all key components on such cards are COTS semiconductor devices, and so are cheap and can be easily replaced or upgraded. Comparatively, introducing FPGA resources via system memory buses or dedicated connections changes the architecture of the host machine so would be relatively expensive. The resulting computer system can be classified as Un-Symmetric Multi-Processor (USMP) machine, different from the prevailing Symmetric Multi-Processor (SMP) architecture because FPGAs are introduced into the system as inhomogeneous computing resources. One obvious benefit of such systems is that CPUs and FPGA devices are now sharing the same memory space so that data exchanging between them is convenient and fast. On the down side, this approach doesn't introduce additional memory space/bandwidth except for a limited number of SRAM modules working as FPGA's cache/buffer space. Because most numerical PDE problems we considered here are memory bandwidth bounded; furthermore, memory bus arbitration complicates quantitative performance analysis, the achievable acceleration of the targeting applications on such systems is hard to predict but would not be optimistic.

There are only a few publications [17] [18] addressing the topic of memory hierarchy and data caching/buffering structures on such systems. This is partly because most existing FPGA-based systems are customized for data-streaming based real-time DSP applications, which are in general computation-bounded and require only a limited number of memory spaces for inputs, outputs, and intermediate results. In such systems,

FPGA and memory resources are always abundant and onboard hardware architecture /interconnection pattern are all well-tailored for particular applications. Also, people tend to use local SRAM or even in-chip SRAM slices as working space so that simple and straightforward data buffering arrangements would be enough for satisfactory computational performance.

2.4.3 Floating-Point Arithmetic on FPGAs

Floating-point computations dominate numerical solutions of PDEs. Standard IEEE-754 compliant floating-point arithmetic units are costly on FPGAs because they consume significantly more programmable hardware resources than their fixed-point counterparts. Recently, as FPGA continues to grow in density, people start noticing its high potential in floating-point computations. A large-scale FPGA chip now consists of hundreds of thousands of island-like reconfigurable logic slices. Considering that a typical floating-point arithmetic unit consumes hundreds to thousands of logic slices, a single FPGA device has the potential to accommodate tens, or even hundreds, of similar or different floating-point arithmetic units so that the aggregate computing power could be much higher than what is provided by a commodity CPU. All in-chip programmable hardware resources can be easily customized into different arithmetic units at runtime. This In-System-Programmability leads FPGA to a very attractive option for applications requiring extraordinary arithmetic units that are unavailable in commodity CPUs. Fine-grain (or low-level) parallelism is an important property of FPGA-based computer systems: Different arithmetic units can manipulate their own data sets concurrently, or they can work together in a pipelined manner to increase data throughput significantly. The interconnections among those units could also be customized to match the requirements of specific algorithms so that high sustained data throughput could always be achieved.

In 1994, Fagin et al [19] first testified the feasibility of implementing single-precision floating-point arithmetic units on FPGAs, though the computational

performance was uncompetitive to contemporary commodity CPUs. Since then, as FPGAs continue to grow in density and speed, their attainable floating-point performance increases significantly faster than commodity CPUs. Architectural changes, such as the introduction of on-chip hardwired multipliers further accelerate this trend. In [20], the authors predicted that FPGA devices would yield three to eight times more peak performance than commodity CPUs by 2009.

With the help of commercial or open-source parameterized floating-point libraries [21] [22], floating-point computations on FPGA-based platform is now straightforward and convenient. However, this simple implementation will cost considerable hardware resources and lead to excessive computation latency, which may be inappropriate in reality. Efforts were made to investigate the use of customized floating-point formats [23] or fixed-point format [24] directly to address such problems. However, all these approaches inevitably result in excessive numerical errors and lead to doubtable solutions for rigorous numerical scientists.

2.4.4 Numerical Algorithms/Methods on FPGAs

Migrating software algorithms/numerical methods onto FPGA-enhanced computers are relatively straightforward: while leaving almost all software subroutines still running on the commodity CPU of the host machine, only the most time-consuming kernel portion of the program are replaced by a subroutine calling for the help of FPGAs. Limited by on-chip hardware resources so far, low-level structural or behavioral hardware description languages (VHDL, Verilog, or System C) are still users' common choices to achieve high hardware resources utilization. High-level languages such as C or FORTRAN are based on instruction-driven model so generally cannot result in efficient implementation [25]. Graphic languages such as Xilinx System Generator [26] and Starbridge Viva [11] are other interesting options but seem only suitable for simple DSP applications. Research on automated software fit-in has been conducted [27] for a long time with limited achievements, and is, thereby, considered far from practical.

Since 2000s, increasing research efforts have been conducted for solving numerical PDE problems on FPGA-based platforms. Highlights of research on this track include computational fluid dynamics [28], computational electromagnetics [29], and molecular dynamics [30], to name a few. Although most demonstrated impressive acceleration over contemporary commodity computers, these results were normally obtained by migrating software subroutine directly into application-specified FPGA-based platforms, and hence may be inefficient, expensive, and incompatible with industrial standards. We believe that most commonly-used software algorithms/numerical methods are well-tuned for commodity computers, and so, in general are nonideal for FPGA-based platforms. In order to achieve much higher computational performance, one should design and/or customize new algorithms/methods for their specific applications. Furthermore, these new algorithms must be flexible and robust so that a wide range of commercial FPGA-based platforms could accommodate it effectively and efficiently. Unfortunately, research in this direction is rare.

3. HARDWARE ARCHITECTURE OF FPGA-ENHANCED COMPUTERS FOR NUMERICAL PDE PROBLEMS

As we discussed in Section 2.4, coupling FPGA resources with commodity CPUs is currently the most feasible way to construct a hardware-reconfigurable computer system with acceptable flexibility at a reasonable cost. Most existing FPGA-enhanced computers that have been proposed in recent years are mainly specified for real-time DSP applications with streamed input/output. Their memory hierarchy is simple because in general, DSP algorithms have relatively localized computational patterns, and so do not require complicated data structures or large memory space for intermediate values. However, numerical methods/algorithms for PDE problems in general exhibit low floating-point operation to memory-access ratio, require considerable memory space for intermediate results, and tend to perform irregular indirect addressing for complex data structures. These intrinsic properties inevitably result in poorly sustained computational performance on existing FPGA-enhanced computer systems as well as modern general-purpose computers.

In this section, a new hardware architecture model of the FPGA-enhanced computer is proposed taking into consideration special computational patterns of our targeting problems. When referring to computer architecture, we use the definition from Hennessy & Peterson's famous computer architecture book [31], which refers not only to the Instruction Set Architecture (ISA) of a machine, but also to its detailed implementation. ISA conventionally serves as the boundary between software and hardware. However, on FPGA-enhanced computers, this boundary is blurred. FPGA's In-System-Programmability (ISP) provides users with multiple choices in customizing ISA for their specific problems. For example, users can select either to express sequences of instructions implicitly with internal data paths, or they can customize function/arithmetic units to extend the ordinary instruction set. Furthermore, users are free to specify data buffering/caching sub-system utilizing in-chip programmable RAM blocks or distributed registers according to specific requirements of an algorithm.

Correspondingly, it is now users' responsibility to take care of almost all circuit design details such as latency, timing, data consistency, cache replacement policy, etc.

3.1 SPACE System for Seismic Data Processing Applications

Because of the lack of appropriate commercial FPGA-based platforms, in 2003, we had to propose an imaginary FPGA-based acceleration card [32] called SPACE (Seismic data Processing Accelerator with reConfigurable Engine) for our computationally-demanding and data intensive seismic data processing applications. Acting as a hardware-programmable coprocessor board attached to an Intel-based workstation via local PCI bus, SPACE consists of three main components: FPGA chip, external memory modules, and PCI interface circuit (Figure 3). All of these components are COTS devices, so can be easily replaced or upgraded in the future. The main difference between this design and those we introduced in Section 2.4 is that we do not depend on the system memory space of the host machine as working space. Alternatively, we select to integrate multiple large-capacity memory modules with dedicated memory controllers onboard. This design stresses the simplicity of hardware architecture as well as its capability to manipulate a large amount of input/output data set or intermediate values. The resulting FPGA-enhanced computer platform is also proven to be appropriate for other large-scale numerical PDE problems.

The kernel component of SPACE is an up-to-date Xilinx Vertex II Pro series FPGA chip, which contains a large array of programmable logic slices, along with special function units such as block RAMs, hardwired multipliers, and gigabit serial transceivers. The reason for integrating only one of the largest contemporary FPGA devices on board is based on the consideration that in-chip programmable routing resources are abundant and much more reliable than wires running on PCB boards. For complex algorithms or large problems, multiple SPACE cards could be attached to a single PC workstation to expand programmable hardware resources as well as memory spaces.

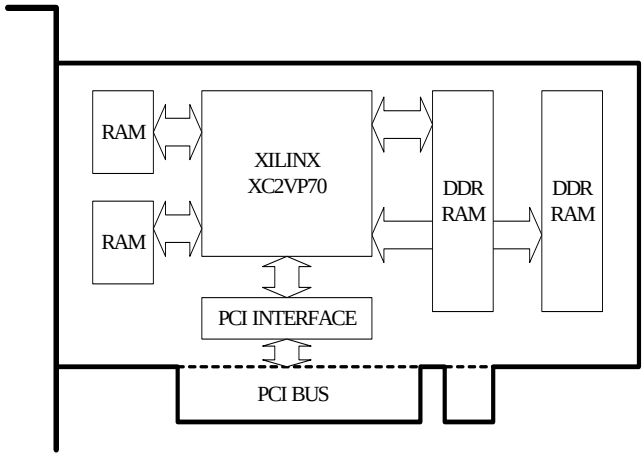

Figure 3. The SPACE Acceleration Card

As we mentioned above, data manipulating capability of a computer system is a pivotal factor for rapidly solving numerical PDE problems. This unfeasible requirement forces us to integrate as many dedicated memory channels as possible into our design to broaden data paths between FPGA device and its external memory. In order to expand memory capacity and bandwidth on SPACE, we integrate two kinds of memory modules with dedicated memory controllers: Two large dual-port static RAM modules acting as high-speed data buffer, and four DDR-RAM modules for saving large data volume up to several Gigabytes. These memory modules are connected directly to the FPGA chip, with a portion of programmable logic resources employed to construct their dedicated memory controllers. For example, a DDR-RAM controller consumes hundreds of logic slices, which is only a trivial fraction of programmable hardware resources inside a large FPGA device. With abundant memory resources, all data and parameters could be placed in-core so that the communication pressure on the local PCI bus is effectively alleviated. Moreover, the main memory of the host workstation can be reduced correspondingly.

The aggregated external memory bandwidth of the proposed SPACE platform is over 20 GByte per second, which is over 3 times wider than a typical PC workstation.

The interconnections among these external memory channels and on-chip arithmetic/function units could be customized on-the-fly with the help of internal programmable routing paths. Consequently, appropriate memory hierarchy could always be adopted for specific problems at hand. Furthermore, users can explicitly manage the access to those independent memory channels so that much higher memory bandwidth utilization could be achieved. However, restricted by the total number of programmable I/O pins of an FPGA device, a limited number of dedicated memory channels could be integrated in practice. Therefore, we predict that external memory-bandwidth would still be the performance bottleneck especially for our targeting numerical PDE problems.

A standard 33MHz PCI bridge chip is used to provide the I/O interface between the coprocessor board and the host workstation, through which about 100Mbyte/s of continuous data transfer rate can be achieved. The narrow I/O bandwidth might become a severe performance bottleneck preventing us from taking full advantage of FPGA's computational potential. Fortunately, we can easily replace it with a 66MHz PCI, 64bit/133MHz PCI-X, or the newest 16X PCI-Express, which can provide people with up to 8GB/s data transfer rate in two directions. However, as the IO bandwidth bottleneck is greatly alleviated, the communication latency between CPU and FPGA caused by PCI controller may pose another problem.

Three years after the SPACE platform was proposed, we noticed that commercial products with similar structures are now being introduced into the market. Minor differences may exist, such as integrating several small but cheap FPGAs, supporting only SRAM or SDRAM modules to save costs, or utilizing part of FPGA resources to implement PCI interface instead of a dedicated PCI controller, those three main components: FPGA devices, large-capacity memory, and PCI interface are indispensable. The emergence of these products also indicates that industry is beginning to not only show its interest in FPGA-based computing technique, but it is also trying to solve realistic problems using the technique.

3.2 Universal Architecture of FPGA-Enhanced Computers

Based on experiences drawn from the SPACE project, we believe that by applying FPGA-based reconfigurable computing technology, we can accelerate the executions of a wide range of numerical PDE problems. Our belief is based on the fact that some kernel subroutines of these programs consume a large portion of the programs' execution time. These subroutines are usually short in length and, hence, are suitable to be accelerated by FPGA. However, the concept of FPGA-based hardware reconfigurable computing is currently still not widely accepted. Also, because standard hardware architecture for such computers has not been developed yet, it is apparent that academia and industry still hesitate in adopting FPGA resources into new mainframes. Grounded on this situation, we abstract in this section a universal architecture of FPGA-enhanced computer model for rapid solution of numerical PDE problems. Besides the capability to accommodate a wide range of S&E problems, we also stress the simplicity of its hardware implementation. We ask the resulting FPGA-enhanced computer system to have a comparable price to a conventional PC workstation. Other appropriate properties, such as compatibility and scalability, are also taken into consideration.

What we proposed is still an FPGA-based acceleration card attached to commodity computers as shown in Figure 4. As the SPACE system, it integrates multiple large memory modules onboard with dedicated memory controllers as working space. Abundant on-board memory space and wide memory bandwidth partly compensate for the relatively narrow IO bandwidth because we can always set subroutines running in-core. IO only happens at the beginning and the end of the process, and therefore will not dominate a program's total running time. The FPGA board and its host computer work as two loosely coupled systems with their dedicated memory space. Original software subroutine could be easily migrated to this new FPGA-enhanced computer platform by invoking the FPGA board as subroutine. Only the most time-consuming kernel subroutines are accelerated by FPGA so that the software migration workload is modest and the correctness of accelerated results can be easily verified.

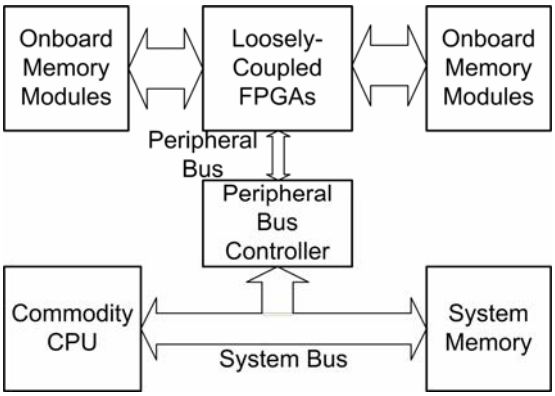

Figure 4. Architecture of FPGA-Enhanced Computer

Instead of proposing a new computer model with fixed architecture, here we tend to specify only guidelines so that computer designers could have more choices in tuning the hardware structure of their FPGA-enhanced computers systems. For example, we do not specify how many SRAM or SDRAM modules/channels on such systems. Indeed, because we adopted an extremely simplified architecture with only FPGA, memory, and CPU interface circuit as core components, we always tend to utilize as many external IO pins of the FPGA device for dedicated memory controllers as possible. With these physical external memory channels deployed on board, users can customize the memory hierarchy based on the demands of specific applications utilizing FPGA's internal programmable routing paths as well as SRAM slices and distributed registers. The clock frequencies applied to FPGA devices and external memory modules are within the same range at hundreds of Millions Hz. Therefore, we could treat all internal and external memory elements as a flat memory space to simplify system architecture and save FPGA resources; or we could introduce complicated caching or buffering schemes to further enhance data reusability and improve utilization of external memory bandwidth.

Besides the choices for memory hierarchy, system designers also have the freedom to select appropriate IO interface between FPGA and CPU. IO bandwidth decides where to place the dividing line between hardware and software, and therefore plays a key role in determining weather a specific application could be accelerated effectively. The

relatively narrow IO bandwidth provided by peripheral bus such as PCI or VME might become a severe performance bottleneck preventing us taking full advantage of on-board FPGA's computational potential. Fortunately, some commodity CPU vendors now start opening their system bus for exterior access. For example, AMD's Opteron processor could support three high-speed HyperTransport links, which provide up to 19.2 GB/s aggregated data communication bandwidth. With the help of these point-to-point paths, FPGA resources now could be placed much closer to commodity CPU with much lower administration overhead. They can even select to share the same system main memory space, although it would not result in significant performance improvements for our data intensive numerical PDE problems.

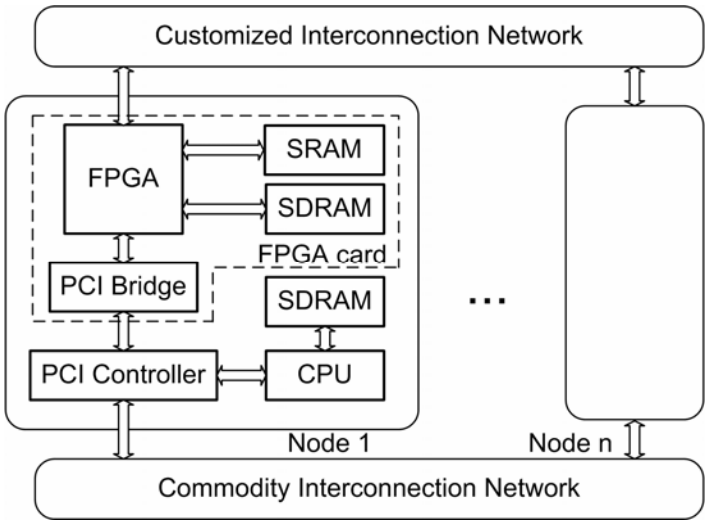

Figure 5. FPGA-Enhanced PC Cluster

Old PC Cluster systems could be easily upgraded to FPGA-enhanced PC Cluster by inserting the acceleration card to each processing node as shown in Figure 5. The existence of FPGA resources doesn't affect the original functionality of host machines so that this upgrade is transparent to old software. The original interconnection network is untouched so that parallel efficiency of the new FPGA-enhanced PC Clusters is at

least as good as its predecessor, although the limited network bandwidth might pose a severe performance bottleneck. An additional newly-deployed interconnection network could be introduced, utilizing FPGA's on-chip Multi-Gigabit Transceivers (MGTs). We will explain this approach and analyze its performance in detail in the next section.

3.3 Architecture of FPGA-Enhanced Computer Cluster

As we mentioned above, multiple FPGA-enhanced computers could be easily interconnected via high-speed commodity network to construct an FPGA-enhanced PC cluster system. However, most prevailing Local Area Network (LAN) standards such as Gigabit Ethernet, InfinityBand, Myrinet, etc., do not provide well-scaled performance, especially when the number of interconnected processing nodes exceeds thousands. The main reasons for this so-called "thousand-processor barrier" are shared physical data links (Switching technique may alleviate this problem to a certain level.) as well as overhead of network protocols.

To improve the scalability of the proposed FPGA-enhanced PC cluster system, we introduce another interconnection network into the system utilizing FPGAs' on-chip Multi-Gigabit Transceivers (MGTs). A single FPGA chip contains 8~20 dedicated MGTs, each of which could provide a point-to-point data link with raw communication bandwidth up to 10 Gigabits per second. For better compatibility, users could select to construct network interfaces with embedded standard protocol utilizing FPGA resources. Furthermore, because there is no specific network protocol binding with these physical resources, we are free to customize our own simplified data communication links for improved bandwidth utilization and shorter communication latencies. Multiple dedicated network interfaces also give us the freedom to select topology of this new interconnection network. Besides simply utilizing standard high-speed commodity network to augment total effective bandwidth, we can introduce localized interconnection data links among adjacent reconfigurable computing nodes. The topology could be set as ring (2 communications channels per node), 2D mesh/torus (4 channels per node), 3D cube (6 channels per node), or their hybrid. The corresponding

network performances are all well-developed, so that specific algorithms could select appropriate network topology to improve their parallel efficiency.

We use seismic wave modeling and migration problems as an example to analyze/predict the performance of high-order FDM on the proposed FPGA-enhanced PC cluster. (Details of the underlying numerical methods can be found in Section 5.1.) For such problems, a 2D torus interconnection network is a good choice to effectively balance the performance and system complexity. Suppose we plan to upgrade a convenient PC Cluster system mounted on multiple industrial standard cabinets. A sketch of the resulting reconfigurable computer cluster with 2D torus network is shown in Figure 6. (Only new interconnection paths are shown here; the original network remains unchanged, so is ignored.) Each FPGA acceleration card is required to equip at least four external physical data link ports built with FPGA's on-chip MGTs. Standard high-speed network optical-fiber or copper cable could be used for these point-to-point communication paths depending on their lengths. The deployment and driving of these short links between neighboring processing nodes would be easy and convenient.

Here, we consider only the simplest 3D acoustic wave modeling problems in large-scale 3D space and assume follows:

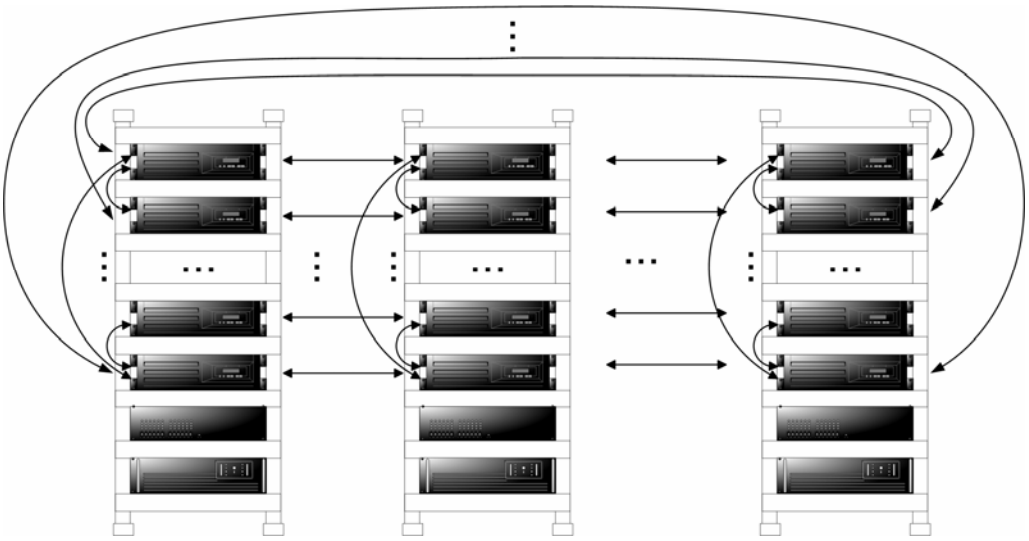

Figure 6. 2D Torus Interconnection Network on Existent PC Cluster

1. Z-axis is shorter than x- and y-axis. In other words, the number of spatial grids along z-direction is much less than the numbers along other two directions. Here we set this number as 2000, which corresponds to 10 kilometer depth if the sampling interval is 5 meters.
2. The entire spatial domain is divided into m-by-n sub-domains along x- and y-directions. Each reconfigurable processing node contains enough FPGA and RAM resources for accommodating all spatial grid values as well as necessary parameters of one sub-domain.
3. The 2D torus interconnection network provides independent point-to-point communication paths, which do not interfere with each other and can work concurrently.

The first example is to show the capacity of the proposed platform for handling large problems. Suppose each FPGA-enhanced processing node contains a large sub-domain with $500 \times 500 \times 2000 = 5 \times 10^8$ spatial grids. We need at least four memory spaces of the same size to save wave field values as well as media parameters such as velocity and density required by 2^{nd}-order time evolution scheme. Therefore, the

memory space on each acceleration card should be at least $5\times10^8\times4=2G\ words=8GByte$, which is feasible with today's high-density SDRAM modules.

Suppose we use 10^{th}-order spatial FD scheme to simulate the propagation of waves. The number of floating-point operations we required to update the wave field value at one grid point for one time-marching step is about 50. So the total number of floating-point computations on each processing node is $5\times10^8\times50=2.5\times10^{10}$. When running on a general-purpose computer with 1G FLOPS sustained floating-point performance, this task could be finished in 25 seconds. (Here, we ignore the performance degradation caused by the network.) However, if we ran the same job on the proposed FPGA-enhanced PC cluster system, a conservative 5G FLOPS sustained performance is achievable so that the job could be finished in only 5 seconds. Please notice that the performance improvement is achieved from FPGA's superior computational power solely because IO communication doesn't pose a performance bottleneck here.

Consider the performance of the newly deployed 2D torus interconnection network. At every time-marching step, each FPGA-enhanced processing node needs to exchange boundary node values with its four neighbors. Simple calculations show that there are in total $490\times490\times2000\approx4.8\times10^8$ internal spatial nodes and about 2×10^7 boundary points. Values of these boundary points are exchanged with neighbors via four on-board MGTs simultaneously, so the lower-bound of the communication bandwidth for each MGT port is 20 million bytes per time-marching step. It also equals to only $2\times10^7\times8\div5=32$ M bits per second, which is only a tiny fraction of what is provided by an MGT port.

Now, we consider a much smaller problem with only $100\times100\times2000=2\times10^7$ spatial grids in each sub-domain. The total number of on-board memory space required in this case is about 320 Mbytes. The total number of floating-point computations is 1G for each time step, which could be finished in 0.2 second on the proposed FPGA-enhanced PC cluster system.

The ratio of the number of internal points and boundary points is changed dramatically. We now have 1.62×10^7 internal points and 3.8×10^6 boundary points. It also means that the ratio of communication to computations is now much larger than the previous case. Thereby, the conventional commodity interconnection network would not work efficiently in this case. For the customized 2D torus interconnection network, the lower-bound of the communication bandwidth for each MGT port is now 3.8 million bytes per time-marching step, which equals to $3.8 \times 10^6 \times 8 \div 0.2 = 152$ M bits per second, still only a fraction of what is provided by a MGT port.

From these two examples, we can conclude that the computational performance of our finite deference based wave modeling problem is now decided solely by the computational power of the FPGA-enhanced processing node with the help of the new 2D torus interconnection network. If the problem size is not too small, we can even execute computations and communications sequentially without significant performance degradation.

4. PSTM ALGORITHM ON FPGA-ENHANCED COMPUTERS

4.1 PSTM Algorithm and Its Implementation on PC Clusters

Diffraction summation based Pre-Stack Kirchhoff Time Migration (PSTM) algorithm is one of the most popular migration methods in seismic data processing because of its simplicity, robustness, and good target-orientation [33]. This algorithm is derived from Huygens' Principle and mathematically provides a high-order approximate integral solution to acoustic wave equations (2.1) [34]. Practical PSTM tasks for large-scale 3D seismic surveys are computationally intensive and cannot be used routinely except in institutes/companies able to afford the high cost of operating and maintaining supercomputers or large PC-cluster systems. In this section, I will use this algorithm as the first example to show the remarkable computational power of the proposed FPGA-enhanced computer system. Specifically, when operating on commodity computers, this algorithm consumes over 90 percent of CPU time to execute its short but time-consuming kernel subroutines for billions of iterations. By accelerating the evaluations of these kernel subroutines with a customized arithmetic unit, our new FPGA-based solution operated over 10 times faster, allowing people to produce a satisfied underground image a lot faster.

Before we continue, I will briefly explain several specialized terms which will be referred in this document. In seismic reflection survey, an array of thousands of geophones is regularly distributed across the ground surface to detect the intensity and elapsed time of reflected seismic waves. The time-series recorded by each geophone during an exploding experiment (one shot) is called a "seismic trace", which is characterized by the ground positions of the source and the receiver. The set of traces recorded by all receivers during one shot forms a "common shot gather". The data set collected from a large number of experiments of multiple shots at all receiver positions forms a 5D data volume $D(s,g,t)$, where $s=(x_s,y_s)$ and $g=(x_g,y_g)$ are the surface coordinates of the shot and receiver, t is the elapsed time of the recorded dataset.

Define “two-way travel time” as the duration of an acoustic impulse starting from the shot position, reflecting at the scatter point, and then traveling back to the ground receiver, the total down-up two-way travel time is determined by:

$$T_{SR} = T_s + T_R = \sqrt{\tau^2 + \frac{S^2}{V_\tau^2}} + \sqrt{\tau^2 + \frac{R^2}{V_\tau^2}} \tag{4.1}$$

Where T_S is the travel time from the shot point to the scatter point; T_R is the travel time from the scatter point to the receiver point; S and R are the distances between the surface mirror of the underground scatter point and the shot point or receiver point, respectively; τ is the pseudo-depth of this scatter point in the output section; and V_τ is a priori estimation of the Root Mean Square (RMS) velocity at point τ. Figure 7 schematically shows the relationship between the source, receiver and scatter points in a 2-D profile.

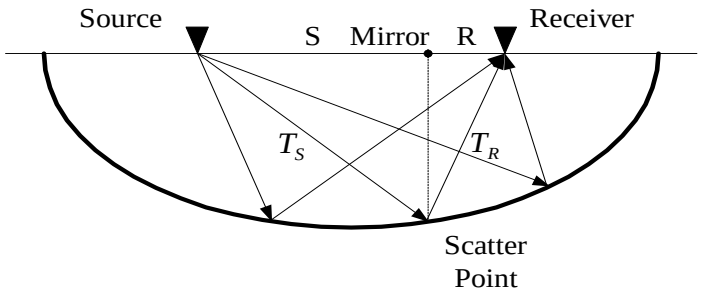

Figure 7. The Relationship Between the Source, Receiver and Scatter Points

The PSTM algorithm assumes that the energy of a sampled point t_0 in an input trace is the superposition of reflections from all the underground scatter points that have the same travel time $T_{SR} = t_0$ for the fixed source and receiver positions. Therefore, for one sample point on an input trace with known source and receiver coordinates, its energy should be spread out to all possible scatter points according to its travel time T_{SR}. The locus of all possible scatter points with travel time T_{SR} in a constant velocity

medium is also depicted in Figure 7, which constitute an ellipse in 2D profile following the geometric definition. In order to retrieve all underground scatter points, the energy of an input trace must be distributed to all possible scatter points correctly, after which the energy from different input traces is added together at each pseudo-depth point in the output section. For amplitude preserving PSTM, additional oblique factor calculations [35] [36] and corresponding multiplications of this factor are also indispensable.

The PSTM algorithm is computationally intensive: we need to evaluate the two-way travel time in Equation (4.1) iteratively for enormous times. The computational complexity of this algorithm is $O(N_x \cdot N_y \cdot N_\tau \cdot N_s \cdot N_g)$ for 3D cases [37]. Traditionally, only high performance super-computers can finish this time-consuming 3D PSTM algorithm within an acceptable time period. Recently, PC cluster has emerged as a cheap and efficient alternative to supercomputers. A PC cluster system consists of one or several servers and many workstations interconnected via high-speed network. Taking advantage of commodity hardware and Linux OS, a PC cluster system sometimes could achieve performance that is comparable to supercomputers at an affordable price for network non-intensive applications.

PSTM algorithm could be parallelized ideally when running on PC cluster systems with limited network bandwidth. The server of the cluster broadcasts input traces to all workstations and each workstation migrates these input traces into its local output section (x, y, τ). Once all input traces are migrated, the server collects all partial results from workstations and forms a final migrated image. Ignoring the initial parameter distribution step and the final data collection step, the only data flow is to broadcast input traces from the server to workstations, and there is no communications among those workstations. By parallelizing the migration task in output surface coordinates (x, y) and sequentially iterating over the pseudo-depth axis τ [38], the communication between the server and workstations is minimized. Algorithm 1 is the kernel portion of PSTM procedure executed on every workstation.

The performance provided by a PC cluster system is nearly linear to the number of interconnected workstations for this parallelized PSTM algorithm. However, when

additional workstations are inserted into the system, the shared communication channel becomes a performance bottleneck. Furthermore, the reliability problem inevitably becomes a serious one for systems with thousands of interconnected workstations. Employing more powerful workstations can effectively alleviate these problems at the cost of increased system price.

Algorithm 1. Program Flow of PSTM Kernel Subroutine Running on PC Workstations

```
......
Receieve one input trace from server
Prepare parameters for this trace
For every local output trace in this workstation
    For every pseudo-depth point on this output trace
        Calculate travel time  TSR for this output point -
                -associated with the position of  this input trace
        IF (TSR > Tmax)
                THEN finish this output trace
        Fetch data from input trace indexed by TSR
        Anti-aliasing filtering
        Calculate oblique factor
        Scaling the selected input data by oblique factor
        Accumulate scaled input data to this output point
    End
End
......
```

4.2 The Design of Double-Square-Root (DSR) Arithmetic Unit

As we mentioned in Section 4.1, the evaluation of two-way travel time defined by Equation (4.1) is the most time-consuming part of the PSTM algorithm. It contains five multiplications, three additions, and two square-root operations, ten floating-point computations in total. Division by velocity in this equation can be replaced by multiplication using slowness table, which is the reciprocal of velocity. To complicate the situation, there is no hardwired floating-point square-root arithmetic units integrated inside most commodity CPUs. People have to rely on software routines to approximate their values, which leads to the evaluations of square root tens, even hundreds, times slower than standard computer arithmetic such as multiplication and addition. Because the evaluation of a square root poses a severe performance bottleneck and will dominate the total running time of PSTM algorithm, people sometimes refer to Equation (4.1) as the "Double Square Root (DSR) equation".

The simplest and most straightforward (although not the most preferred) way to evaluate the DSR equation on FPGAs is to construct a large computing engine with ten standard floating-point arithmetic units. To ensure high computational throughput, each arithmetic units in the computing engine should be fully-pipelined internally. However, besides consuming too many hardware resources, the latency of a fully-pipelined square-root unit is almost ten times more than a pipelined multiplier or adder. Correspondingly, the accumulated pipelining latency of the large computing engine would become a serious issue and significantly degrade its achievable computational performance.

4.2.1 Hybrid DSR Arithmetic Unit

It is not always the best solution for an FPGA-based system to implement an algorithm by simply mapping software subroutines into its programmable hardware resources. Taking into consideration special properties of an algorithm along with characteristics of FPGA would always produce a hardware-efficient solution. For the

PSTM algorithm, CORDIC [39] unit is a good choice to evaluate T_s and T_R in Equation 4.1 by regarding $\sqrt{X^2+Y^2}$ as a vector norm in 2D Cartesian coordinates. CORDIC is a vector-rotation-based hardware algorithm for evaluating trigonometric and other transcendental functions using only hardwired shifters and adders [40]. The highly regular structure attains it appropriately for FPGA-based implementation. Now the computing engine needs only two multiplications, two CORDIC units, and one addition to evaluate the DSR equation, which achieves more than 50% of resources reduction.

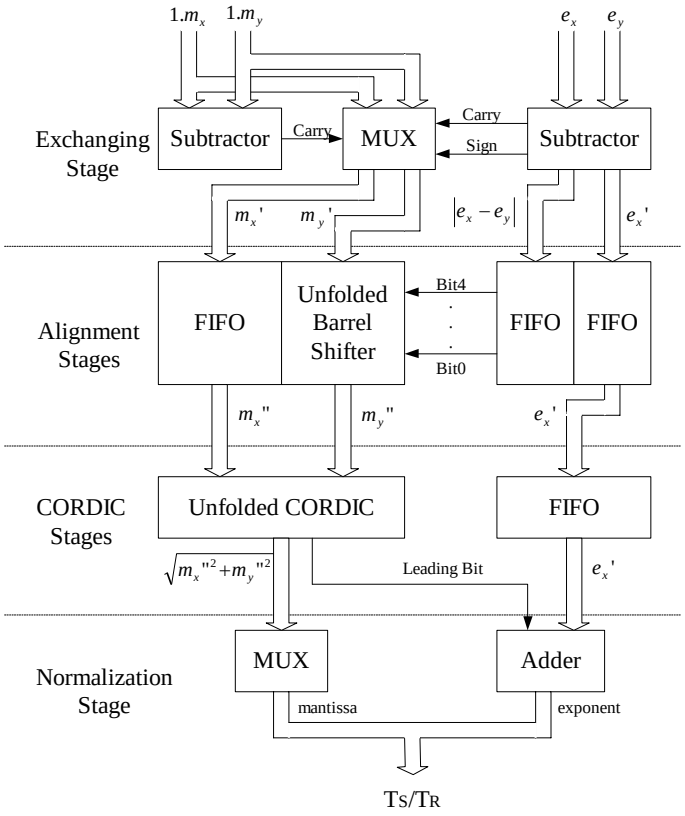

Figure 8. Hardware Structure of the Hybrid DSR Travel-Time Solver

To be compatible with ordinary seismic data processing software, the input and output values of the arithmetic units in our design should all be in floating-point format.

Floating-point CORDIC in pure hardware is expensive because of the indispensable normalization and alignment stages before and after every CORDIC iteration. According to characteristics of our application, we designed a modified floating-point/fixed point hybrid CORDIC unit. This arithmetic unit can provide similar, or even better, error bound than standard floating-point arithmetic, but consumes nearly the same hardware resources as pure fixed-point CORDIC. Figure 8 shows the corresponding hardware structure.

This hybrid CORDIC unit has the following unique properties, which distinguishes this design from others [41]:

- It utilizes fully pipelined hardwired add-and-shift stages, so can achieve much higher computational throughput than the recursive implementation proposed in [41].
- The first stage accepts two floating-point inputs and converts them into an internally aligned floating-point format. The exponent of these two aligned operands is kept unchanged while two mantissas are fed into a pipelined fixed-point CORDIC unit as inputs. The word width of the mantissas is extended to ensure high numerical accuracy.
- The alignment of two floating-point inputs guarantees the same dynamic range as standard floating-point arithmetic. The extended word width of mantissas ensures the same, or even better, numerical error bound.
- The physical character of the application implies that all inputs to the CORDIC unit are positive numbers. By parallel comparison of two exponents and two mantissas, we can decide which operator is larger in one pipeline stage. So an exchanging stage is placed in front of those alignment stages to eliminate one costly barrel-shifter.
- Ordinary CORDIC algorithm needs two additional leading bits to prevent overflow because the largest amplitude gain is 2.33. The first exchanging stage in our design eliminates the first 45° rotation, so changes the processing gain to 1.647. Correspondingly, only one aditional leading bit is needed here.
- No Leading-Zero-Detector (LZD) is needed in this new implementation. LZD is used in ordinary CORDIC units for normalizing intermediate results of floating-point

arithmetic. Its implementation is not as easy as it looks. Ordinarily, complex hardware structures such as array or tree would be involved [42].

- The final normalization step of the floating-point multiplication $X \cdot \frac{1}{V}$ is eliminated because the carry bit of the exponent subtraction in the exchanging stage can absorb the possible exponent increment.
- Only the vector norm output of the CORDIC unit is needed, so we can save about 33 percent of FPGA resources by omitting the Z channel output completely.
- The final floating-point addition of T_s and T_R is simpler than an ordinary floating-point adder because all of these two operators are positive so that subtraction doesn't need to be taken into account.

In order to analyze the error property of our design, a C program was used to simulate the exact iterative operations of the proposed CORDIC unit. In every single experiment, we randomly create one million pairs of single-precision floating-point numbers as inputs. Outputs of the proposed hybrid CORDIC unit are compared with results produced by standard double-precision floating-point arithmetic. Maximum errors and average errors are recorded/calculated and their values are amplified by 2^{20}.

Table 2. Error Property of the Hybrid CORDIC Unit with Different Guarding Bits

Word-width/Guarding Bits	Average Error (ppm)	Max. Error (ppm)
25/0	0.0376	0.305
25/1	0.0181	0.170
25/2	0.0092	0.075
25/3	0.0046	0.034
25/4	0.0023	0.018
25/5	0.0012	0.009
floating-point	0.0260	0.12

Table 2 shows the simulation results of the proposed hybrid CORDIC unit for fixed word-width b=25 (One bit is added for preventing overflow.) with different guarding bits. Computational errors with single-precision floating-point arithmetic for the same operations are also listed as references. We can observe from this table that a 25-bit hybrid CORDIC unit with two additional guarding bits provides people with similar precision as single-precision floating-point arithmetic. If the same average relative error criteria is acceptable, a 25-bit CORDIC with one additional guarding bits is enough.

4.2.2 Fixed-point DSR Arithmetic Unit

A more ambitious design tries to take into consideration the physical meaning of two-way travel time. The value of travel time is used in the final accumulation stage as a time index to fetch the proper sampling point in input traces. Its value should be bounded by the moment of the last sample in an input trace $T_{\max}$, which is less than 16 seconds in most realistic cases. The sample interval of input trace is usually coarser than 1ms. If we treat those time indices as fixed-point numbers, much simpler fixed-point arithmetic could be employed to evaluate Equation (4.1). Figure 9 shows the fixed-point format used in this approach and Figure 10 is the structure of the fixed-point DSR travel time solver.

According to the preceding analysis, the worst-case absolute errors of time indices should be smaller than 0.5ms, which means the error bound for every fixed-point CORDIC channel is 0.25ms. There are two error sources: the rounding error produced by the first conversion stage that converts two floating-point numbers into aligned fixed-point values, and the computation error introduced by the fixed-point CORDIC unit. Table 3 lists average and maximum rounding errors caused by the conversion stage with different fraction word width. Implementation results of the fixed-point CORDIC unit with different word-width and guarding-bits are listed in Table 4.

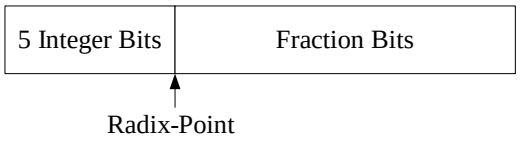

Figure 9. Output Format of the Conversion Stage

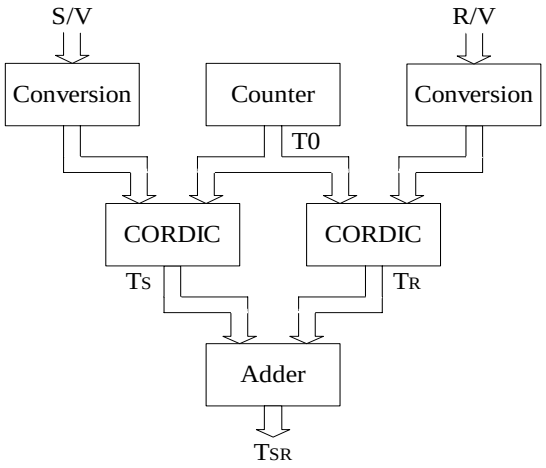

Figure 10. Hardware Structure of the Fixed-Point DSR Travel-Time Solver

We can conclude from these two tables that computational errors would dominate our final results. Obviously, 19-bit fixed-point CORDIC with zero guarding bit (5 integer bits and 13 fraction bits) is the best choice. Another bonus of this pure fixed-point travel-time solver approach is that the T_0 can be easily generated by an integer counter.

Although this fixed-point travel-time solver is practical for the engineering standard and could save considerable FPGA hardware resources, a weak point of this implementation is that the result produced by fixed-point CORDIC would be too coarse to be utilized as inputs to interpolation methods. Therefore, we will use only the hybrid CORDIC approach in the following performance comparison.

Table 3. Rounding Error of the Conversion Stage with Different Fraction Word-Width

Fraction Word-width	Ave. Rounding Error	Max. Rounding Error
10	0.000236	0.000691
11	0.000118	0.000345
12	0.000059	0.000174
13	0.000030	0.000087

Table 4. Errors of the Fixed-Point CORDIC Unit with Different Word-Width and Guarding Bits

Word-width/Guarding Bits	Average Error	Max. Error
17/0	0.000123	0.000669
17/1	0.000077	0.000386
17/2	0.000066	0.000312
18/0	0.000061	0.000337
18/1	0.000042	0.000240
18/2	0.000038	0.000198
19/0	0.000033	0.000193
19/1	0.000029	0.000172
19/2	0.000027	0.000141

4.2.3 Optimized 6th-Order DSR Travel-Time Solver

The travel time in Equation (4.1) is a 2nd-order approximation to the following Taner's travel time equation [43] for horizontally stratified medium model:

$$T_X{}^2 = c_1 + c_2 X^2 + c_3 X^4 + c_4 X^6 + \cdots . \qquad (4.2)$$

Where X is the offset and c_k are priori-estimated coefficient tables associated with the output section. (For example, $c_1 = \tau^2$ and $c_2 = \frac{1}{V_\tau^{\,2}}$)

The accuracy will be improved significantly when factoring higher order series into the travel time calculation. People may prefer the 4th-order or 6th-order schemes because the evaluation of the coefficients for higher than 6th order terms are impractical. In our design, we adopt the optimized 6th-order scheme proposed by Sun et al. [44] as follows:

$$T_{SR} = (T_S + cc\frac{S^6}{T_S}) + (T_R + cc\frac{R^6}{T_R}) \tag{4.3}$$

Where

$$T_S = \sqrt{c_1 + c_2 S^2 + c_3 S^4} \tag{4.4}$$

$$T_R = \sqrt{c_1 + c_2 R^2 + c_3 R^4} \tag{4.5}$$

The definitions and values for the first three coefficients are the same as Equation (4.2), but the c_4 term is modified by taking other higher order terms into account, thereby resulting in the coefficient cc.

The hardware implementation of the optimized 6th-order DSR travel time solver is a direct expansion of our previous design. The evaluation of Equation (4.3) is straightforward. Equation (4.4) and (4.5) can be rewritten as:

$$T_X = \sqrt{(\sqrt{c_1 + c_2 X^2})^2 + c_3 (X^2)^2}\ . \tag{4.6}$$

Obviously, two cascaded CORDIC units can finish this calculation.

Three coefficients table are needed in this scheme compared with only one RMS velocity table in the previous case. So each evaluation of T_{SR} has to access three coefficients from memory. Now, memory capacity and its bandwidth could become a problem for this scheme.

4.3 PSTM Algorithm on FPGA-Enhanced Computers

Although the programmable hardware resources inside an FPGA chip have increased greatly in recent years, they are still not rich enough to fit in a complicated program. The hardware-software hybrid approach is the most feasible way to accelerate a program on FPGA-enhanced computer platform. In order to be an effective alternative, FPGA-based solutions should be at least one-order faster than software executed on traditional CPU-based computers. Although some kernel subroutines could be accelerated significantly by FPGA, governed by Amdahl's Law (Refer to Equation (4.7) ~ (4.9) in Section 4.4), the overall acceleration of a program may not be satisfactory because of the existence of un-accelerated subroutines.

The feasibility of applying FPGA technology to accelerate the PSTM algorithm is based on the fact that over 90 percent of the CPU time of the program is consumed by billions of iterations of inner loops. This short but time-consuming kernel subroutine is suitable for acceleration by the FPGA device. As we mentioned in Section 3, good acceleration results depend greatly upon where to place the dividing line between hardware and software. In our design, this line is placed to balance the computational workload between FPGA and CPU: we always tend to exploit most of FPGA's computational potential accelerating the execution of a program, at the meantime, still keep enough workload running on the host machine to saturate its computing power.

Algorithm 2 shows the program flow of the PSTM kernel subroutine executed on FPGA-enhanced PC workstations. The bold portion of the program is now migrated into FPGA. We can easily tell that the dividing line of hardware and software was placed so that two of the most inner loops are executed in FPGA. All related computations are executed in-core, in other words, there are no intermediate results transferring via the interface between the acceleration card and the host CPU, except in the initial transmission of input traces. There are only trivial differences between Algorithm 1 and Algorithm 2, which means that the software migration workload is trivial. The new PSTM program running on the FPGA-enhanced PC workstation is almost the same as

the software version except that it invokes the FPGA-based acceleration card as a subroutine. Input traces are transmitted into the card as input parameters. When all calculations regarding one input trace and all local output traces are finished, a signal is sent back from FPGA to the host machine to activate the transmission of the next input trace. After all input traces are processed, the final output result is read from the acceleration card for display or further processing steps. When running on an FPGA-enhanced PC cluster system, because the execution of the program's inner loops are accelerated significantly by FPGAs, more input traces could be processed in unit time. Obviously, the actual data transferal rate via the interconnection network would be much higher than before. Because most traffic is to broadcast input traces from the server to workstations, the additional communication overhead in general introduce only moderate performance degradation.

Algorithm 2. The Program Flow of the Accelerated PSTM Kernel Subroutine

```
.....
For every input trace in a field data volume
Prepare parameters for this trace
    Download this trace and its parameters to SPACE
    For every output trace allocated to this board
        For every pseudo-depth point on this output trace
                Calculate travel time Tsr for this output point-associated with the position
                - of this input trace
                IF (Tsr > Tmax)
                THEN finish this output trace
                Fetch data from input trace indexed by Tsr
                Anti-aliasing filtering
                Calculate oblique factor
                Scaling fetched data by oblique factor
```

```
                Accumulate scaled input data to this output point
            End
        End
    End
.....
```

Figure 11 is the structure of one customized computing engine for evaluating the bolded section of the PSTM algorithm shown in Algorithm 2. In order to achieve a high computing speed at one accumulation per clock cycle, every arithmetic unit inside the computing engine is carefully designed to maximize its data throughput. They also are carefully deployed inside the FPGA chip because their physical layout will affect the data flow paths, which in turn affect the sustained execution speed. If there were still free FPGA resources available on board, several identical computing engines could be instantiated to manipulate their own data sets concurrently. Furthermore, multiple FPGA boards could be attached to a single host workstation to increase its computing power dramatically.

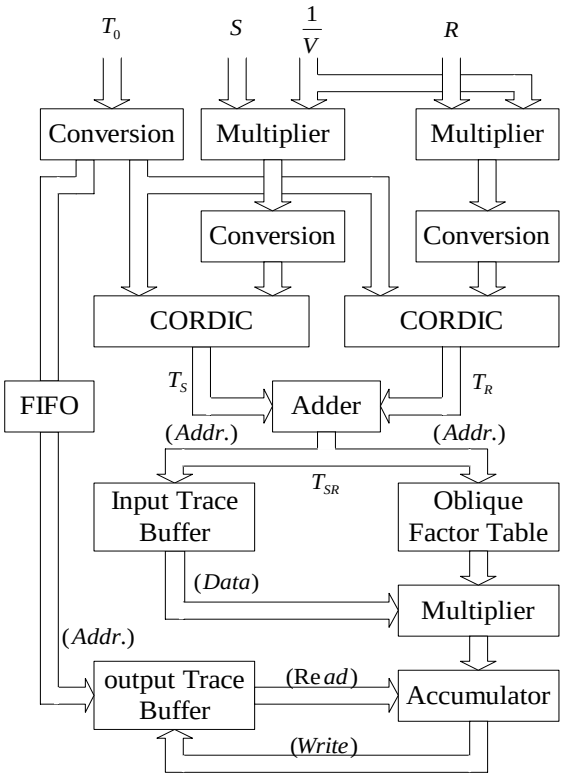

Figure 11. Hardware Structure of the PSTM Computing Engine

4.4 Performance Comparisons

In this section, I compare the computational performance of the FPGA-specific PSTM algorithm with its pure software counterpart running on a referential Intel P4 2.4GHz workstation. The performance comparison contains precision comparison and speed comparison. A real 3D input data volume that contains 186512 input traces is used as input. Each trace has 1500 samples with a 4ms sampling interval. The 3D output image cube contains 90 by 500 surface positions, by about 1500 pseudo-depth points per output trace.

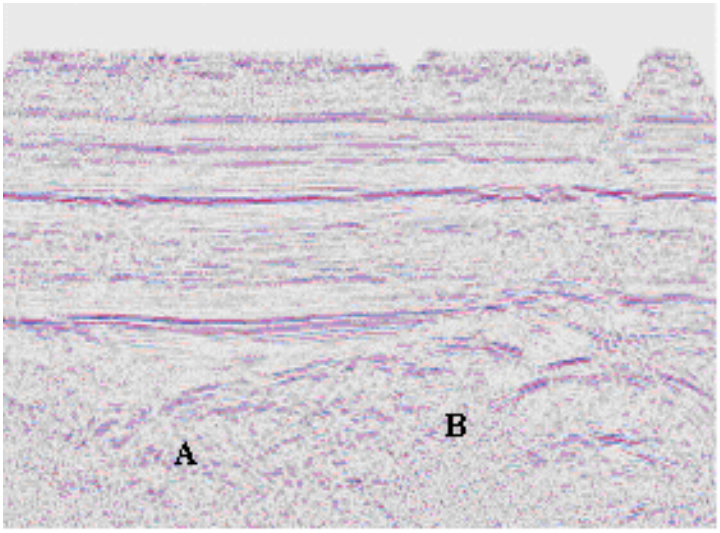

Figure 12. A Vertical In-Line Unmigrated Section

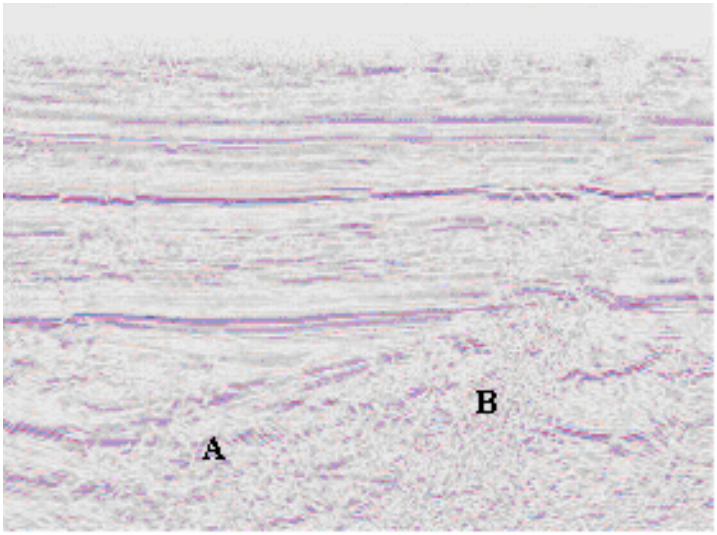

Figure 13. The Vertical In-Line Migrated Section

Figure 12 shows the image of a vertical in-line section selected from the stacked input data. Figure 13 is the migrated image for the same output section created by a simulation program, which imitates the same operations and precision as the FPGA-based hardware design. This migrated image is nearly the same as the result produced by

the pure software version of the PSTM algorithm running on the referential workstation. Notice that migration provides people with a clearer and more reliable underground image and facilitates detailed and easily recognizable subsurface structures. For example, the inclination of the reflection event A is increased in Figure 13; the vague event B in Figure 12 is clarified and turned into a syncline at the same position in Figure 13.

Define an elementary computation as all the calculations required for each input-output point pair to calculate the two-way travel time, oblique factor, anti-aliasing filtering and output accumulation. The total number of all elementary computations for this data volume is about 644 billions taking the migration aperture and the $T_{\max}$ limitation into consideration. The total execution time of the original program operating on the referential Intel workstation is 206570 seconds, in which more than 98% (202468 Seconds) are consumed by elementary computations and less than 2% (4102 Seconds) by others. In the following quantitative performance analysis, we use T_{CORE} as the elapsed time for all the elementary computations, T_{OTHER} as the time for other assistant works including initialization, data preparation, communication, etc. We have:

$$T_{SOFT} = T_{CORE} + T_{OTHER} \tag{4.7}$$

The proposed FPGA-specific PSTM computing engine accelerates only the elementary computations and leaves all other operations to the host machine. So the new total execution time will be:

$$T_{HARD} = T'_{CORE} + T_{OTHER} \tag{4.8}$$

According to Amdahl's Law, the overall speeding-up is:

$$Speedup = \frac{T_{SOFT}}{T_{HARD}} = \frac{1}{(1 - CoreFraction) + \frac{CoreFraction}{CoreSpeedup}} \tag{4.9}$$

Table 5 lists the performance comparison results for the designated task between the referential Intel workstation and the proposed FPGA-based approach with different configurations.

Table 5. Performance Comparison of PSTM on FPGA and PC

Configurations	Clock Frequency (Hz)	FPGA Resources Occupation	Kernel Code Speed (million/s)	Kernel Code Speedup	Overall Speedup
Intel P4 Workstation	2.4G	NA	3.2	1	1
One Computing Engine	50M	18.6	50	15.6	10.8
Two Computing Engine	50M	32.8	100	31.2	16.4
Four Computing Engine	50M	61.4	200	62.4	22

The following observations can be drawn from Table 5:

- The execution speed of a single FPGA-specific PSTM computing engine is 15.6 times faster than the speed of the referential Intel workstation. This impressive result is credited to the fully pipelined structure of the computing engine.
- The acceleration of the kernel subroutine of PSTM algorithm would increase linearly with the number of in-chip computing engines, but the overall acceleration is bounded by T_{OTHER}, which is constant for a designated processing task. A bigger task will increase the proportion of elementary computations and the overall acceleration will rise accordingly.
- The density (hardware resources) of an FPGA device will restrict the number of in-chip PSTM computing engines. On the other hand, computational performance of this algorithm could be further improved by integrating larger FPGA devices on board in the future.
- Memory bandwidth is another bottleneck, especially when more PSTM computing engines are integrated into one FPGA chip. Employing faster memory modules (for example, DDR400) or more dedicated memory controllers partially alleviates this problem.

- This comparison doesn’t take into consideration the speed degradation caused by pipeline stalls. In the hardware-based implementation of the PSTM algorithm, switching to next input/output traces would lead to control hazard, which is similar to branching stall of a pipelined commodity CPU. This control hazard is hard to predict because of the changing migration aperture, and so cannot be avoided. Theoretically, simply flashing a pipeline will cause at most 10% of performance degradation taking into consideration the gap between the number of pseudo-depth points per output trace and the number of pipeline stages in the computing engine.

5. FDM ON FPGA-ENHANCED COMPUTER PLATFORM *

In this section, we will introduce our work on accelerating Finite Difference Methods (FDM) on the proposed FPGA-enhanced computer platform. FDM is one of the oldest, but the most popular, numerical methods for solving various scientific & engineering problems governed by ODEs or PDEs. Although extremely computationally intensive, this class of methods is always a user's first choice because of its simplicity and robustness. Furthermore, such methods have the capability for dealing with complex geological models, which in general could not be handled effectively by Fourier transformation or other approximation methods. In the past decade, FD-based numerical modeling efforts for transient wave propagation problems in computational acoustics, computational electromagnetics, or geophysics fields have grown rapidly with performance improvements in commodity computers and parallel computing environments. Various software techniques, from high-level parallelism on PC-Cluster system to low-level memory and disk optimization, or even instruction-reordering, have been developed to accelerate the execution of these simulation tasks. However, these procedures are still time-consuming, especially when the geometrical size of the computational domain is much larger than the wavelength of sources. Therefore, they cannot be used routinely, except in institutions able to afford the high cost of operating and maintaining high-performance computing facilities.

This section is organized as follows: In Section 5.1, the standard second-order and high-order FD schemes for acoustic wave equations are derived based on Taylor expansion, and their deficiencies in numerical accuracy and computational costs are analyzed in detail. In Section 5.2, after a brief review regarding the state-of-art of solving linear wave modeling problems on FPGA-based systems, I present in detail our solutions to accelerate FDM on the proposed FPGA-enhanced computer platform. I first

---

* Reprinted with permission from "Optimized high-order finite difference wave equations modeling on reconfigurable computing platform" by C. He et al., 2007. Microprocessors and Microsystems, 31 103-115. Copyright 2007 by Elsevier.

introduce our design of the fully-pipelined FD computing engine and the sliding window-based buffering subsystem using the (2, 4) FD scheme as an example. Next, I extend this design to higher order schemes in time and in space to demonstrate its good scalability. For 3D cases, I propose the partial caching scheme utilizing external SRAM blocks as page buffer. The floating-point operation to memory-access ratio of FD schemes is analyzed and compared to emphasize its impact on achievable sustained computational performance of this implementation. Absorbing Boundary Condition (ABC) is one of the most troublesome parts of these modeling tasks, but is pivotal to the accuracy of final results. In this work, I adopt artificial damping layers to absorb and attenuate outgoing waves. This simple Damping Boundary Condition (DBC) scheme introduces only moderate additional workload and consumes limited hardware resources, so would be perfect for our high-order FD-base implementation.

Section 3 provides the performance comparisons between the new FD computing engine implemented on Xilinx ML401 FPGA evaluation platform and its pure software counterpart running on a P4 workstation. Conventionally, scientists in this field select to compare only the execution time of numerical experiences to show the superiority of their FPGA-based solutions over general-purpose computers. However, these comparisons did not take into account other cost factors such as system complexity, commonality, etc., so is more or less unfair for PC-based solutions. Here, the fairness of the performance comparison is emphasized. The aim is to facilitate results of this research work convincing for people who are familiar with coding on conventional software environment.

Standard floating-point arithmetic units are the main components of the resulting high-order FD computing engine and consume most in-chip programmable hardware resources. Sometimes, the computing engine for complex PDE problems may require tens, even hundreds, of such arithmetic units, which might be unfeasible for systems with limited FPGA resources. From Section 4, I introduce our efforts to address this problem with improved numerical methods/algorithms. I first present our design of FPGA-specific FD schemes using optimization methods. A heuristic algorithm is

proposed to adjust FD coefficients so that considerable hardware resources could be saved without deteriorating numerical error properties. In Section 5, I propose a group-alignment based summation algorithm to accumulate those floating-point products produced by coefficient multipliers in floating-point/fixed-point hybrid arithmetic. This hardware-based algorithm can result in similar, or much better, worst-case absolute and relative numerical errors as standard floating-point arithmetic with only a fraction of hardware resources consumed. Also the total number of pipeline stages required for the new FD computing engine could be reduced significantly.

5.1 The Standard Second Order and High Order FDMs

5.1.1 2nd-Order FD Schemes for Wave Equations in Second Derivative Form

Linear wave equations are in general represented in first derivative form. It is well known that they can also be written in second derivative form without losing generality [45]. Representing linear wave equations in second derivative form has no benefit for conventional Finite-Difference Time-Domain (FDTD) algorithms executed on general-purpose computers. However, as we will see later in this section, it plays a key role in our FPGA-based solution to improve the efficiency of memory access.

Let's consider the simplest 2D scalar acoustic case in the form of second-order linear PDE, which relates the temporal and spatial derivatives of the vertical pressure field as follows,

$$\frac{\partial^2 P(x,z,t)}{\partial t^2} - \rho(x,z)v^2(x,z)\nabla \bullet \left(\frac{1}{\rho(x,z)} \nabla P(x,z,t) \right) = f(x,z,t) \qquad (5.1)$$

Where P is the time-variant scalar pressure field (pressure in vertical direction) excited by an energy impulse $f(x,z,t)$; $\rho(x,z)$ and $v(x,z)$ are the density and acoustic velocity of underground media, which are all known as input parameters for wave modeling (forwarding) problem.

Define the gradient of a scalar field S as: $\nabla S \equiv \frac{\partial S}{\partial x}\vec{x} + \frac{\partial S}{\partial z}\vec{z}$ and the divergence of a vector field $\vec{V}$ as, $\nabla \bullet \vec{V} \equiv \frac{\partial V_x}{\partial x} + \frac{\partial V_z}{\partial z}$, Equation (5.1) describes the propagation of acoustic waves inside 2D or 3D heterogeneous media with known physical properties. The numerical modeling problem we considered here is to simulate the time evolution of the scalar pressure field P at each discrete grid point in 2D or 3D space accurately. It is straightforward to extend the numerical methods and corresponding hardware implementation proposed here to other FD-based numerical simulations. For example, the classical 3D Maxwell's equations in computational electromagnetic problems can also be rewritten as three second-order wave equations in x, y, or z direction respectively with similar but more complex forms as Equation (5.1).

We assume underground media a constant density to further simplify Equation (5.1) as follows,

$$\frac{\partial^2 P(x,z,t)}{\partial t^2} - v^2(x,z)\Delta P(x,z,t) = f(x,z,t) \qquad (5.2)$$

Here, $\Delta \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$ stands for the Laplace operator. Notice that the vector field $\vec{V} = \nabla P(x,z,t)$ in Equation (5.1) disappears here and the input and output of this Laplace operator are all scalars. This new equation is still practical for 2D and 3D acoustic modeling problems and widely used in seismic data processing field.

FDM starts from discretizing this continuous equation into discrete finite-dimensional subspace in time and/or space. Given the values of variables on the set of discrete points, the derivatives in the original equation are then expressed as a linear combination of those values at neighboring points. Equation (5.2) is usually discretized on unstaggered grids, where the second-order spatial differential operators are approximated by the standard 2nd-order central FD stencil as follows,

$$P_{i,k}^{n+1} = 2 \cdot P_{i,k}^{n} - P_{i,k}^{n-1} + (dt)^2 \cdot v_{i,k}^2 \cdot \Delta^{(2)} P_{i,k}^{n} + f_{i,k}^{n} + O(dt^2) \qquad (5.3)$$

and

$$\Delta^{(2)}P_{i,k}^{n}=\left(\frac{P_{i+1,k}^{n}-2\cdot P_{i,k}^{n}+P_{i-1,k}^{n}}{(dx)^{2}}\right)+\left(\frac{P_{i,k+1}^{n}-2\cdot P_{i,k}^{n}+P_{i,,k-1}^{n}}{(dz)^{2}}\right)+O(dx^{2})+O(dz^{2}) \qquad (5.4)$$

Here we use $\Delta^{(2)}$ represent the 2nd-order accurate FD approximation of the Laplace operator. The subscripts in these equations mark the spatial positions of discrete pressure field values or parameters; superscripts mark the time points when pressures are evaluated. dx and dz define the spatial interval between two adjacent grids in x or z direction, respectively. dt stands for the time-evolution step.

Equation (5.3) shows us the second-order time-evolution scheme and equation (5.4) is the second-order FD scheme evaluating the spatial Laplace operator. Figure 14 depicts the corresponding FD stencil in 2D space. We also draw the 3D spatial stencil of $\Delta^{(2)}$ in figure 15. All grid points that are involved in calculation are marked out in these figures. We can observe in Figure 14 that six grid values together with one parameter value ($v_{i,k}$) are needed to evaluate 2D pressure field P at grid point (i,k) to a future time step. Five of those grid values come from the present pressure field at this spatial point and its four orthogonal neighbors; the last one is the pressure value at the same grid point but from previous time step.

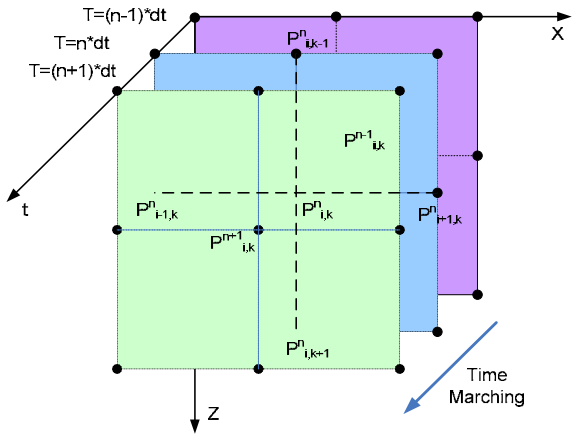

Figure 14. (2, 2) FD Stencil for the 2D Acoustic Equation

Starting from two known wave fields working as initial conditions, FD wave modeling tasks progress the evaluation of wave propagation grid point-by-grid point and time step-by-time step. Realistic seismic wave modeling problems may have thousands of discrete grid points along each spatial axis. So the total number of grid points in computational space could be in the millions for 2D cases or in the billions for 3D cases. The number of discrete time evolution steps is at least the same as the number of discrete spatial points along the longest axis according to Courant-Friedrichs-Lewy (CFL) stability condition [46]. Correspondingly, FD solutions of such time-dependent problems are in general computationally demanding as well as data intensive.

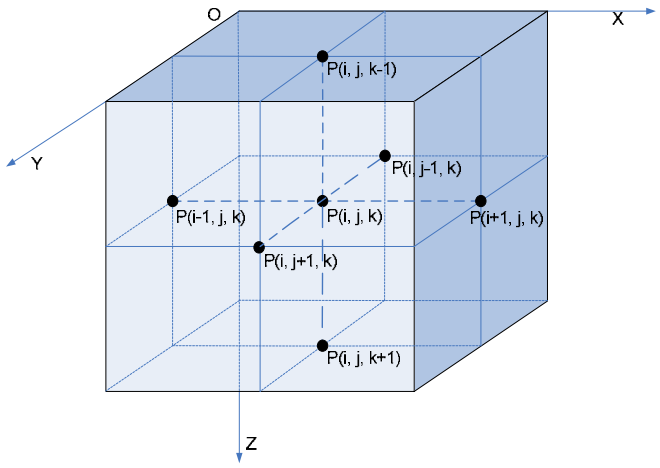

Figure 15. Second-Order FD Stencil for the 3D Laplace Operator

However, the extraordinary computational workload of FDM is not the entire story: finite difference approximations also introduce numerical truncation errors. Such errors arise from both the temporal and spatial discretizations and can be classified into numerical dispersion errors, dissipation errors, and anisotropy errors. Here, I omit tedious mathematical theories of numerical analysis but give the readers an intuitive explanation that numerical errors would cause the high frequency wave components propagating at slower speeds, damped amplitudes, or wrong directions in numerical

simulations than in the reality. These errors will accumulate gradually, finally destroy the original shape of wave sources after propagating over a long distance or time period. Here, we use the FD scheme shown in Equation (5.3) and (5.4) as an example, which is of second order accuracy with respect to time and space (a so-called (2, 2) FD scheme). To show the effects of spatial discretization only, we assume that the temporal derivative term can be approximated precisely by reducing time-evolution step (dt). If we select the spatial sampling interval to be 20 points per shortest wavelength, the simulation results obtained by this (2, 2) FD scheme are considered satisfactory only in moderate geological area, generally a computational domain on the order of 10 wavelengths [47]. For waves propagating over longer distances, the spatial interval required by this (2, 2) scheme should be further refined, leading to an enormous number of spatial grid points and time-evolution steps, impractical memory requirements, and unfeasible computational costs. This is the main motivation of the development of higher-order FD schemes. We have to point out that the famous Yee's FDTD method, which has been widely adopted for electromagnetic modeling problems, is also a (2, 2) FD scheme but for the first derivative Maxwell equations discretized on staggered spatial grids. So, it also suffers the same numerical errors we discussed above, although they are in general a little less serious.

5.1.2 High Order Spatial FD Approximations

We first consider spatial higher-order FD schemes and remain the second-order time-evolution stencil in equation (5.3) unchanged. Numerical derivative of a function defined on discrete points can be derived from Taylor expansion. The goal of the so-called maximum order FD schemes [48] is to attain accurate approximation by canceling as many the lower order terms in Taylor expansion formula as possible. The first un-cancelled Taylor series term determines the formal truncation error and the accuracy order of the corresponding finite difference scheme. For example, the one-dimensional Taylor expansion along x-axis at $x = (i \pm 1) \cdot dx$ for P is,

$$P(x_{i\pm1}) = P(x_i) \pm dx \cdot \frac{\partial P(x_i)}{\partial x} + \frac{(dx)^2}{2}\frac{\partial^2 P(x_i)}{\partial x^2} \pm \frac{(dx)^3}{6}\frac{\partial^3 P(x_i)}{\partial x^3} + \frac{(dx)^4}{24}\frac{\partial^4 P(x_i)}{\partial x^4} + \cdots \quad (5.5)$$

When we add these two equations together to eliminate odd derivative terms at the right hand side, we have:

$$\frac{\partial^2 P(x_i)}{\partial x^2} - \frac{P(x_{i-1}) - 2P(x_i) + P(x_{i+1})}{(dx)^2} = -\frac{(dx)^2}{12}\frac{\partial^4 P}{\partial x^4} + \mathrm{O}\left((dx)^4\right) \quad (5.6)$$

Equation (5.6) shows us that the difference (truncation error) between the second derivative of P and its FD approximation $\frac{P(x_{i-1}) - 2P(x_i) + P(x_{i+1})}{(dx)^2}$ is proportional to $(dx)^2$. That is where the name of (2, 2) FD scheme shown in Equation (5.3) and (5.4) originates. Applying the same idea to more discrete points along the x-axis, we can cancel out higher order truncation terms, so the resulting approximation to the second derivative operator would be more accurate in the sense of truncation errors. Systematically, we can approximate $\frac{\partial^2 P}{\partial x^2}$ to $(2m)th$ accurate order by linear combination of the values of P at $(2m+1)$ discrete grid points as follows,

$$\left(\frac{\partial^2 P(x_i)}{\partial x^2}\right)^{(2m)} = \frac{\alpha_0^m \cdot P_i + \sum_{r=1}^{m} \alpha_r^m \cdot (P_{i+r} + P_{i-r})}{(dx)^2} + O\left((dx)^{2m}\right) \quad (5.7)$$

where $\alpha_0^m = -2 \cdot \sum_{r=1}^{m} (-1)^{r-1} \frac{2m!^2}{r!(m-r)!(m+r)!}$ (5.8)

and $\alpha_r^m = (-1)^{r-1} \frac{2m!^2}{r!(m-r)!(m+r)!}$ (5.9)

which are all selected to maximize the order of the first un-cancelled truncation term.

Expanding the higher-order FD schemes to y- and z-axis is straightforward, so a class of $(2m)^{th}$-order FD approximation of the Laplace operator in 2D or 3D space can be obtained. Similar to the standard (2, 2) FD scheme, we draw in Figure 16 the FD stencils for (2, 4) FD scheme in 2D space. The 3D spatial stencil for $\Delta^{(4)}$ is also shown in Figure 17. We observe that more spatial grid values around the grid point (i, j, k) are required to evaluate the Laplacian value at the central position.

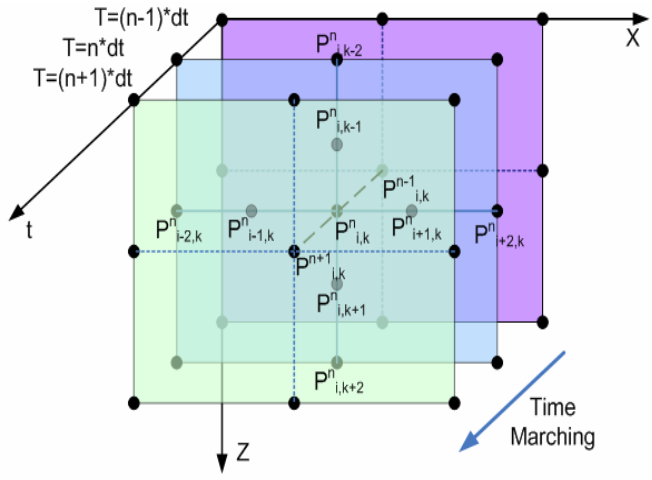

Figure 16. (2, 4) FD Stencil for the 2D Acoustic Equation

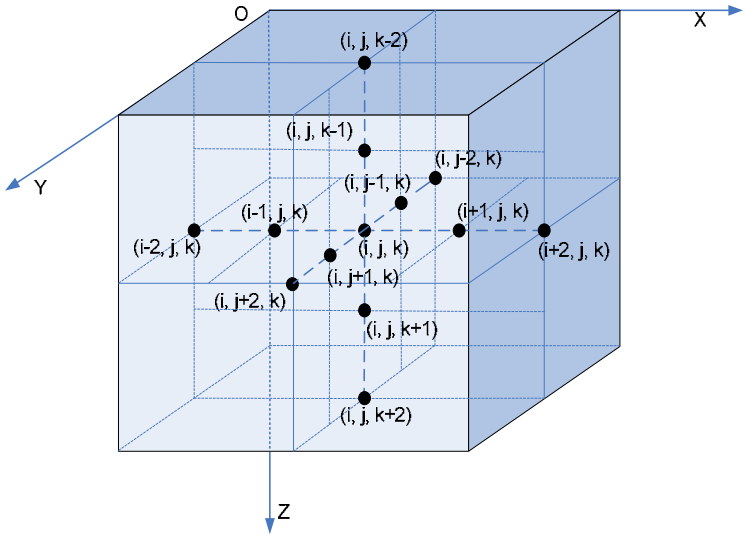

Figure 17. 4th-Order FD Stencil for the 3D Laplace Operator

Although the evaluations of Equation (5.7) are much more complex than Equation (5.4), higher-order FD schemes have higher-order un-cancelled truncation term, which

leads to much smaller approximation errors. This property can be clearly depicted by dispersion relations plotted in Figure 18, which is obtained by taking Fourier transform of the governing equation and its approximation in time and space. An intuitive criterion is that the dispersion relation of FD schemes should be close enough to the ideal wave equation. In other words, the dispersion error caused by numerical approximations should be kept as small as possible. From this figure, we can tell that higher-order schemes have less dispersion error for gradually larger wave-numbers, thereby leading to improved results. Put it another way, by using high-order FD schemes, we can enlarge the spatial sampling interval so that the number of grid points can be reduced without deteriorating accuracy criterion [49]. Figure 19 shows the dispersion error between the ideal wave equation and its approximations. We can easily draw the same conclusion from this figure as from Figure 18.

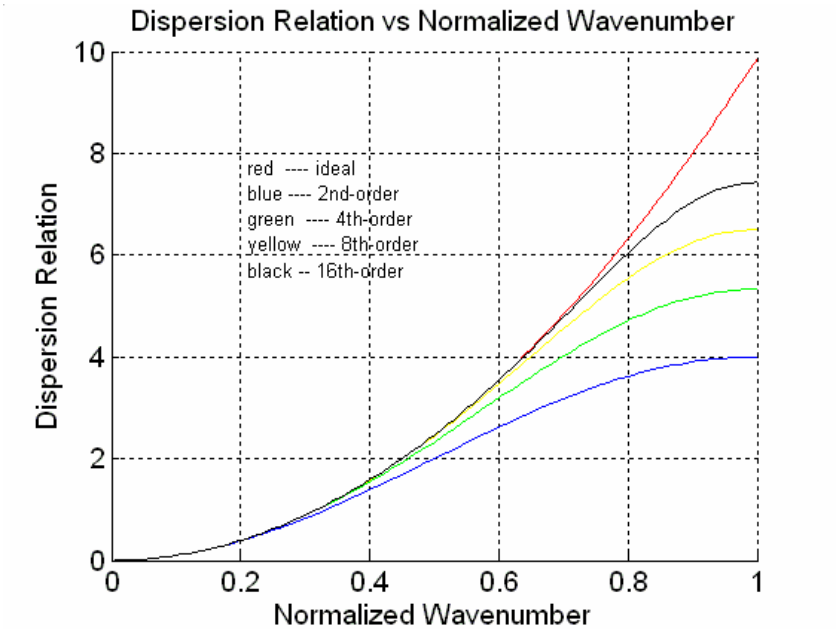

Figure 18. Dispersion Relations of the 1D Acoustic Wave Equation and Its FD Approximations

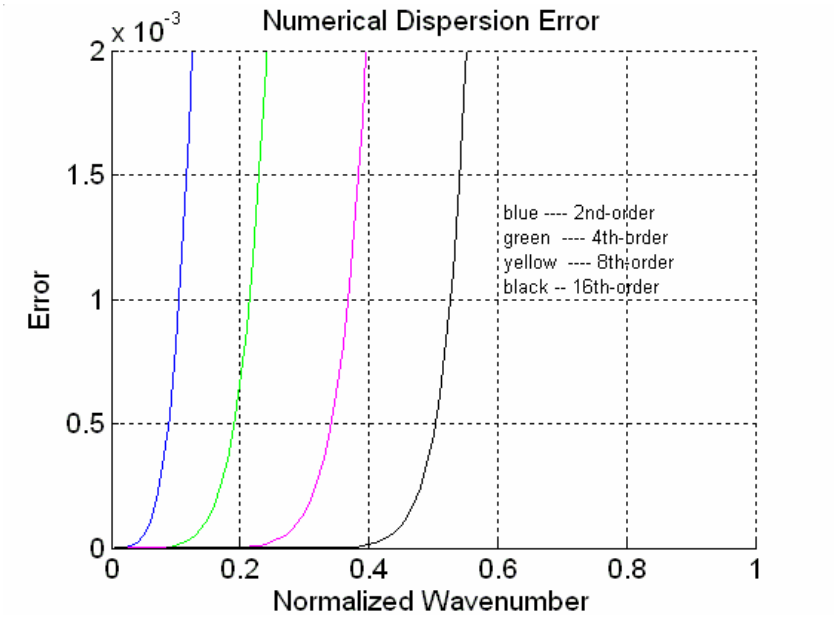

Figure 19. Dispersion Errors of Different FD Schemes

We designed a simple experiment to show the effectiveness of higher-order FD schemes. Here, we simulate the propagation of an exponentially-attenuated single-frequency sine wavelet in 1D homogenous media (constant velocity) along the x-axis. The time-evolution step is set small enough to attain negligible temporal truncation errors. We try to determine the spatial sampling interval where the power of numerical errors is reduced to be around 0.1 percent of the total energy of the original wavelet after it propagates a distance of 400 wavelengths. The simulation results are concluded in Table 6 for different FD schemes. We can observe that the (2, 16) FD scheme needs only 1600 spatial grids in our test (a propagation distance of 400 wavelengths times four points per wavelength), which is about five times less than (2, 4) scheme or over ten times less than the standard (2, 2) FD scheme. The reduction in the number of grid points will become much more significant if we apply higher order schemes to 2D or 3D cases. Please note that the propagation distance for the standard (2, 2) FD scheme is set to be 40 wavelengths because the FD scheme is incapable of simulating the wavelet

propagating for hundreds of wavelengths accurately with a reasonable spatial sampling interval.

Table 6. Performance Comparison for Different HD Schemes

FD schemes	Propagation Distance (Wavelength)	Grid Density (Grid/Wavelength)	Total Number of Grid Points	Relative Error Power
(2, 2)	40	40	1600	0.0024
(2, 4)	400	19	7600	0.0037
(2, 8)	400	7	2800	3.8e-4
(2, 16)	400	4	1600	0.0010

However, high-order FD schemes are ineffective for abrupt discontinuous media, so people tend to be conservative in enlarging spatial sampling interval. The result is that the decrement of spatial points achieved by high-order FD schemes in general is not enough to compensate for the additional computations they introduced. That explains why high-order schemes are always more computationally expensive than the standard second order schemes and why they are seldom utilized in reality.

5.1.3 High Order Time Integration Scheme

People also hope to enlarge the time-evolution step by adopting high-order time integration schemes so that the numerical simulations could be more accurate. Unfortunately, it has been proven that any Taylor-expansion based higher-order approximation to the second derivative in time in equation (5.2) leads to unconditional unstable schemes [50]. An alternative is the modified wave equation introduced by Dablain in [49] as follows,

$$\frac{P^{n+1}-2P^{n}+P^{n-1}}{(dt)^2}=\frac{\partial^2 P}{\partial t^2}+\frac{(dt)^2}{12}\cdot\frac{\partial^4 P}{\partial t^4}+O(dt^4)=v^2\Delta^{(2m)}P+\frac{(dt)^2}{12}\cdot\frac{\partial^2}{\partial t^2}\left(v^2\Delta^{(2m-2)}P\right)$$
$$=v^2\Delta^{(2m)}P+\frac{(dt)^2}{12}\cdot\left(v^2\Delta^{(2m-2)}\left(v^2\Delta^{(2m-2)}P\right)\right) \qquad (5.10)$$

By applying the original wave equation (5.2) twice to its second-order FD approximation, the second-order temporal truncation error hidden in equation (5.3) is compensated by a higher-order spatial Laplacian term, which results in a class of (4, 2m) FD schemes. The coefficient $(dt)^2$ of the compensation term allows the accurate order of its FD approximation two less than the original Laplace operator. For example, Taylor expansion-based 4th-order accurate approximation to the right-hand-side of equation (5.10) leads to a 13-point FD stencil in space. This spatial stencil is shown in Figure 20 together with the 9-point stencil for (2, 4) FD scheme. As for computational cost, the modified wave equations almost double the number of floating-point operations for every time step because of the existence of the position-variant parameter $v(x, z)$.

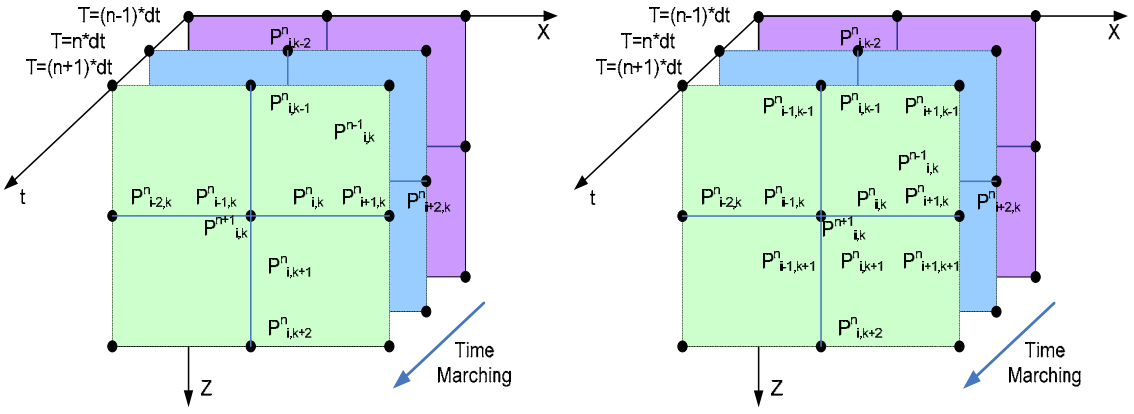

Figure 20. Stencils for (2-4) and (4-4) FD Schemes

Here, I also applied this new approach to the previous experiment to show its effectiveness. From Table 7, we observe that the time-marching step is enlarged greatly when we migrate to the 4th-order time-integration scheme. This impressive result is

partly attributed to the simple experiment we selected. As we mentioned above, the time evolution step is constrained by the Courant-Friedrichs-Lewy (CFL) stability condition, so is related to spatial sampling interval. For realistic wave modeling problems in abrupt discontinuous media, the progress in time step is not always good enough to remedy the additional computational costs it introduced, which is the same case that we encountered in spatial higher-order schemes.

Table 7. Performance Comparison for High-Order Time-Integration Schemes

FD schemes	Grid Density (Grid/Wavelength)	Number of Grid Points	Time-Marching Step	Relative Error Power
(2, 4)	19	7600	0.001	3.7e-3
(4, 4)	19	7600	0.008	3.0e-4
(4, 8)	7	2800	0.008	2.3e-3
(6, 8)	7	2800	0.02	5.4e-3

5.2 High Order FD Schemes on FPGA-Enhanced Computers

5.2.1 Previous Work and Their Common Pitfalls

Recently, as FPGA continues to grow in density, people start trying to accelerate FD-based numerical PDE problems on an FPGA-based hardware platform. Compared with pure software running on commodity computers or pure hardware-based ASIC devices, FPGA technology can provide people with a compromise between the best flexibility of software and the highest computational performance of fully-customized hardware. The idea of accelerating acoustic wave simulations using the fully-customized hardware system for geophysical applications can be traced back to the 1990s [51]. The first attempt to implement an FPGA-based stand-alone seismic data processing platform was described in [52]. For computational electromagnetics problems,

several authors proposed their FPGA-based solutions to accelerate the standard Yee's Finite-Difference Time-Domain (FDTD) algorithm from the early 1990s [53] [54]. Recent work in this field can be found in [55-58].

Although most recent efforts on this track reported impressive acceleration over contemporary general-purpose computers, we can observe that some common pitfalls exist in their FPGA-based system designs and performance comparisons. The first problem is that people still tend to build their application-specific FPGA-based hardware platforms, where the FPGA devices are simply used as an alternative to ASIC to reduce the high NRE costs. In these systems, hardware architecture and interconnection pattern are well-tailored for particular applications so that the computational potential of FPGA devices could be completely bring into play. However, system costs of such a fully-customized approach would still be much higher than commodity computers and the system flexibility would be poor.

"Toy" problems were commonly used as examples to demonstrate performance improvements of FPGA-based systems over commodity computers. Onboard FPGA resources and memory space are always abundant so that the scalability of FPGA-based hardware systems is usually left out of consideration. External memory bandwidth never imposes a performance bottleneck, which is not the case for most data intensive applications in reality. Small but fast onboard SRAM modules or internal RAM blocks were selected as working space for small problems, which made the FPGA-based solutions expensive or unrealistic for most real-life problems. Correspondingly, the resulting performance comparisons are more or less unfair for commodity computers.

Software algorithms are commonly migrated to an FPGA-based system directly without or with only limited modifications such as instruction rescheduling or arithmetic unit customization. We know that most existing software algorithms and numerical methods are well-tuned for commodity CPUs, so may not be ideal for FPGA-based systems. As we will see later in this section, we emphasize modifying or designing new numerical methods/algorithms specified for this new computing resources so that satisfactory accelerations could be expected.

The last problem exists in performance comparison between FPGA-based systems and commodity-CPU based general-purpose computers. Sometimes, the comparisons are made between one PC workstation and a complex FPGA-based system with multiple FPGA chips; sometimes naïve software implementation is used as reference without careful performance tuning. Such comparison results are unconvincing for people who are accustomed to working on commodity computers with conventional software coding environment.

5.2.2 Implementation of Fully-Pipelined Laplace Computing Engine

As we introduced in Section 2, the fundamental hindrance of simulating wave propagation problems numerically is the massive data volume along with the complex numerical algorithms. Specifically, memory bandwidth available between the computing engine (FPGA) and onboard memory modules has been proven a bottleneck preventing people taking full advantage of FPGA's computational potentials [32, 57, 58]. In this section, I try to alleviate this bottleneck by adopting high-order FD methods together with a fully-customized in-chip memory subsystem. Sustained high computational throughput would be achieved by effectively mapping all related computations into the proposed FPGA-enhanced computer system without changing memory bandwidth requirements. Also, those common pitfalls we just mentioned are taken into consideration to facilitate these new computing resources appropriate for realistic applications.

We select realistic seismic acoustic and elastic modeling problems as our target applications. These simulation tasks are conventionally solved by the standard second-order FD schemes in a parallel computing environment. To overcome numerical deficiencies of these low-order schemes, we resort to high-order FD schemes that are seldom being adopted in reality because of their enormous computational cost. Here, because of the adoption of large-scale FPGA devices, in which people could easily integrate tens to hundreds of standard floating-point arithmetic units, computational

power does not seem to be a serious problem. This fact encourages us to adopt higher order FD schemes for better numerical performance.

The implementation of high-order FD schemes with standard floating-point arithmetic units on FPGA is simple and straightforward. For example, Figure 21 depicts the block diagram and dataflow of a 2D 4th-order Laplacian computing engine with 15 pipelined floating-point arithmetic units based on Equation (5.7). We can easily observe the adder tree structure with embedded constant multipliers. All arithmetic units are pipelined internally to achieve high data throughput. It is convenient to extend this design to higher-order schemes with more arithmetic units and tree levels. For example, a 16th-order 2D Laplace operator can be easily constructed with 33 adders and 20 multipliers.

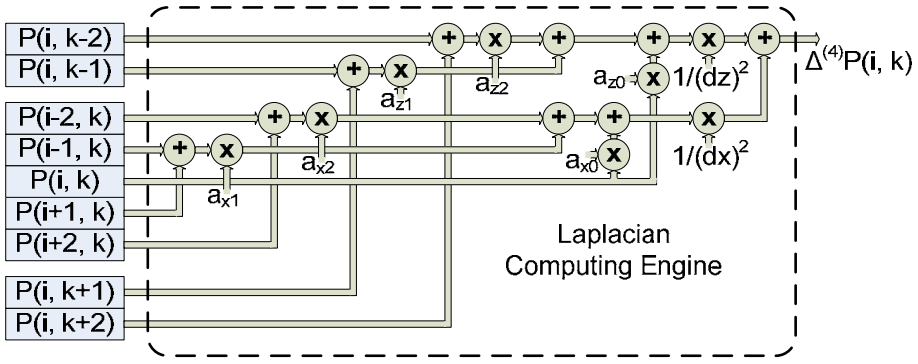

Figure 21. 2D 4th-Order Laplacian Computing Engine

5.2.3 Sliding Window Data Buffering System

However, in order to operate the large computing engine in its full speed to produce one output at each clock cycle, we need to feed it with a new set of operands at the same speed. This unfeasible data manipulation requirement forces people to integrate as many dedicated memory channels as possible onto their FPGA-based system. DDR-SDRAM can provide high bandwidth at relatively low price, so it becomes a prevailing

choice as large capacity onboard memory. The number of dedicated memory channels available on FPGA-based systems could be significantly more than, but still comparable to, commodity computers. For example, an up-to-date PC workstation has two DDR memory channels compared with four on the FPGA-based coprocessor platform presented in [58], which is the top record to our best knowledge. To complicate the situation, the bandwidth utilization is usually poor in practice due to the random-access nature of most applications. Previous designs in [56-58] tried to migrate the software version of Yee's FDTD algorithm directly into their customized FPGA-based platform. Their efforts concentrated on integrating more hardware arithmetic units into FPGA so that the aggressive computational speed of their designs would exceed commodity computers. This approach is very effective, with much higher acceleration than with contemporary PCs. However, the memory bandwidth bottleneck will finally be reached and after that, no more improvement will be obtained.

Consider first the standard second-order FD scheme. We rewrite Equation (5.3) and (5.4) here and ignore the source term,

$$P_{i,k}^{n+1} = 2 \cdot P_{i,k}^{n} - P_{i,k}^{n-1} + (dt)^2 \cdot v_{i,k}^2 \cdot \Delta^{(2)} P_{i,k}^{n} \tag{5.11}$$

$$\Delta^{(2)} P_{i,k}^{n} = \left(\frac{P_{i+1,k}^{n} - 2 \cdot P_{i,k}^{n} + P_{i-1,k}^{n}}{(dx)^2} + \frac{P_{i,k+1}^{n} - 2 \cdot P_{i,k}^{n} + P_{i,k-1}^{n}}{(dz)^2} \right) \tag{5.12}$$

We need one velocity and six pressure values in total to evaluate these two equations for one grid point at position (i,k). As for the computational costs, we have five additions and two multiplications to approximate the 2D Laplace operator; another one multiplication and two additions are needed to calculate the final result. (Three multiply-by-two operations are ignored because they can be easily merged into adjacent arithmetic units.) The ratio of floating-point computations to memory accesses is $10/7$, so its execution speed on commodity computers would be decided by main memory bandwidth but not the CPU's nominal speed.

Although higher-order schemes reduce the total number of grid points by complicating the computations at each grid point, they do require more operands in their computational stencils. Sequentially, the ratio of computations and memory accesses

remains virtually unchanged. Table 8 lists the number of floating-point operations and the number of operands required by different FD schemes to update the wave field at one spatial grid point for one time-evolution step. We note that the FP operations to operands ratio is always less than 2. This observation means that this class of methods is memory bandwidth bounded, which explains why only 20-30% of a commodity CPU's peak performance could be achieved when running on general-purpose computers. It is also the reason why high-order schemes result in only limited benefits and are, therefore, seldom put into practice in reality.

Table 8. Comparison of FP Operations and Operands for Different FD Schemes

FD Schemes	(2, 2)	(2, 4)	(2, 8)	(2, 16)
Total FP Operations	11	21	33	57
Total Operands	7	11	19	35

Here, we try to find appropriate on-chip memory structure by exploring data dependencies of the numerical algorithms so that the limited onboard memory bandwidth could be utilized more wisely. Define "row" as a line of spatial grids along the X-axis and "column" as a line of grids along Z-axis in 2D space. Because little optimization can be applied to equation (5.11) to reduce its computations or memory accesses, we consider equation (5.12) only. Our approach evaluates the 2D wave field grid by grid along the column. It can be visualized as moving a striped 2D operands mesh into the fixed computing engine via its input ports. Figure 22 depicts this idea, where only data dependencies along Z-axis are explored. If the evaluation of $P_{i,k}^{n+1}$ starts when the operand at grid point (i,k) reaches the center of the computing engine, we can notice that almost all pressure values we needed to calculate $P_{i,k}^{n+1}$ have been encountered/used except the value of $P_{i,k+1}^{n}$. This observation implies that if we could store some used grid values inside the FPGA chip temporarily, we may avoid accessing

the same data repeatedly from external memory, thereby saving a significant amount of memory bandwidth. This idea is reflected in the implementation in Figure 23, where values at grid points of a whole column $(:, k-1)$ are saved in the computing engine. Notice two input ports are needed here, one less than the previous case.

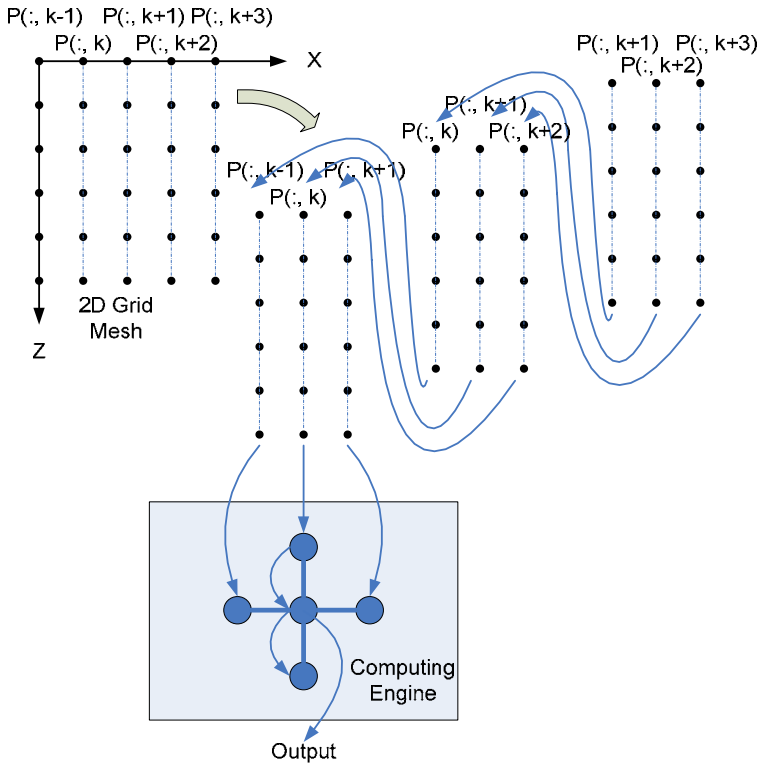

Figure 22. Stripped 2D Operands Entering the Computing Engine via Three Ports

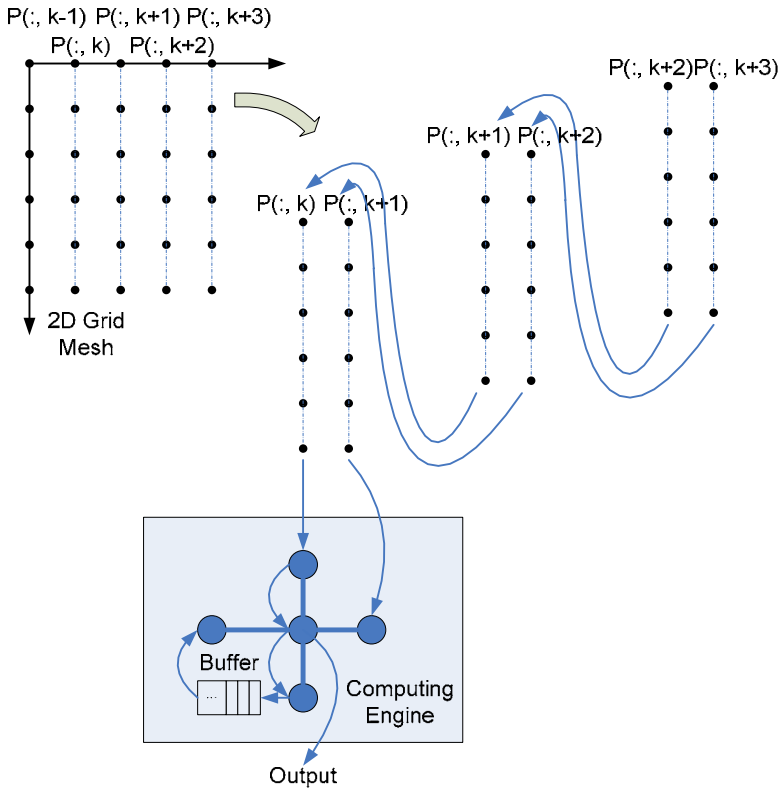

Figure 23. Stripped 2D Operands Entering the Computing Engine via Two Ports

This basic idea is almost the same as on-die data caches appearing in most modern commodity CPUs. However, the caching mechanism used in general-purpose computers is too complex and expensive to be implemented on FPGA-based hardware platform. Furthermore, it does not work well for our targeting numerical PDE problems because of their unfeasible data manipulation requirements. We need to design an efficient on-chip data-caching mechanism specified for the problem at hand. Figure 24 illustrates the block diagram of the data buffering system we designed for FD methods on FPGA-enhanced computer system.

In this design, we utilize two cascaded FIFOs as data buffers, each of which has the capacity to contain a whole row of discrete grid values. In general, the number of sampling points in each row is in the low thousands, so can be efficiently implemented with one or several SRAM blocks inside the FPGA device. Pressure values are fed into

the FPGA chip from one input port at the bottle of the first FIFO and discarded at the top of the last one. We delay the calculation of $P_{i,k}^{n+1}$ a whole row until the grid value $P_{i,k+1}^{n}$ enters our data buffer so that all operands we need to evaluate equation (5.12) are available inside FPGA chip. Taking into consideration the grid value $P_{i,k}^{n-1}$ at the previous time step, the parameter $v_{i,k}$ that is needed for calculating equation (5.11), and also the inevitable save back operation, we need only four memory accesses to update wave field values at one grid point for one time-marching step. In theory, this is the best result that can be achieved by a data caching system. We also introduce simple input caching circuits after SDRAM modules to hide their irregular data-accessing pattern. Consequently, input data can be fed into the buffering structure at a constant speed and the computing engine can be fully pipelined to achieve high computational throughput. We will revisit this input cache design in Section 5.3.

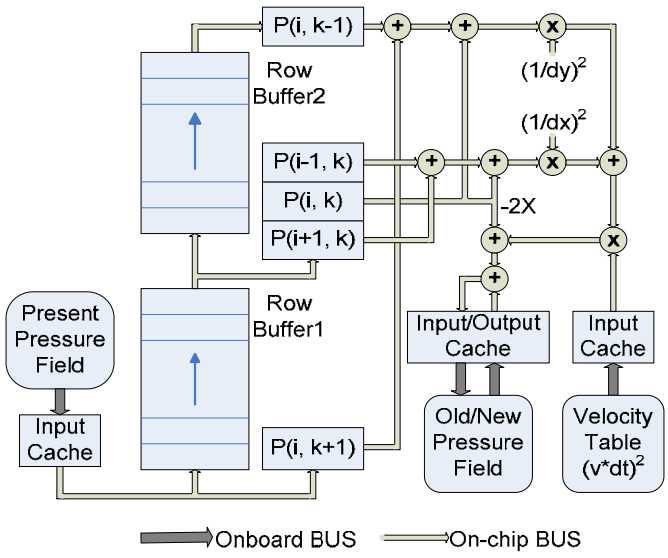

Figure 24. Block Diagram of the Buffering System for 2D (2, 2) FD Scheme

The principle of this data buffering system can also be abstracted into a sliding window as we demonstrated in Figure 25. Here, I use (2, 4) FD scheme in 2D space to show the good scalability of this design. We rewrite the equations here with the source term ignored,

$$P_{i,k}^{n+1} = 2 \cdot P_{i,k}^{n} - P_{i,k}^{n-1} + (dt)^2 \cdot v_{i,k}^2 \cdot \Delta^{(4)} P_{i,k}^{n} \tag{5.13}$$

$$\Delta^{(4)} P_{i,k}^{n} = \left(\frac{\partial^2 P}{\partial x^2} + \frac{\partial^2 P}{\partial z^2} \right)\Bigg|_{i,k}^{n}$$

$$= \frac{-\frac{1}{12} \cdot P_{i+2,k}^{n} + \frac{4}{3} \cdot P_{i+1,k}^{n} - \frac{5}{2} \cdot P_{i,k}^{n} + \frac{4}{3} \cdot P_{i-1,k}^{n} - \frac{1}{12} \cdot P_{i-2,k}^{n}}{(dx)^2} \tag{5.14}$$

$$+ \frac{-\frac{1}{12} \cdot P_{i,k+2}^{n} + \frac{4}{3} \cdot P_{i,k+1}^{n} - \frac{5}{2} \cdot P_{i,k}^{n} + \frac{4}{3} \cdot P_{i,k-1}^{n} - \frac{1}{12} \cdot P_{i,k-2}^{n}}{(dz)^2}$$

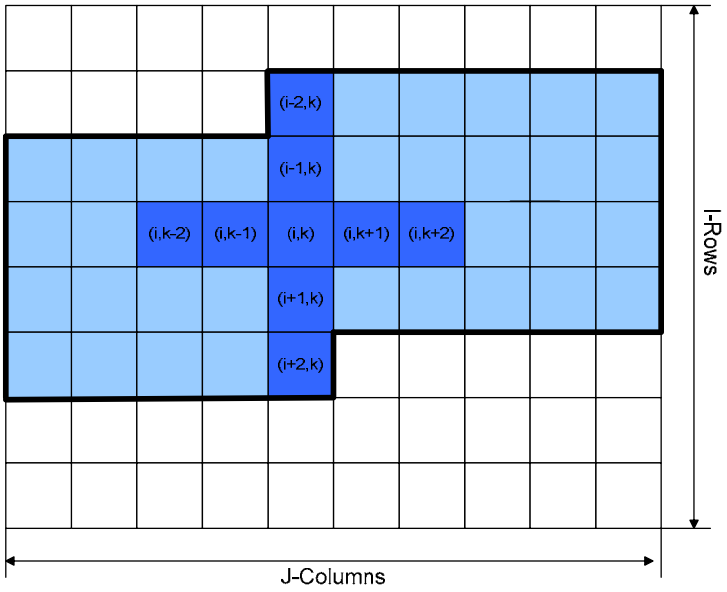

Figure 25. Sliding Window for 2D (2, 4) FD Scheme

Suppose we have in total I rows and J columns of spatial grids in 2D computational domain. The memory buffering system can be imagined as moving a siding window (grids enclosed by the bold line in Figure 25) that buffers $(4*J+1)$ continuous grids inside the 2D mesh. In addition, there are nine active points inside this window, whose values can be sent to the computing engine simultaneously via internal data paths. By connecting the input port of the sliding window with external memory, only one external read is needed to move the sliding window one grid right.

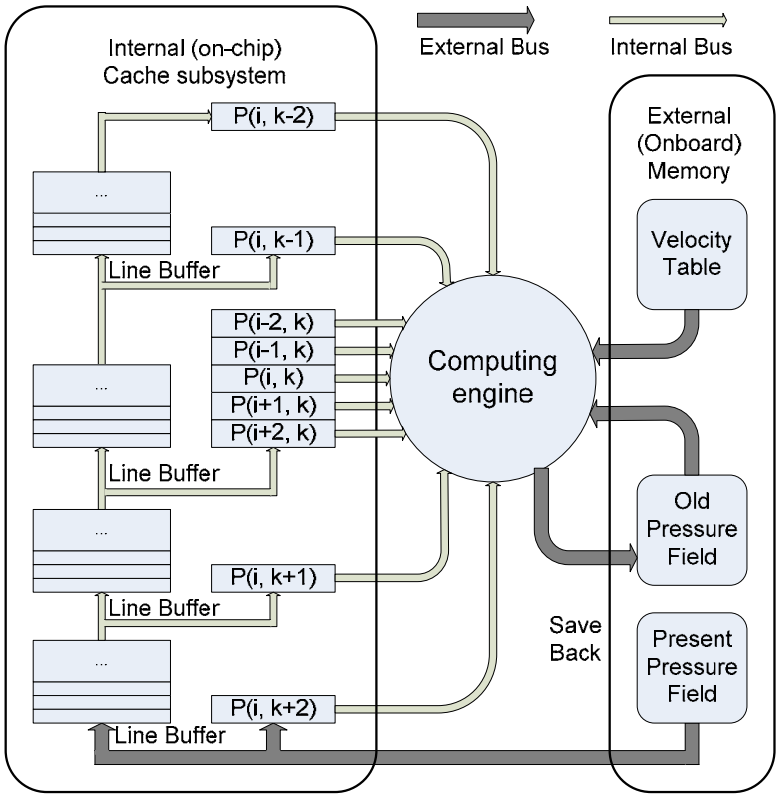

Figure 26. Function Blocks of the 2D (2, 4) FD Scheme

Figure 26 illustrates the block diagram and dataflow of the sliding window buffering system for the (2, 4) FD scheme. We use four cascaded FIFOs as our data buffer, each of which has the capacity to save a whole row of grid values. We delay the

evaluation of $P_{i,k}^{n+1}$ until the grid value $P_{i+2,k}^{n}$ enters our buffering structure from onboard memory so that all the operands we need are available to the computing engine. Taking memory accesses for grid value $P_{i,k}^{n-1}$, parameter $v_{i,k}$, and the save back operation into account, we still need on more than four memory accesses to evaluate one time-evolution step at one grid point.

The extension of figure 26 to (2, 2m) FD schemes is simple and straightforward. Now (2m) cascaded FIFOs are needed to build the sliding window. Correspondingly, we have to delay the calculation of $P_{i,k}^{n+1}$ for m rows to ensure all necessary operands appearing at correct positions in the buffering circuit. Inevitably, some arithmetic units should be inserted into the Laplacian computing engine and additional pipeline stages might be needed to guarantee the sustained computational throughput. In Table 9, we extend table 8 with two additional rows showing the number of external memory accesses and floating-point operation to memory access ratio for different FD schemes.

Table 9. Comparison of Caching Performance for Different FD Schemes

FD Schemes	(2, 2)	(2, 4)	(2, 8)	(2, 16)
Total FP Operations	10	20	32	56
Total Operands	7	11	19	35
External memory accesses	4	4	4	4
FP operations to memory accesses ratio	2.5	5	8	14

The most exciting observation is that although the memory bandwidth required by FD methods running on commodity computers increase linearly with the accuracy order, this requirement remains unchanged for our design on FPGA-enhanced computers, i.e., the number of memory accesses to evaluate one time-evolution step at one grid point is

always kept at four for different FD schemes if the data-buffering system could be implemented in-core. Correspondingly, the floating-point operations to memory accesses ratio continues to increase with the order of FD schemes, which implies improved data reusability. The only costs we pay for higher-order accuracy are on-chip memory blocks and conventional addition or multiplication arithmetic units, which are all abundant inside an up-to-date high density FPGA device. This result encourages us to adopt extraordinarily higher-order FD schemes in our design to further enlarge the spatial sampling interval until we reach the extreme at two samples per shortest wavelength, which is bounded by the Nyquist-Shannon sampling theorem.

5.2.4 Data Buffering for High Order Time Integration Schemes

Consider the modified wave equation we derived in Section 5.1.3: it is straightforward to allow us to extend our previous design of (2, 4) FD scheme to (4, 4) FD scheme based on its 13-point stencil shown in Figure 20. Unfortunately, this simple extension is only practicable for homogeneous media. (v is a constant inside the computational domain.) For inhomogeneous cases, because the coefficient varies in space, this approach would lead to additional computations and much more complex hardware architecture.

Rewriting equation (5.10) in the following form leads to a two-step scheme without degrading accurate order,

$$\frac{P^{n+1} - 2P^{n} + P^{n-1}}{(dt)^2} = v^2 \Delta^{(4)} \left(P^{n} + \frac{(dt)^2}{12} \cdot \left(v^2 \Delta^{(2)} P^{n} \right) \right) \tag{5.15}$$

Based on this expression, two Laplacian computing engines could be employed together with dedicated data buffering circuits to evaluate its right-hand-side: the first one is used to evaluate the Laplacian of $P_{i,k}^{n}$ to 2nd-order accuracy. Next, the compensated pressure field is fed into the second 4th-order accurate Laplacian unit to finish the calculation of $P_{i,k}^{n+1}$. Figure 27 illustrates the corresponding block diagram and dataflow design. Notice that another velocity buffering circuit is required here.

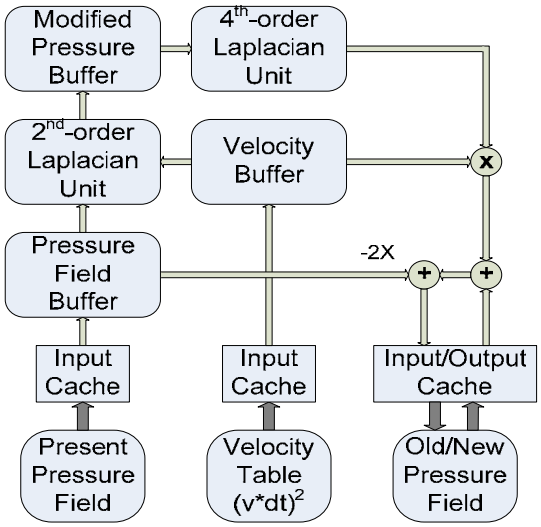

Figure 27. Block Diagram and Dataflow for 2D (4, 4) FD Scheme

5.2.5 Data Buffering for 3D Wave Modeling Problems

Extending the sliding window buffering system to 3D space is also straightforward, but the hardware implementation will become less efficient than 2D cases. Now, we need several large-capacity FIFOs to buffer 2D pages instead of 1D grid lines. Practical 3D wave simulations in general contain hundreds to thousands of grid points along each spatial axis. Correspondingly, the capacity of page buffers could easily reach several millions of words, which is almost approaching the maximum capacity of internal block memory inside an up-to-date FPGA chip. We now have to sacrifice some onboard memory bandwidth to meet the buffering requirements. Correspondingly, the caching behavior would not be optimal in these cases.

Fortunately, as we proposed in Section 3, there are multiple high-speed low-latency SRAM modules integrated on the FPGA-enhanced computer platform, which can be easily customized as page buffers with simple control logics. Thanks to the excellent scalability of high-order FD schemes, we can even adopt different accuracy order and different sampling interval for different spatial axis to accommodate the

specific hardware architecture of the platform. For example, if there are two onboard SRAM modules, each of which has enough capacity to contain one 2D page, we can select the standard 2nd-order FD scheme in Z-direction with fine sampling interval. This arrangement requires only two X-Y page buffers at runtime and doesn't affect the choice of FD schemes in the other two directions. Their orders can still be customized freely according to available hardware resources inside the onboard FPGA device. Figure 28 depicts the block diagram of the hybrid (2, 4-4-2) FD scheme.

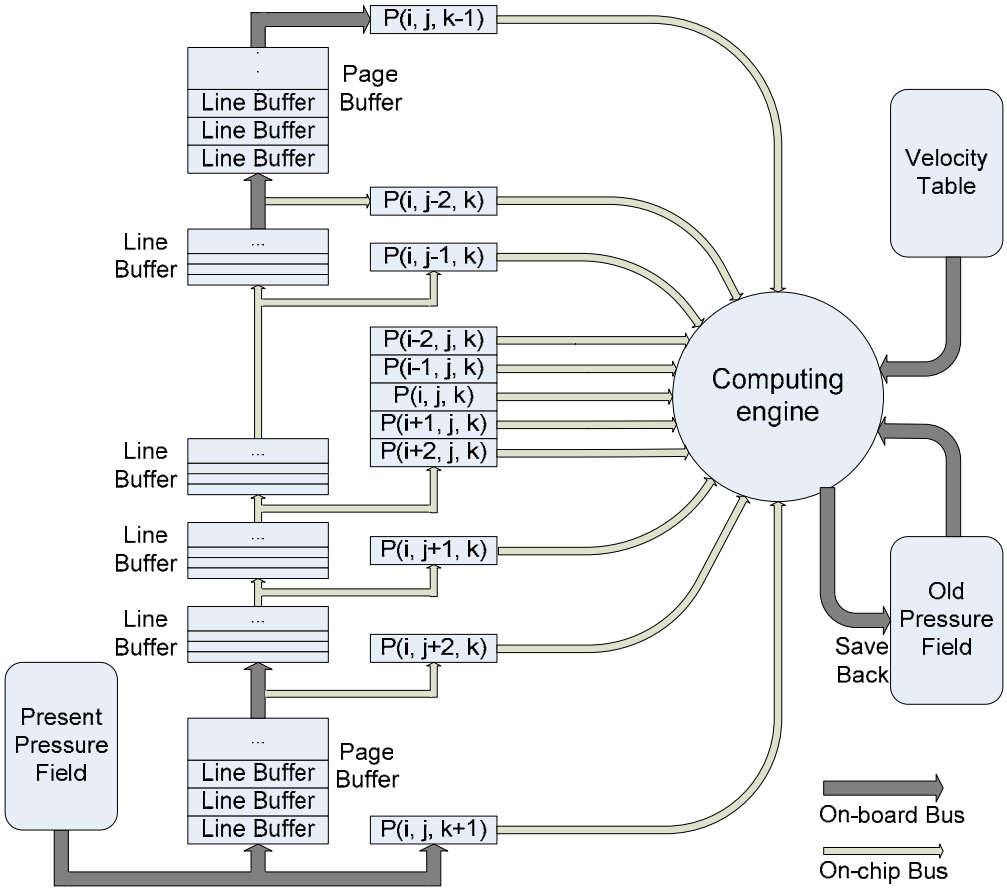

Figure 28. Function Blocks of the Hybrid 3D (2, 4-4-2) FD Schemes

5.2.6 Extension to Elastic Wave Modeling Problems

We now extend this design to elastic wave modeling problems. The 3D linear elastic wave equation can be represented as 9 first-order PDEs in Cartesian coordinates as follows [59],

$$\rho \partial_t v_i = \partial_i \sigma_{ii} + \partial_j \sigma_{ij} + \partial_k \sigma_{ik} \tag{5.16a}$$

$$\partial_t \sigma_{ii} = \lambda \left(\partial_i v_i + \partial_j v_j + \partial_j v_j \right) + 2\mu \partial_i v_i \tag{5.16b}$$

$$\partial_t \sigma_{ij} = \mu \left(\partial_i v_j + \partial_j v_i \right) \tag{5.16c}$$

Where λ and μ are Lame parameters, ρ is density, $v_{i,j,k}$ are particle velocities, σ_{ij} are stress tensor components, and we have $\sigma_{ij} = \sigma_{ji}$ for isotropic media. Staggered descretization for Equations (5.16) is always considered beneficial because all first-order finite difference stencils are naturally centered around relevant unknowns spatially and temporally, which leads to less numerical dispersion and efficient 2nd-order in-place time-evolution scheme. There are in total, 12 floating-point values at each discrete grid point: 3 media parameters, 3 particle velocities, and 6 independent stress tensor components. At the beginning of each time-evolution step (which is divided into two half steps), all 12of these floating-point values are read out from external memory. After FD computations, 9 updated wave field variables should be re-written. As for the computational cost, (2, 2) staggered FD scheme requires more than 90 floating-point operations at each grid point, and the number for the staggered (2, 4) FD scheme is about 140. This large number partly explains why only low-order FD schemes could be adopted for elastic wave simulations in reality. However, we should note that the underlying FD-based numerical algorithms for elastic modeling problems do not significantly differ from acoustic cases. So, all of the basic ideas for FD computing engine and sliding window buffering structure we proposed before are still practicable.

It is possible to design a large computing engine with hundreds of FP arithmetic units, although this approach may reach the capacity extreme, which one of the largest FPGA could support. An alternative is to divide this large computing engine into two

smaller ones for Equation (5.16a) and equation (5.16b, c), respectively. Thanks to FPGA's run-time reconfigurability, the host machine can reconfigure FPGA resources in seconds so that those two smaller computing engine can work alternately, each answers for half of one time-marching step.

Now, the basic idea of FD methods on FPGA-enhanced computers is clear: An efficient on-chip sliding window data-buffering circuit is placed between the FD computing engine and onboard memory modules using internal RAM blocks as the core component. By exploring data dependency properties of high-order FD schemes as well as the specific wave modeling problems, this data buffering system is capable of providing a new set of input operands to the FD computing engine at every clock cycle. Although operand stencils of high-order schemes are much wider than the standard second-order method, the number of external memory accesses to evaluate one time-evolution step at one grid point is kept unchanged. In other words, the onboard FPGA device effectively absorbs all additional computations introduced by high-order accuracy without speed penalty.

Because the clock rate applied to the FD computing engine and external memory modules is within the same range at hundreds of millions Hz, the bandwidth of onboard memory would be saturated rapidly and considerable FPGA resources would be wasted. Considering the (2, 2) FD scheme we proposed in Section 5.2.3, the computing engine for this simplest case consists of ten floating-point arithmetic units (seven additions and three multiplications), which cost only a small portion of hardware resources even for the fastest fully-pipelined implementation. By choosing high-order FD schemes, we can always complicate the computations as necessary to increase the floating-point computations to memory accesses ratio and alter the performance bottleneck back to the computing engine. In other words, by adopting appropriate high-order FD schemes, we can always find a point at which the utilization of the FPGA resources and onboard memory bandwidth are well balanced. Moreover, high-order FD schemes allow larger sampling intervals so that the total number of spatial grid points is considerably reduced.

Consequently, memory bandwidth requirements for the same problem decrease in an indirect way.

5.2.7 Damping Boundary Conditions

Seismic modeling problems are naturally posed in unbounded spatial domain, which of course are infeasible for digital computer based numerical simulations. A conventional approach is to surround the truncated computational domain with artificial layers, in which non-physical Absorbing Boundary Conditions (ABC) are applied. ABCs are designed to absorb or damp outgoing waves and ensure the consistency of the simulation results with respect to the solution of the original problem. Although it is not the kernel portion of numerical methods, ABCs always plays a pivotal role in numerical modeling tasks: a bad ABC would contaminate the entire simulation result gradually, and finally ruin any accuracy we have already gained. Ideal ABCs provide excellent attenuation property, but require only a few artificial layers to minimize additional computational costs. Unfortunately, these two aspects are contradictory to each other. Specifically for our wave field simulation tasks, an effective ABC scheme could easily introduce over 20 percent of additional workload. Previous FPGA-based implementations chose to ignore this troublesome problem deliberately [58] or simply migrate the software version without modification [57]. The later solution inevitably led to a complicated implementation and consumed considerable hardware resources.

One-way wave equations-based ABCs such as Clayton [60], Mur's [61], and Hidgon's [62] can only effectively absorb waves with small incident angle, and so are unsuitable for high accuracy simulations. By adopting variable splitting technology, Perfectly Matched Layer (PML) [63] can provide nearly perfect attenuation with only a few artificial layers, and so becomes more and more popular in computational electromagnetics and acoustic/elastic wave modeling fields since the 1990s. Despite of its excellent effects, PML introduce non-physical variables in its absorbing boundary layers, thereby leading to many more computations than the first approach. One common

problem for these two classes of ABCs is that the FD stencils they introduced inside absorbing boundary layers have different forms for different layers, or even different positions. Although not a problem for general-purpose computers, these ABCs would lead to complicated hardware implementation on FPGAs. Another problem is their high-order variations are either too complex to construct efficiently or inconsistent with internal high-order FD schemes.

Damping Boundary Conditions (DBC) was first introduced by Orlianski as sponge layer [64] in 1977. Although its performance was improved significantly later in the 1980s by Cerjan [65] and Burns [66], they still do not attract attention because the required thickness of damping layers may easily exceed hundreds when used with conventional low order FD schemes, and so are extremely inefficient in computational cost. The situation is significantly changed once we combine DBC with high-order FD schemes. Here, we select to modify the original wave equation by introducing a simple exponentially damping term. Consequently, a unified equation that governs both the truncated computational domain and absorbing boundary layers is obtained. This artificial damping term could be imagined as another media-related parameter to imitate natural wave attenuations caused by media's viscosity. Its value is set to be zero inside the original physical domain, but in damping layers, the attenuation intensity is enhanced exponentially from inner to outer.

Compared with those complex boundary equations introduced by other ABCs, this single equation approach is extremely attractive for our FPGA-based solution because of its simplicity and consistency: only one additional FP multiplier and a short damping coefficients table are required in its hardware implementation. Moreover, high-order internal FD schemes now are in the same form as absorbing boundary layers except the damping coefficients, which can be simply assumed as another media-related parameter. In other words, the modified wave equation has a physical correspondence. As we introduced before, high-order FD schemes are capable of enlarging spatial sampling interval without deteriorating numerical accuracy. Consequently, the damping layers with the same physical thickness as before now contain much less grid points. Our

numerical tests show that 20~40 damping layers are enough to provide satisfactory damping results for high-order schemes, results in only less than 5~10 percent of additional computations for realistic simulation tasks.

5.3 Numerical Simulation Results

In this section, we use two examples to show the correctness and effectiveness of the PFGA-based high-order FD methods for seismic acoustic wave modeling problems. Table 10 shows their computational workload and storage requirement. These two problems are selected carefully to ensure that they can be fit into our hardware development platform. The target FPGA-based prototyping platform we used in this work is an entry-level Xilinx ML401 Virtex-4 evaluation board [67]. Although this board provides only limited onboard hardware resources (one XC4LX25 FPGA chip embedding 24,192 Logic Cells, 48 DSP Slices and 72 18-kb SRAM Blocks; 64MB onboard DDR-SDRAM modules with 32-bit interface to the FPGA chip; and 9Mb onboard ZBT-SRAM with 32-bit interface.), it does contain all necessary components we need to validate our design. The development environments are Xilinx ISE 6.3i and ModelSim 6.0 se. The simulation results are compared with their software counterparts running on an Intel P4 3.0 GHz workstation with 1GB memory. The referential C program is compiled on Linux OS using Intel C++ v8.1 with optimization for speed (-O3 –tpp7 and –xK). About 20 percent of CPU's peak performance is achieved.

The implementation of the FD computing engine is based on the block diagrams presented in Figure 26. As we show in Table 10, there are over one million discrete grids for each problem. Onboard DDR-SDRAM spaces are assigned as working space. These storage units are organized as four arrays to save the previous pressure field, the present pressure field, the unknown future pressure field, and the velocity table, respectively. In order to utilize the bandwidth of external DDR-SDRAM more efficiently, an aditional cache circuit is constructed for each data array using two on-chip RAM Blocks at the interface between SDRAM modules and internal data-buffering subsystem. This input

cache contains two parallel-working 512-word data buffers, each of which can accept a whole physical column of data values from SDRAM. They operate in swapping manner to hide the irregular data accessing and periodic refreshing behavior of SDRAM components. This implementation isolates the data-buffering system and the computing engine from the memory interface circuits. Correspondingly, a constant high-speed computational throughput could be achieved.

Table 10. Size of Wave Modeling Test Problems

	2D Constant Media	2D Marmousi Model
Number of Spatial Grids	1100×1100	2340×770
Total Time Steps	6000	8250
Storage Requirements	4 Million Words	4 Million Words
Number of Grid Computations	7.26×10^9	1.49×10^{10}

5.3.1 Wave Popagation Test in Constant Media

The first example is designed to show the computational performance of our FPGA-based FD computing engine compared with the referential PC workstation. It is a simple 2D seismic modeling task in constant velocity media with 1000×1000 spatial grids and 6000 time-integration steps, which, in total, leads to 6×10^9 grid computations. DBC is applied to 50 outer layers on all four boundaries to achieve a nearly-perfect absorbing result, which increases the total number of grid computations to 7.26×10^9. In other words, the introduction of DBC leads to a 20 percent additional workload. We

purposely remain the total number of spatial grids fixed for different FD schemes to evaluate the acceleration attributed purely to our hardware implementation.

Utilizing the onboard 100MHz oscillator, we simply set the clock rate applied to onboard DDR-SDRAM modules at 100MHz and to the computing engine at 50MHz. (The maximum clock frequency for the DDR-SDRAM modules on the ML401 platform is 133MHz. So, the theoretical maximum computational throughput is 66 million grids per second.) Compared with the fifty million grid-per-second peak computational throughput, the speed of our implementation is degraded for less than 2 percent because of pipeline stalls. These stalls occur mainly when we swap input/output caches and when we flush the cascaded data FIFOs at the beginning of each time-marching step.

We modified the single-precision floating-point adder and multiplier proposed in [68] as our arithmetic units. The floating-point multiplier unit is constructed as a constant multiplier to save on-chip DSP slices. The floating-point adder is also redesigned by ignoring some impossible exceptions. Our simulation results of the software and hardware implementations for different FD schemes are shown in Table 10. The choice of FD schemes is restricted by available in-chip programmable hardware resources on the entry-level evaluation board. We must admit that the performances of referential software program might be further improved by low-level tuning and optimizations. However, this approach is so experience-intensive that only specialists could benefit from it [69].

The results shown in Table 11 are satisfactory considering the entry-level evaluation board we used in this test. The most exciting observation in this table is that the aggregate FP performance of the FPGA-based implementation keeps increasing with the order of FD schemes so that a nearly constant grid pressure updating rate is maintained. Comparatively, the sustained FP performances for high-order FD schemes on commodity computers are reduced significantly due mainly to the poor Level-2 caching behavior. We emphasize again that the limited main memory bandwidth of the evaluation board considerably restricts the clock rate we applied to the FD computing engine. For example, a typical fully-pipelined floating-point arithmetic unit such as a

multiplier or adder can work at over 200 to 300 MHz on FPGAs. Suppose we had integrated one 200MHz 72-bit DDR-SDRAM memory module on the FPGA-enhanced computer platform, (This is what we generally have in today's commodity computers.) Because the aggressive onboard memory bandwidth is 800 million words per second, this imaginary platform could allow our computing engine to operate at a sustained speed of 200 million grid computations per second, which is four times faster than the result we obtained on the ML401 platform. Furthermore, if there were more dedicated memory channels available on board, we can select to construct multiple FD computing engines and let them work concurrently to process their own dataset. Because of the nearly perfect parallelism of FD methods, the computational performance would be scaled almost linearly.

Table 11. Performance Comparison for FD Schemes on FPGA and PC

FD schemes	Software Computational Throughput (Million Grid / second)	Hardware Implementation		
		Computational Throughput (Grid/second)	Speedup	Resource Utilization (RAM Blocks/ DSP Slices/ Logic Slices)
(2, 2)	33.27	49.71	1.49	14/10/4885
(2, 4)	27.53	49.63	1.80	18/22/7212
(2, 8)	19.90	49.50	2.49	26/30/9732
(4, 4)	15.10	48.38	3.20	26/30/9818

5.3.2 Acoustic Modeling of Marmousi Mode

In this example, we apply our FPGA-based FD methods to accelerate a realistic 2D seismic modeling problem. The Marmousi model is a prevailing 2D model with a complex underground structure. It was synthesized in the 1990s and had been widely used as a benchmark in the seismic data processing industry for calibrating migration (imaging) algorithms. The 2D grid mesh covers a geographical domain of 9200m by 3000m with 4m sampling interval, which leads to 2300×750 spatial grids, 8250 time-integration steps for 3 second wave traveling time, and so 1.432×10^{10} grid computations in total.

20 damping layers are attached on the left, right and bottom of the computational domain to absorbing outgoing energies, while at the mean time, free surface boundary condition is applied on the top. The DBC introduces only less than 3 percent of additional computational workload. We excite the 2D field with a Ricker wavelet at x=5000m, z=8m. The maximum frequency in source wavelet is set at 80Hz. Figure 29 shows the Marmousi velocity model together with four snapshots obtained by the finite-accuracy optimized (2, 8) FD computing engine (see Section 5.4 for details) on the ML-401 evaluation board. We can notice that there are no visible numerical reflections arising from the vicinity of boundaries, which reveals the effectiveness of our DBC scheme.

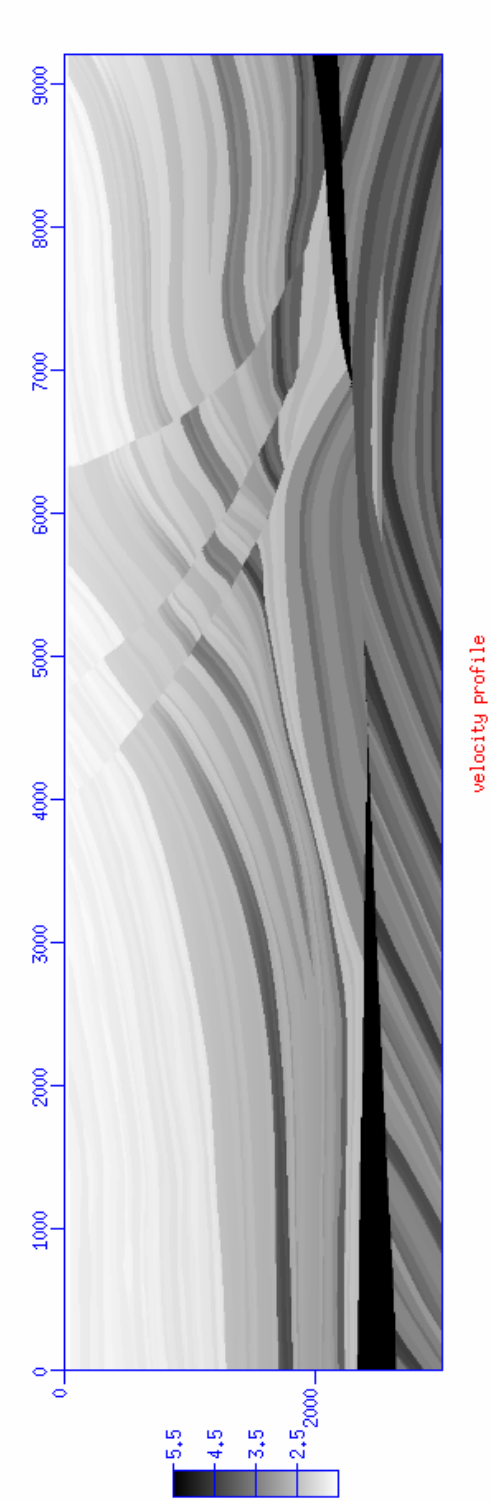

Figure 29. Marmousi Model Snapshots (t=0.6s, 1.2s, 1.8s, and 2.4s. Shot at x=5km)

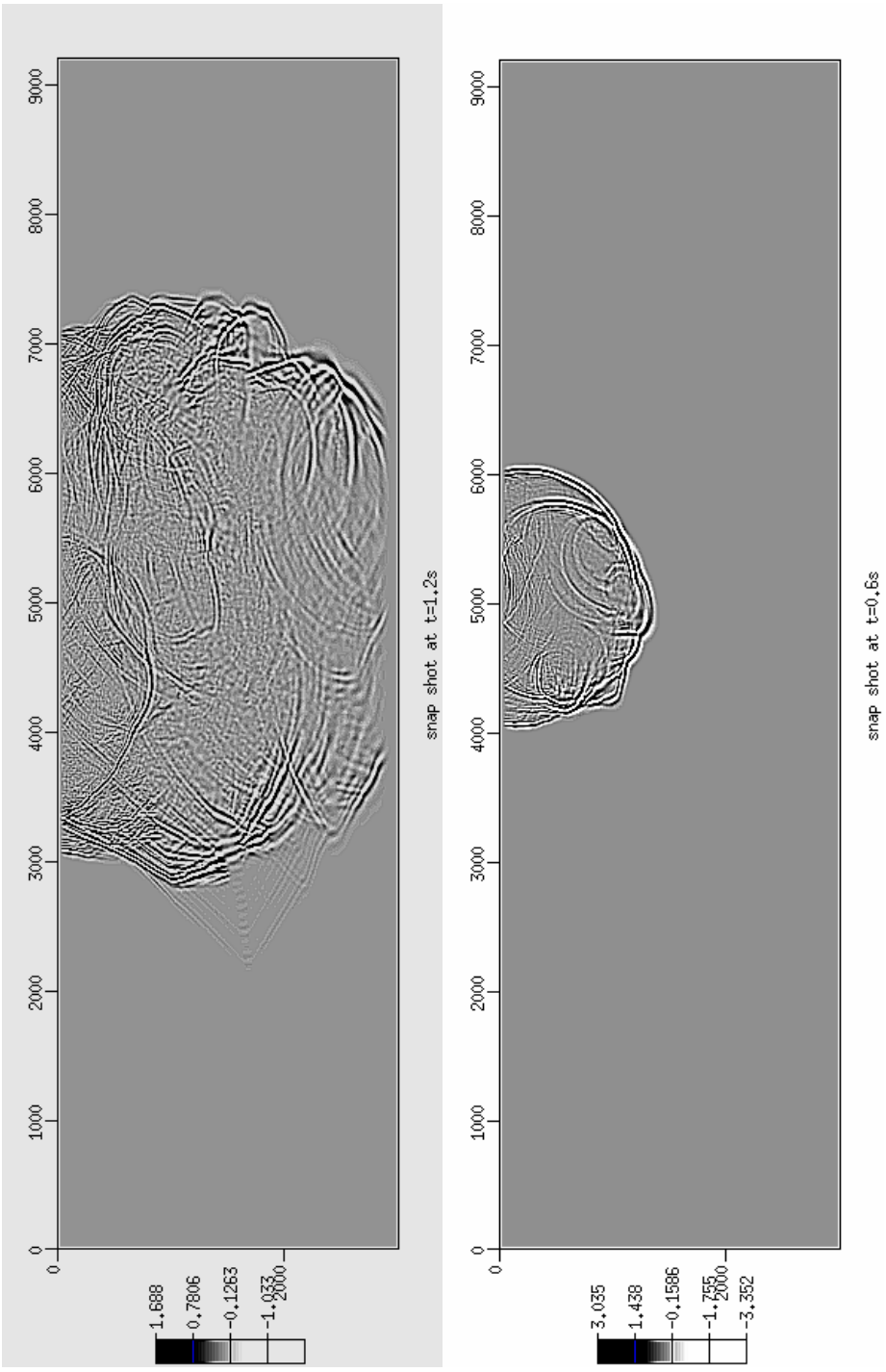

Figure 29. Continued

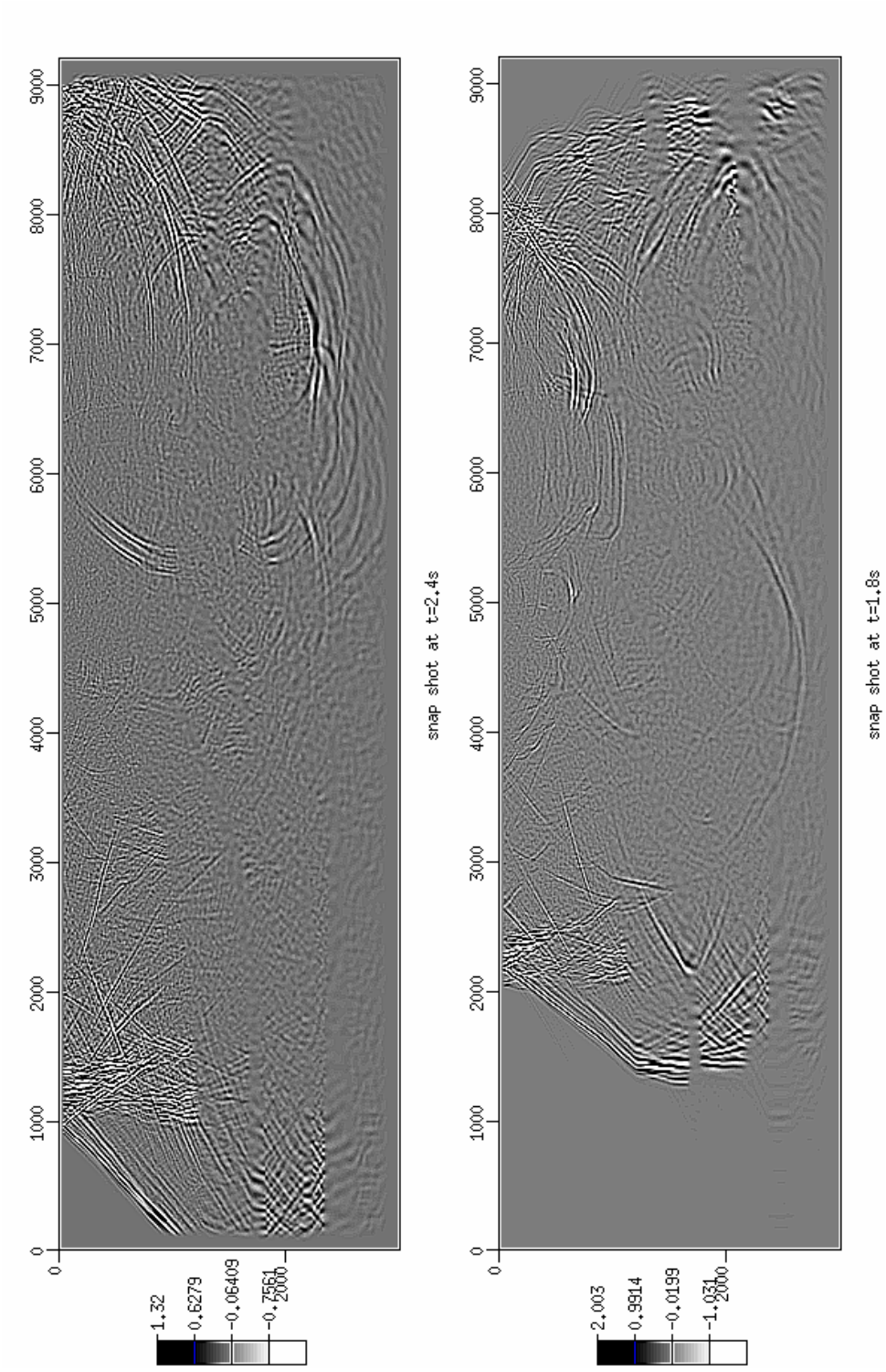

Figure 29. Continued

5.4 Optimized FD Schemes with Finite Accurate Coefficients

Using IEEE-754 compliant floating-point cores as core components to construct the high-order FD computing engine on FPGAs, although straightforward and convenient, will consume considerable hardware resources and leads to excessive pipeline stages. As we mentioned above, constructing a 16th-order 2D Laplace computing engine needs 53 floating-point arithmetic units (33 adders and 20 multipliers). The number will rise to about 140 if we decide to build a staggered 4th-order FD computing engine for elastic modeling problems [59]. For these complex cases, the computing engine becomes too big to be accommodated in even the largest FPGA device. In [57], the authors tried to solve electromagnetic FDTD methods using pure fixed-point arithmetic on FPGAs. Although significant hardware resources were saved, the well-bounded worst-case numerical error provided by standard floating-point arithmetic was destroyed. Correspondingly, this approach might work well in some tests, but it is not guaranteed to produce correct result for every problem without formal error analysis.

In this work, we decide to remain the floating-point format of wave field values unchanged, while adjusting those FD coefficients to simplify the implementation of floating-point arithmetic and save programmable hardware resources in FPGA. Because values of those floating-point FD coefficients span in a wide range, representing them in a fixed-point format will lead to excessive binary bits. Another option is to remain the floating-point representation, but customize their format by rounding the mantissa portion to fewer binary bits than the standard. Unfortunately, this approach has only limited impact and will lead to unacceptable numerical dispersion errors in most cases. In Figure 30, we compare dispersion errors of the maximum 8^{th}-order FD scheme with accurate FD coefficients, the scheme that truncate the mantissa portion of its FD coefficients to 16 binary bits, and the scheme that use only 8 mantissa bits. We can observe that simply truncating mantissas may save us only minimal programmable hardware resources but not too many. For example, we may use customized floating-

point format with 16 mantissa bits without significantly deteriorating numerical features. If we were overly aggressive by choosing only 8 binary bits, the numerical dispersion error becomes unacceptable.

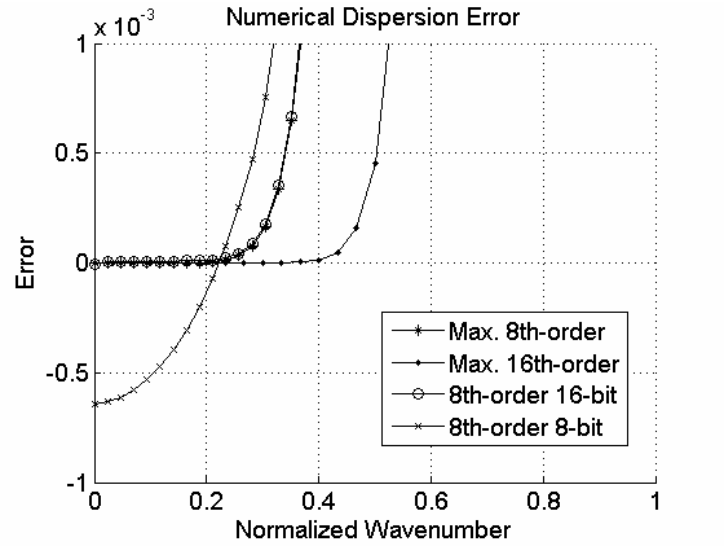

Figure 30. Numerical Dispersion Errors for the Maximum 8th-Order FD Schemes with 23, 16, or 8 Mantissa Bits

We attack this problem by means of improved numerical algorithms/methods. We try to design a new class of FPGA-specific FD schemes, which can be implemented much more efficiently with similar numerical performance and computational accuracy as the standard maximum order FD schemes. As we introduced above, maximum order FD schemes determine their coefficients by cancelling as many the lower order Taylor expansion terms as possible. Although optimal from a mathematical standard, by examining the dispersion relation plotted in Figure 18 and 19, we can tell that this class of FD schemes provides excessive accuracy within low wave-number band at the cost of rapidly-worsening numerical performance in high wave-number band. To compensate for this disadvantage, we designed a new class of FD schemes by improving their

numerical dispersion relation rather than pursuing formal accuracy representation. This class of methods also utilizes central $(2m+1)$-point stencil to approximate each spatial second derivative term. By dropping the requirement of maximizing the order of un-cancelled truncation term, we now need only the least necessary accuracy order to keep our FD schemes numerically convergent to the original PDE. The values of all remaining FD coefficients in Equation (5.7) are optimized to minimize the square of phase-speed errors of the discretized wave equation inside its working wave-number band, which also corresponds to a frequency domain Least Square (LS) error criterion.

Keeping the LS criterion in mind, we now further optimize these FD coefficients so that they can be represented by only a few binary bits without seriously deteriorating its frequency-domain dispersion relation. This optimization is meaningless for commodity computers with standard floating-point arithmetic units. However, its advantage on FPGA-enhanced computers is obvious: For FPGA devices without hardware multipliers, fewer partial sums would lead to less occupation of programmable hardware resources; FPGA devices with on-chip multipliers can also benefit from this approach because of the reduction of latency. Mathematically, this is a constrained optimization problem, which can be solved by standard methods such as the Simplex method [70]. In practice, Simplex may stagger at concave points or even diverge when the problem is ill-conditioned. To address this problem, in this work, we designed a class of finite-accurate FD coefficients optimization algorithm, which is a simple heuristic approach to reach sub-optimal results.

Algorithm 3. Finite-accurate FD coefficients optimization

Calculate all (m+1) LS-based optimal coefficients $\alpha_r^m\big|_{r=0\cdots m}$

For r = m down to 2

Restore the leading one of the mantissa portion of α_r^m

Round it to the most six significant bits

Solve the LS problem with $\alpha_k^m\big|_{k=r\cdots m}$ as constraint conditions

End

Set α_0^m equals to $2\cdot\sum_{r=1}^{m}\alpha_r^m$ for consistency

Round α_0^m and α_1^m to 18-bit fixed-point format

In this algorithm, the largest two coefficients in the middle of FD stencils are represented by 18-bit fixed-point format because their values are pivotal to the performance of numerical dispersion relation. (The number 18 is selected because of the word-width of on-chip multipliers inside Xilinx's FPGA devices.) For all others, we represent them in floating-point format with only six mantissa bits so that corresponding multiplications can always be finished by an on-chip multiplier within one clock cycle. Further hardware optimization is possible if we select to make use of some new features of up-to-date FPGA devices. We always normalize the leading bit of these coefficients to be 1 by shifting so that all six bits are effective in our representation. Expressing exponents of these coefficients explicitly are unnecessary because the corresponding exponent adjustment can be easily implemented by adding a constant to the original exponent of the multiplicand. Figure 31 shows us the structure of this fully-customized floating-point constant multiplier. We note that the word-width of the product could be 42 or 31 according to the word-width of the coefficient.

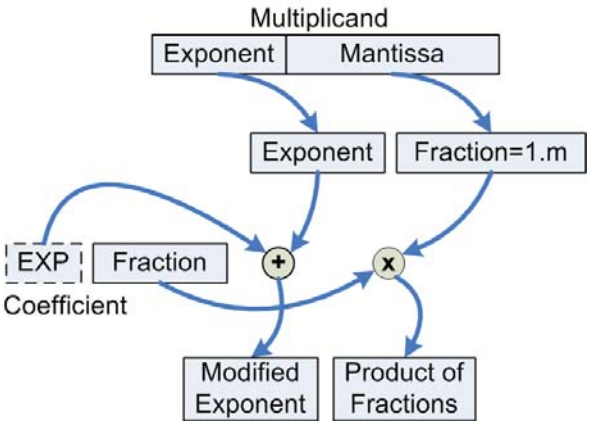

Figure 31. Structure of Constant Multiplier

We notice from the algorithm that those FD coefficients are decided one by one from the smallest to the largest. This sequence is critical to the final numerical accuracy of this class of FD methods. An intuitive explanation is that once rounding errors are introduced into large coefficients, it is difficult to compensate by only adjusting those smaller ones.

Table 12. Coefficients of 3 FD Schemes with 9-Point Stencils

Coefficients	FD schemes		
	Maximum 8^{th}-order	Optimal	Finite-accuracy (Hex.)
α_0^4	2.8472223	2.9555101	2.9552002(403d2200)
α_1^4	3.2000000	3.3781638	-3.3778076(c0582e00)
α_2^4	0.4000000	0.4969828	0.4970703(3efe8000)
α_3^4	0.0507937	0.0828245	-0.0830078(bdaa0000)
α_4^4	0.0035714	0.0084953	0.0085449(3c0c0000)

A simple example is used to show the effectiveness of this heuristic optimization algorithm. We select FD schemes with 9-point stencil, so there are in total five unknown

coefficients. We also restrict the normalized working wave-number band to be less than 0.5, which corresponds to four spatial sampling points per shortest wavelength. Table 12 lists values of these coefficients for the 8th-order FD scheme, the full-accurate optimal scheme, and our finite-accurate scheme. Figure 32 plots dispersion relations for these three FD schemes together with the ideal wave equation. Figure 33 is an amplified version of Figure 32, showing us numerical errors caused by FD approximations. We also show in these figures the dispersion relation of the maximum 16th-order FD scheme as reference. From these figures, we can observe that optimized FD schemes provide much wider effective working wave-number band than maximum-order schemes at the cost of negligible approximation error. Furthermore, our finite-accurate schemes can provide similar performance as full-accurate optimal schemes with much simpler coefficient representations.

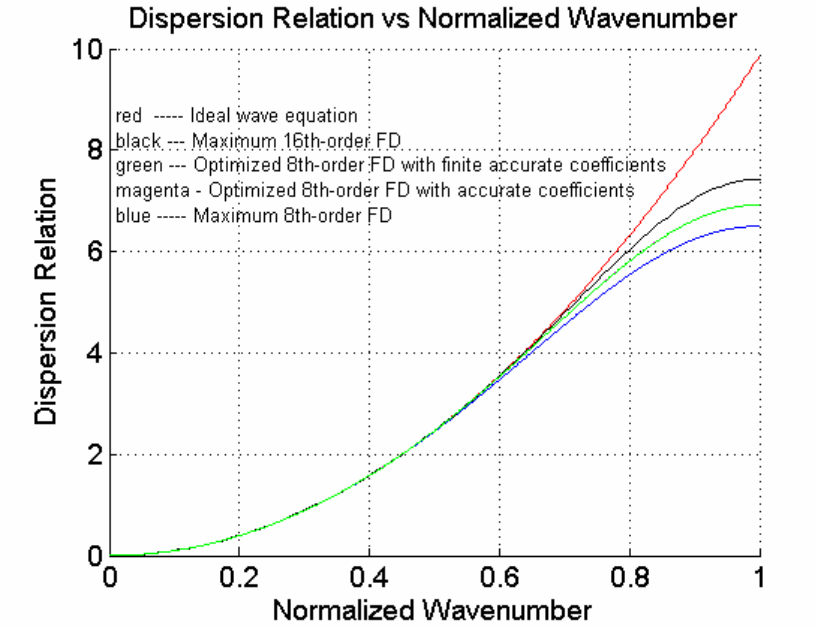

Figure 32. Comparisons of Dispersion Relations for Different FD Approximations

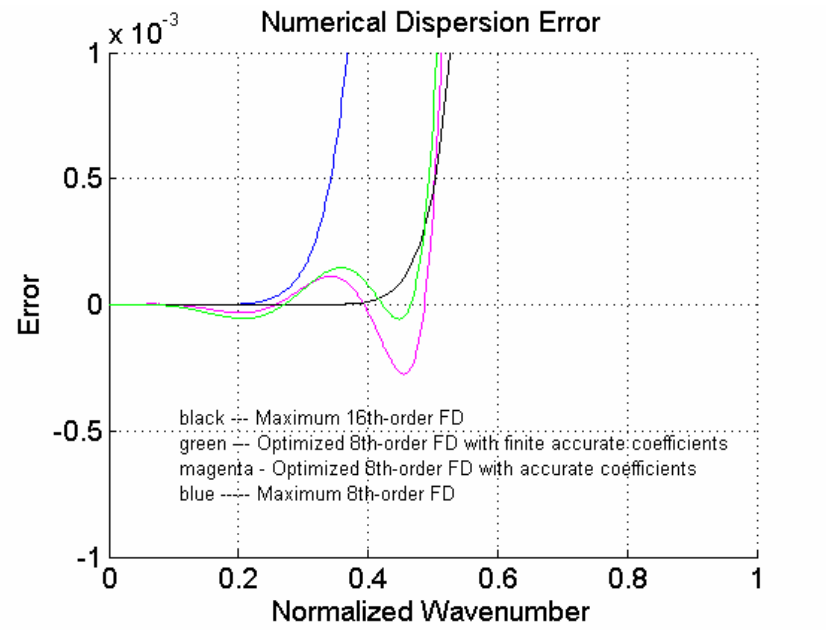

Figure 33. Dispersion Errors for Different FD Approximations

5.5 Accumulation of Floating-Point Operands

In previous sections, we introduced our design of a fully-customized data buffering system to improve data reusability. This sliding window-based structure is constructed to feed the FD computing engine a new set of operands at every clock cycle so that the wave-field grids can be updated at the same speed. Correspondingly, all of those constant coefficient multipliers work parallel to create a new set of products, after which the intermediate results are accumulated set by set at the same speed to achieve sustained high computational throughput.

A fully-pipelined floating-point adder tree structure, as shown in Figure 34, is most people's first choice to meet the requirements. However, it is not always the best choice for us to implement a numerical algorithm/method by simply mapping the software subroutine or computations into FPGA. Specifically, for summation of floating-point operands, this naive binary-tree based reduction circuit has the following pitfalls:

- Floating-point addition is one of the most complicated computer arithmetic, which in general, costs an FPGA device several hundreds of Logic Slices. Only the largest and most expensive FPGA chips can provide enough programmable resources to accommodate tens of floating-point adders demanded by FD schemes with wide stencils.
- In order to achieve high data throughput, a floating-point adder in general consists of seven to ten sub-stages. A large adder tree structure can easily introduce hundreds of pipeline stages, which may complicate the internal data buffering design, and also affect the pipelining efficiency because of the start-up latency.

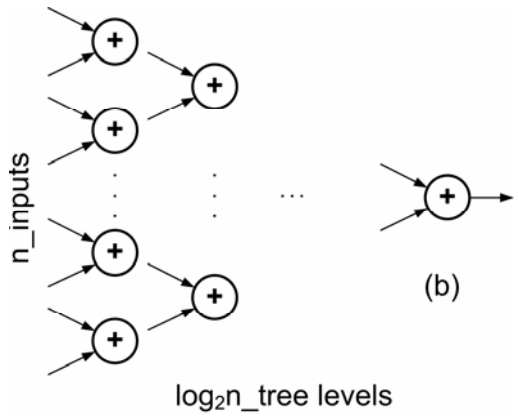

Figure 34. Binary Tree Based Reduction Circuit for Accumulation

In this section, we propose a group-alignment based summation algorithm to accumulate those floating-point products produced by coefficient multipliers in floating-point/fixed-point hybrid arithmetic. This hardware-based algorithm can result in similar, or even better, worst-case absolute and relative errors as ordinary summation methods using standard floating-point arithmetic. Also, the total number of pipeline stages required for the new FD computing engine would be reduced significantly. As we will introduce in the next section, the combination of the optimized FD coefficient multipliers and the group-alignment based summation unit would save us considerable

hardware resources while implementing on FPGA-enhanced computers. Consequently, higher-order FD schemes could be adopted for better numerical performance. Here, we bring out the basic idea of this approach and postpone detailed numerical analysis and proof to the next section.

Our goal is to design an FPGA-specific numerical algorithm to compute the floating-point summation $S = \sum_{i=1}^{n} s_i$ with similar or better relative and absolute errors than the standard IEEE-754 compliant floating-point arithmetic. Furthermore, the algorithm's hardware implementation should be compatible with the proposed FPGA-enhanced computer platform. It is asked to accept a new set of floating-point inputs and produce a new output in the same format at each clock cycle if pipelining technique is properly applied. Here, we propose the group-alignment based floating-point summation algorithm as follows:

Algorithm 4. Group-alignment based floating-point summation

Input: A set of n floating-point numbers s_1, s_2, …, s_n from preceding multipliers

Output: The summation result $S = s_1 + s_2 + \cdots + s_n$

1. Split each summand s_i into two portions as Mantissa and Exponent and restore every Mantissa portion M_i to normalized form F_i.
2. Find the largest Exponent E_{max} within E_i
3. Calculate (E_{max} -E_i) for i=1 to n
4. Shift F_i to right by (E_{max} -E_i) bits, round the shifted fractions to nearest if necessary.
5. Sum up all shifted F_i
6. Normalize the summation result into IEEE- compliant format.

The basic idea of this group-alignment based summation algorithm is relatively straightforward: Instead of the original floating-point adder tree structure, where the

comparison of exponents and the addition of shifted mantissa are scattered in different adders, we now collect them to form an exponent comparison tree in Step 2 and a fixed-point adder tree in Step 5. The hardware correspondences for other steps such as 1, 3, and 4 are also clustered together so that only one pipeline stage is needed for each function step. The advantage of this arrangement is obvious: We now do not have to round or normalize every intermediate addition result at the final stage of each floating-point adder. Instead, only one step at the end of this large summation unit is sufficient. Correspondingly, we can save almost half of reconfigurable hardware resources in FPGA-based implementation, and the total number of pipeline stages is reduced significantly. Figure 35 shows us the structure of this group-alignment based floating-point accumulator.

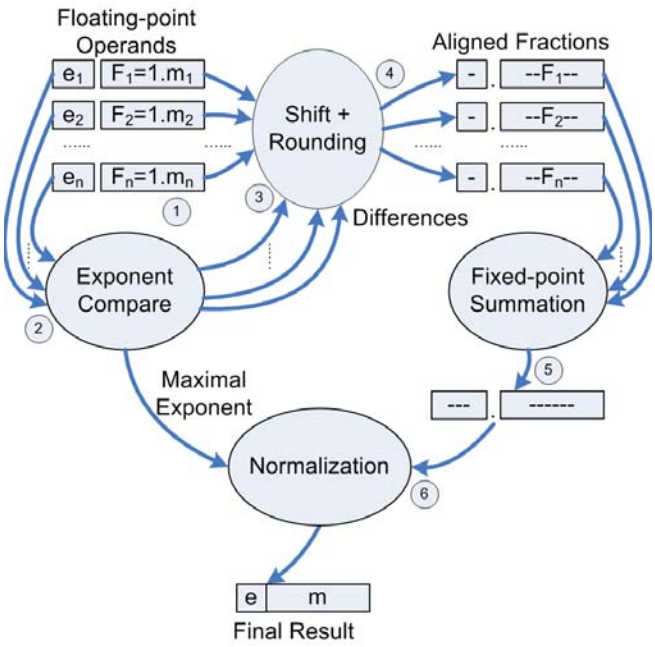

Figure 35. Structure of Group-Alignment Based Floating-Point Accumulator

5.6 Bring Them Together: Efficient Implementation of the Optimized FD Computing Engine

We've proposed a new class of optimized finite-accurate FD schemes, whose FD coefficients are optimized to be represented by only a few binary bits without deteriorating numerical accuracy criterions. Furthermore, we replace the subsequent costly floating-point adder tree by a floating-point/fixed-point hybrid accumulator utilizing group-alignment technology. The resulting fully-pipelined FD computing engine with finite accurate coefficients can provide similar, or even better, worst case relative and absolute rounding errors than ordinary design using standard floating-point arithmetic, but consumes only a fraction of programmable hardware resources.

As an example, the 9-point finite-accurate FD computing engine is implemented on an entry-level Virtex-4 ML-401 evaluation board. FPGA resource occupations, as well as computational performance, are analyzed and compared with our previous designs using standard single-precision floating-point arithmetic units. We rewrite the 9-point 1D Laplace operator as follows:

$$\left.\frac{\partial^2 P}{\partial x^2}\right|_i^{9-point} = \left(\frac{\alpha_0^4 \cdot P_i + \sum_{r=1}^{4} \alpha_r^4 \cdot (P_{i+r} + P_{i-r})}{(dx)^2} \right) + O\left((dx)^8\right) \qquad (5.17)$$

Those constant values of coefficients $\alpha_r^4\big|_{r=0,\cdots,4}$ for different FD schemes can be found in Table 9.

Ignoring the division by $(dx)^2$, which can be easily absorbed into those constant coefficients, there are, in total, 5 multiplications and 8 additions. The fully-pipelined implementation shown in Figure 36 is based on standard single-precision floating-point arithmetic. It costs 20 embedded 18X18 multipliers and over 3,300 Logic Slices. The total number of pipeline stages is about 50.

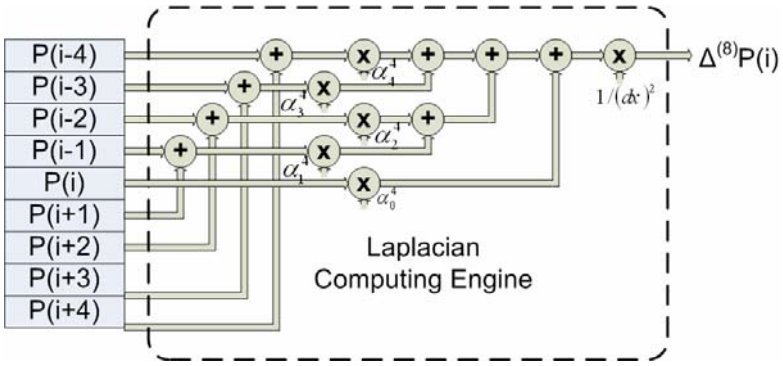

Figure 36. Structure of 1D 8th-Order Laplace Operator

An in-depth observation reveals that for this straightforward implementation, considerable hardware resources are wasted mostly on unnecessary normalization or alignment stages before and after every floating-point arithmetic unit. Moreover, inappropriate rounding on intermediate results may significantly jeopardize the numerical accuracy of ill-conditioned summations. In our new implementation, the optimized finite-accurate FD computing engine is customized globally to eliminate redundant operations. A floating-point/fixed-point hybrid approach is adopted, where the input and output values of this computing engine are all in floating-point representations to be compatible with conventional data processing software, but almost all internal stages use fixed-point arithmetic to save hardware resources as well as pipelining latencies. Figure 37 shows the corresponding hardware structure.

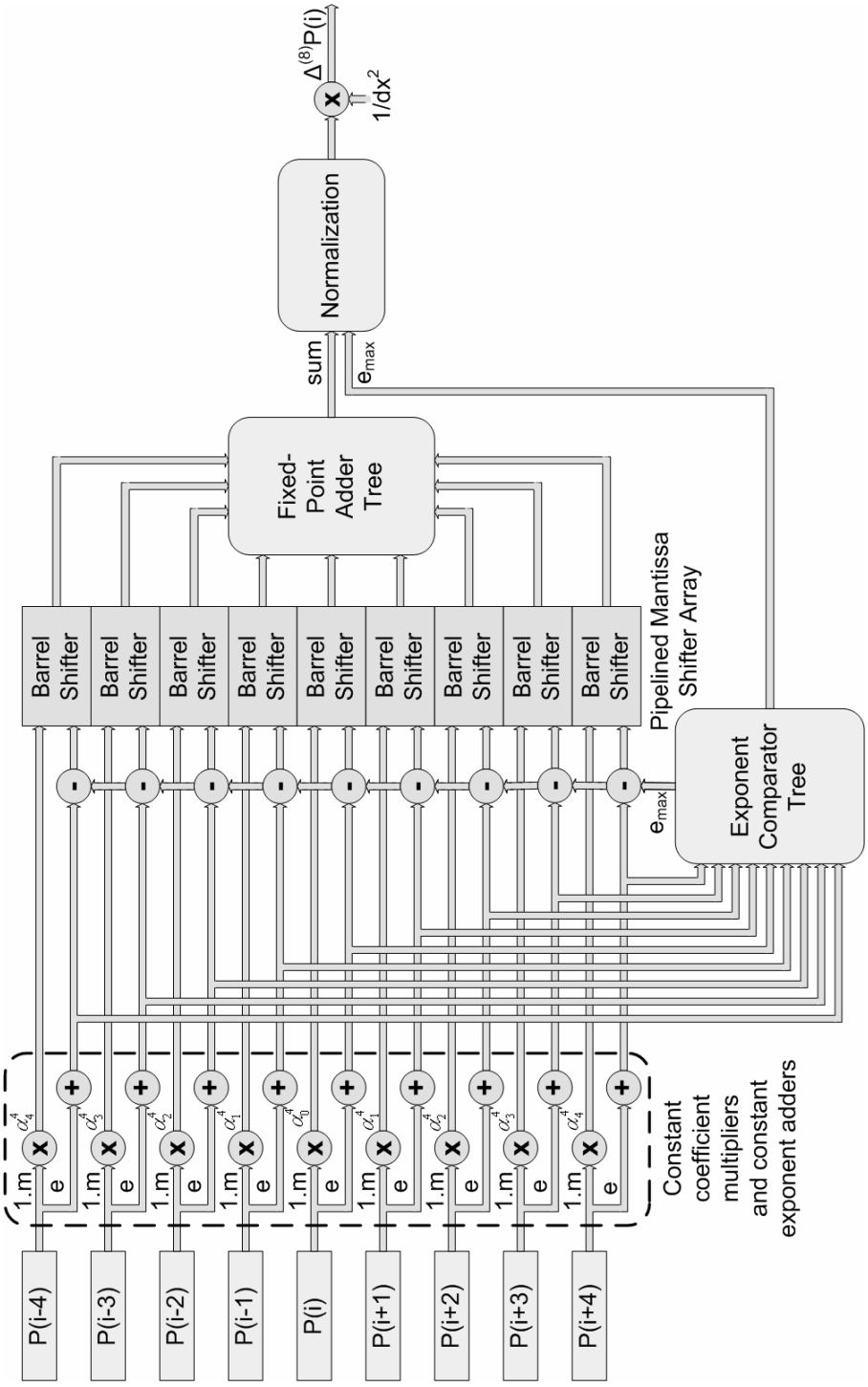

Figure 37. Structure of 1D 8th-Order Finite-Accurate Optimized FD Scheme

Utilizing embedded hardware multipliers may result in efficient and high-speed implementation for computation-dominated applications. However, because their layout inside an FPGA chip is pre-assigned to satisfy algorithms with some special patterns, one might always feel that the position constraints seriously limit their utilization for the application at hand. One important benefit of the optimized finite-accurate FD coefficients is that only the largest two coefficients in the middle of FD stencils are in 18-bit fixed-point format, which is ideal with embedded 18X18 multipliers. All other coefficients are represented by only six binary bits. The corresponding multipliers can be constructed efficiently by distributed logic slices. For example, a fixed-point 24-by-6 bits fully-pipelined parallel multiplier costs less than 90 logic slices. Moreover, there are no particular position constraints for it so that this multiplier could be placed anywhere inside an FPGA device.

Another benefit is not so obvious: In Figure 36, considerable global interconnection resources inside the FPGA chip are consumed to send operands to first-level adders. This structural property inevitably leads to cumbersome layout and low clock frequency, especially for FD schemes with wide stencils. By adopting the finite-accuracy optimized FD schemes, hardware resources required by coefficient multiplications decrease significantly, and so we are now allowed to exchange the arithmetic order of first-level additions and subsequent multiplications as we show in Figure 37. Although the number of coefficient multipliers is almost doubled, they occupy similar, or even less, hardware resources than before. Because most data paths are localized, the computing engine can now work at a much higher clock rate. Consequently, better computational performance would be achieved.

Utilizing nine unfolded barrel shifters, outputs of nine constant multipliers are aligned to the one with the largest exponent. Round-to-nearest scheme is applied to every shifter. In our implementation, the word-width of these barrel shifters is set to 42 bits conservatively, which is also equal to the largest word-width of those products produced by preceding multipliers.

This new floating-point/fixed-point hybrid approach provides much better worst case error bound than standard floating-point arithmetic with less than half of programmable hardware resources consumed. Our implementation costs only 6 embedded 18X18 multipliers and about 1500 Logic Slices, which corresponds to less than 20% of available FPGA resources on the evaluation board. Furthermore, the regular layout and localized interconnections of this design lead to much higher computational throughput, especially for FD schemes with wide stencils. The clock rate applied to the computing engine is about 200MHz for 13 pipeline stages or 230MHz for 15. Higher clock rate can also be achieved if more pipeline stages are introduced. Simplicity and scalability are other desirable properties of this design. Choosing more fraction bits into the new summation unit consumes negligible additional hardware resources, but can significantly improve numerical error bounds.

6. FEM ON FPGA-ENHANCED COMPUTER PLATFORM

In this section, we will introduce our work on accelerating Finite Element Methods (FEM) on the proposed FPGA-enhanced computer platform. The origins of FEM can be traced back to the 1940s, when they were developed to solve complex structural modeling problems in civil engineering. Mathematically, the solution is based either on eliminating the differential equation completely (steady state problems), or rendering the PDE into an equivalent ordinary differential equation (ODE), which is then solved using standard numerical methods. Today, FEM has been generalized into an important branch of applied mathematics for numerical modeling of physical systems in a wide variety of scientific & engineering fields such as electromagnetics, fluid dynamics, climate modeling, reservoir simulation, etc.

As we've already learned in Section 5, the primary challenge in solving PDEs is to approximate the equation to be studied with an expression that is numerically consistent and stable. By doing so, the numerical solution would finally converge with the original PDE if appropriate discretization is applied. In other words, errors hidden in input data sets and emerging from intermediate calculations would not accumulate severely, and would eventually lead to contaminated results. Just as in FDM, FEM begins with mesh discretization of a continuous physical domain into a set of discrete sub-domains so that the original infinite-dimensional equation is projected into finite dimensional subspace. From this point of view, FDM could be treated as a special case of FEM. However, FDM requires the entire computational domain to be discretized with rectangular shapes or simple alternatives. Moreover, the discretization must be sufficiently fine to resolve both the high-frequency wavelet components and the smallest geometrical feature. Consequently, large computational domains could be developed, which would result in extraordinary computational workload. Comparatively, FEM is a good choice for solving PDEs over a large computational domain with complex boundaries and minute geometrical features, or when the desired precision varies over the entire domain. For

instance, when simulating weather patterns on Earth, it is more important to make accurate predictions over land than over the wide-open sea.

The first step of FEM is meshing, which is the process of breaking up a continuous physical domain into smaller sub-domains (elements). For example, surface domains may be subdivided into triangular or quadrilateral shapes, while volumes may be subdivided into tetrahedral or hexahedral shapes. Although mesh generation is a key portion of FEM and will decide the final approximation accuracy, it is only one of the pre-processing steps of FEM and consumes a small portion of the total execution time. Moreover, mesh generation methods in general integrate complex control-dominated subroutines such as automatic grid generation or adaptive mesh refinement, and therefore could not be accelerated effectively with today's PFGA devices.

The final and most time-consuming step of FEM is the solution of large linear system equations generated by discretizing PDE. Basic Linear Algebra Subprograms (BLAS) are standard toolkits for scientists and engineers to perform such linear algebra operations. Because of their popularity, these subroutines are always well-tuned by software vendors, and therefore execute very fast on commodity computers. We know that the computational complexity of the solution of a linear system equations using standard linear algebra methods such as Gaussian Elimination (GE), QR or LU factorization is $O(n^3)$, where n is the number of system unknowns. Consequently, such conventional methods are considered feasible only for small problems, with up to thousands of system unknowns. For most realistic numerical PDE problems, where large linear system equations with millions, even billions, of unknowns are involved, such computation-dominated methods are generally impractical. Fortunately, the matrices derived from such problems are sparse (most entries of the matrix are zero). So, another class of linear system equations solvers, called iterative methods, could be applied for their rapid (but inaccurate) solutions.

Because of all the reasons mentioned above, in this work, we decide not to reference mesh generation subroutines of FEM, but investigate FPGA-specific methods for rapid solutions of basic BLAS subroutines on the proposed FPGA-enhanced

computer platform. We will introduce our work on accelerating all three levels of BLAS functionality as Level-1 regarding vector operations, Level-2 regarding matrix-vector operations, and Level-3 regarding matrix-matrix operations. Results of this work can be applied directly to accelerate the solution of dense/sparse linear system equations.

Floating-point arithmetic is always preferable to fixed-point arithmetic in numerical computations because of its ability to scale the exponent for a wide range of real numbers. With commercial or open-source parameterized floating-point libraries [21] [22], floating-point computations on FPGAs become straightforward and convenient. However, this standard approach requires much more programmable hardware resources than their fixed-point counterparts. Moreover, it leads to excessive computation latency. For example, constructing a fully pipelined single-precision floating-point adder on Xilinx Virtex-II Pro device consumes over 500 logic slices with 16 pipeline stages [71]. Analogously, a double-precision unit with similar performance costs nearly 1000 logic slices with over 19 pipeline stages.

For specific S&E problems at hand, it is always possible, and indeed desirable, to adopt customized floating-point formats by making tradeoffs among accuracy, area, throughput, and latency [72] [73]. Efforts were also made to investigate the possibility of replacing floating-point operands and operations by fixed-point arithmetic thoroughly [74] [57]. However, most of these works only demonstrated these feasibilities by numerical evidences without rigorous mathematical proofs, and therefore were unconvincing for religious numerical scientists.

Instead of directly enhancing the capability of standard floating-point arithmetic, in this work, we select to boost the computational performance of basic BLAS subroutines on FPGAs by improving their numerical accuracy and hardware efficiency. Here, we interpret improved numerical accuracy as similar, or even better, relative/absolute rounding error compared with software subroutines using standard floating-point arithmetic. Hardware efficiency means that the resulting implementation consumes comparable or less hardware resources on FPGAs than existing conventional designs,

but can always achieve better sustained execution speed with only moderate start-up latency.

6.1 Floating-Point Summation and Vector Dot-Product on FPGAs

6.1.1 Floating-Point Summation Problem and Related Works

Floating-point summation is such an important operation in numerical computations that some computer scientists even suggested to import it into general-purpose CPUs as "the fifth floating-point arithmetic" [75]. Unlike other fundamental floating-point arithmetic, summation is not well-conditioned because of so-called "catastrophic cancellations" [76], i.e. small relative errors hidden in inputs might lead to significant relative error in output. To complicate the situation, the sequence of consecutive additions will also affect the final sum, resulting in un-unique solutions for the same input data set on different computer platforms with different programming languages or software compilers [77]. On the bright side, this un-uniqueness relieves us from strictly obeying the IEEE standard in our FPGA-based implementation. The only constraint imposed in our design is to select the exact solution as accuracy criterion.

Given a set of n floating-point numbers $s_1, s_2, \cdots, s_n$ with small relative errors ε_M, our goal is to design an FPGA-specific numerical algorithm as well as its hardware implementation to compute the summation of $S = \sum_{i=1}^{n} s_i$ accurately and efficiently. Specifically, we assume that the summation unit has only one input port to feed in summands as well as one output port to send out final results. This assumption is consistent with the memory-bandwidth-bounded property of floating-point summations because there is only one addition corresponding to each summands.

The following criteria are used to compare the performance of our FPGA-based design with others:

a. Efficiency. The summation unit consumes comparable FPGA programmable hardware resources and can work at a similar or even higher speed as a conventional fully-pipelined floating-point accumulator.
b. Throughput. The fully-pipelined summation unit should work at its highest sustained speed, accepting a new summand at every clock cycle without excessive pipelining stalls.
c. Latency. The summation result should be available with only moderate latency after the last summand entering this arithmetic unit.
d. Accuracy. Summation results of this new arithmetic unit should have similar, or better, relative and absolute errors than results produced by the standard sequential accumulation algorithm utilizing IEEE-754 compliant floating-point arithmetic.

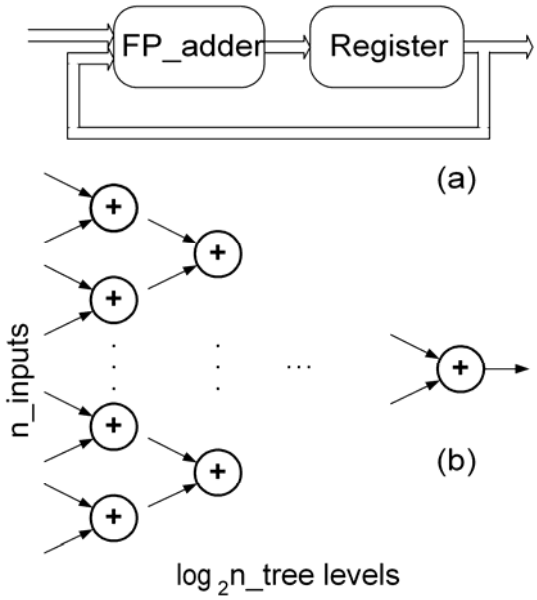

Figure 38. Conventional Hardwired Floating-Point Accumulators (a) Accumulator with Standard Floating-Point Adder and Output Register; (b) Binary Tree Based Reduction Circuit

At first glance, a simple accumulator using one floating-point adder with its registered output connecting to one of the adder's input ports can do the job perfectly (Figure 38(a)). However, unlike a single-cycle fixed-point accumulator whose output can be fed back to the input port immediately as intermediate results for consecutive summations, the data dependency associated with the pipelined floating-point adder will severely slow down the data throughput of the corresponding accumulator. This deficiency doesn't meet the requirement we just mentioned in criterion (b). Another extreme is to accumulate n floating-point numbers using a binary tree based reduction circuit with $(n-1)$ adders and $\log_2 n$ tree levels (Figure 38(b)). If the pipelining technique was properly applied, this approach can accept a new set of inputs and produce a new output at each clock cycle. Therefore, this approach is too extravagant for our single input/output unit and won't result in efficient implementation as we required in criterion (a).

One possible way to address the data dependency problem caused by pipelined addition is to introduce a "schedule" circuit and carefully-designed input/intermediate-sum buffers [78]. This approach treats all intermediate-sums the same way as input summands, and so the original summation task with $(n-1)$ summations and n summands are now converted to $(n-1)$ additions of $(2n-2)$ addends. With the help of efficient data buffering design, ideally this task could be finished by an adder within $(n-1)$ clock cycles. However, limited by the only external input port of the summation unit, the adder has to rely on its own output to feed back operands, which may not always arrive on time because of the adder's deeply-pipelined internal stages. In order to finish the summation task as soon as possible, the main task of the scheduler is to monitor the internal dataflow of the pipelined adder and try its best to feed the adder's two input ports with operands. Because considerable idle cycles would always be inserted at the beginning and the end of the process, latency may become a potential problem for this approach, especially when n is not a large number. Moreover, the scheduler requires considerable register resources to buffer inputs and intermediate-sums, which may be unfeasible on FPGAs without internal SRAM blocks.

In [79], the authors proposed a technique called “delayed addition” to remove carry propagation from the critical paths of pipelined integer or floating-point accumulations. The resulting FPGA-based implementation meets our requirements for high-throughput and low-latency. However, because considerable hardware resources are consumed to construct Wallace-tree based 3-2 compressors as well as overflow detection/handling circuit, this summation unit is four times more expensive than what we expected in criterion (a). Furthermore, unlike the design in [78], where the accuracy of the final result is guaranteed by the standard floating-point adder, the authors only demonstrated the feasibility of their approach by simple numerical tests without rigorous proof, and so its correctness and accuracy is still questionable.

6.1.2 Numerical Error Bounds of the Sequential Accumulation Method

Over the last few decades, people devised and analyzed many numerical algorithms for accurately computing floating-point summation. However, most of these software approaches are based on sorting the input data set in increasing or decreasing order by absolute value. Their computational complexity is dominated by sorting $(O(n \cdot \log n))$, so is much more expensive than the naïve sequential accumulation method.

Here, we first analyze error properties of sequential accumulation. Then, we use the result as reference to steer our new FPGA-specific summation algorithm. Assuming the rounding scheme of a computer with standard floating-point arithmetic is round-to-nearest-even, we have the following error bounds for fundamental floating-point arithmetic [80]:

Lemma 1: Let ε_M be the unit rounding-off on a computer with standard floating-point arithmetic ($\varepsilon_M = 2^{-24}$ for single-precision arithmetic, or $\varepsilon_M = 2^{-53}$ for double-precision arithmetic). Then the absolute error and relative error for fundamental floating-point operations (op can be either $+, -, \times, or \div$) can be represented as:

$$\left| fl(x\ op\ y) - (x\ op\ y) \right| \le \left| x\ op\ y \right| \cdot \varepsilon_M \tag{6.1}$$

$$\left| fl(x\ op\ y) - (x\ op\ y) \right| / \left| x\ op\ y \right| \le \varepsilon_M \tag{6.2}$$

Keeping this result in mind, we can easily prove by mathematical induction the following lemma:

Lemma 2: Let ε_M be the unit rounding-off on a computer with standard floating-point arithmetic. Then the absolute error and relative error bounds for floating-point summation: $S = s_1 + s_2 + \cdots + s_n$ introduced by the naïve sequential accumulation algorithm can be represented as:

$$\begin{aligned} \left| e_S \right| = \left| fl(S) - S \right| &\le \left(\left| s_1 \right| + \left| s_2 \right| + \cdots + \left| s_n \right| \right) \cdot (n-1)\varepsilon_M \\ &\le \max\left\{ \left| s_1 \right|, \left| s_2 \right|, \cdots, \left| s_n \right| \right\} \cdot n \cdot (n-1)\varepsilon_M \end{aligned} \tag{6.3}$$

$$\frac{\left| e_S \right|}{\left| S \right|} \le \frac{\left| s_1 \right| + \left| s_2 \right| + \cdots + \left| s_n \right|}{\left| s_1 + s_2 + \cdots + s_n \right|} \cdot (n-1)\varepsilon_M \tag{6.4}$$

where $\dfrac{\left| s_1 \right| + \left| s_2 \right| + \cdots + \left| s_n \right|}{\left| s_1 + s_2 + \cdots + s_n \right|}$ is the relative condition number of floating-point summation.

Proof:

Let $S_i = fl\left(s_1 + s_2 + \cdots + s_i \right)$. From Lemma 1, we have:

$$S_2 = fl(s_1 + s_2) = (s_1 + s_2)(1 + \varepsilon_1) = s_1(1 + \varepsilon_1) + s_2(1 + \varepsilon_1) \tag{6.5}$$

Similarly,

$$\begin{aligned} S_3 = fl(S_2 + s_3) &= (S_2 + s_3)(1 + \varepsilon_2) = s_1(1 + \varepsilon_1)(1 + \varepsilon_2) \\ &+ s_2(1 + \varepsilon_1)(1 + \varepsilon_2) + s_3(1 + \varepsilon_2) \end{aligned} \tag{6.6}$$

Continuing in this way, we can prove by induction that:

$$\begin{aligned} S_n = fl(S_{n-1} + s_n) &= (S_{n-1} + s_n)(1 + \varepsilon_{n-1}) \\ &= (s_1 + s_2)(1 + \varepsilon_1)(1 + \varepsilon_2) \cdots (1 + \varepsilon_{n-1}) + s_3(1 + \varepsilon_2) \cdots (1 + \varepsilon_{n-1}) \\ &+ \cdots + s_{n-1}(1 + \varepsilon_{n-2})(1 + \varepsilon_{n-1}) + s_n(1 + \varepsilon_{n-1}) \end{aligned} \tag{6.7}$$

Where $\left| \varepsilon_i \right| \le \varepsilon_M \quad for\ i = 1, 2, \cdots, n-1$

Because $\varepsilon_M << 1$, we have the following two relations:

$$(1+\varepsilon_1)(1+\varepsilon_2)\cdots(1+\varepsilon_{n-1}) \le 1+(n-1)(1+\delta)\varepsilon_M \tag{6.8}$$

$$(1+\varepsilon_i)(1+\varepsilon_{i+1})\cdots(1+\varepsilon_{n-1}) \le 1+(n-i+1)(1+\delta)\varepsilon \quad \textit{for } i = 2,3,\cdots,n \tag{6.9}$$

where δ is also a tiny quantity and can be absorbed into ε_M without affecting the final conclusion. We choose to ignore it from now on.

The worst-case absolute rounding error can be represented as:

$$\begin{aligned} |e_n| = |S_n - S| &\le |s_1(1+(n-1)\varepsilon_M) + s_2(1+(n-1)\varepsilon_M + \cdots s_n(1+\varepsilon_M)| \\ &\le \left(|s_1|+|s_2|+\cdots+|s_n|\right)\cdot(n-1)\varepsilon_M \le \max\left\{|s_1|,|s_2|,\cdots,|s_n|\right\}\cdot n\cdot(n-1)\varepsilon_M \end{aligned} \tag{6.10}$$

Correspondingly, the relative error can be represented as:

$$\frac{|e_n|}{|S_n|} \le \frac{|s_1|+|s_2|+\cdots+|s_n|}{|s_1+s_2+\cdots+s_n|}\cdot(n-1)\varepsilon_M \tag{6.11}$$

■

We also note from Equation (6.7) that the naïve sequential accumulation method doesn't result in a unique solution for different summation sequences. The "less than or equal to" signs in expression (6.3) and (6.4) imply that the error bounds here are tight. Putting it another way, we may always encounter the worst case if we mistakenly choose summation sequence or input data set.

6.1.3 Group-Alignment Based Floating-Point Summation Algorithm

The un-uniqueness of floating-point summations relieves us from strictly complying with the IEEE standard. Here, we select the exact solution as the only accuracy criterion and propose the group-alignment based floating-point summation algorithm as shown in Algorithm 5.

We notice that this new algorithm is almost the same as Algorithm 4 we proposed in Section 5.5 for accumulating intermediate results produced by FD coefficient multipliers. Indeed, we can consider Algorithm 5 as a generalized version of Algorithm 4: The same group-alignment technique is also applied as before. The main difference is that the input dataset is now sequentially fed into the only input port of the hardwired

summation unit. Correspondingly, internal data buffering is necessary here. However, the size of the input dataset could range from as few as three to as many as millions. Sometimes it is even impossible to buffer all elements of the dataset on a chip. To relieve this capacity constraint, we have to partition the input dataset into small group of summands and apply the group-alignment based summation method group by group. Therefore, the complex synchronization circuit is indispensable for automatic partitioning with only moderate idle cycles. We will revisit the detailed implementation of the summation circuit on FPGAs in Section 6.1.5.

Algorithm 5. Group-alignment based floating-point summation

Input: A set of n floating-point numbers s_1, s_2, …, s_n which are partitioned into groups with m summands

Output: The summation result $S = s_1 + s_2 + \cdots + s_n$

For k=1 to n/m

1. Split each summand s_i in this group into two portions as Fraction and Exponent. Restore the hiding bit in F_i to form the mantissa M_i.
2. Find the largest Exponent E_{max} within E_i
3. Calculate (E_{max} -E_i) for i=1 to m
4. Shift M_i to right by (E_{max} -E_i) bits, round the shifted fractions to nearest-even if necessary.
5. Sum up all shifted M_i using fixed-point arithmetic
6. Feed rounded M_{sum} together with E_{max} to the subsequent simplified floating-point accumulator

End

Normalize the final summation result to IEEE- compliant floating-point format.

6.1.4 Formal Error Analyses and Numerical Experiments

We predict intuitively that this group-alignment technique facilitates floating-point summations more accurately than standard arithmetic because almost all rounding errors arising in partial-sums within a group are eliminated except the last one in Step 6. To prove the correctness of this declaration, let's consider the group-alignment based summation with m summands. Because this algorithm utilizes the same monotone rounding scheme (round-to-nearest, towards-zero or away-from-zero) as standard floating-point arithmetic, averaging will have the same effect on rounding error propagations. Here, we analyze the worst case errors only.

Theorem 1: The group-alignment based floating-point summation algorithm can always result in a unique solution for the same summand group, regardless of the accumulation sequence.

Proof: In Step 4 of the algorithm, all summands within a group are aligned to the one with the largest exponent. Also, because a fixed-point accumulator is used in Step 5, which introduces no rounding errors in computations, a unique solution can always be achieved regardless of the sequence by which those summands are accumulated. ■

Let ε'_M be the unit rounding-off on a computer with the new summation unit. Its value now equals to half of the minimal quantity represented by the mantissa accumulator, which is set to be the same as standard floating-point arithmetic temporarily, i.e., 23 fraction bits for single-precision inputs or 52 bits for double-precision. Numerical error analyses show us the following result:

Theorem 2: The group-alignment based summation algorithm achieves similar, or even better, worst case relative and absolute errors than the sequential accumulation algorithm with standard floating-point arithmetic, depending on the choice of ε'_M.

Proof: After aligning all mantissas of summands to the one with the largest exponent, the absolute rounding error for each right-shifted floating-point summand except the largest one (which introduces no rounding error assuming the number itself is exact.) can be represented as:

$$\left|e'_i\right| \le \max\left\{\left|s_1\right|,\left|s_2\right|,\cdots,\left|s_m\right|\right\}\cdot \varepsilon'_M \tag{6.12}$$

The worst case absolute error of the summation after fixed-point aligned-mantissa accumulation is:

$$\left|e'_S\right| \le (m-1)\cdot \max\left\{\left|s_1\right|,\left|s_2\right|,\cdots,\left|s_m\right|\right\}\cdot \varepsilon'_M \tag{6.13}$$

The corresponding maximum relative error is:

$$\begin{aligned}\frac{\left|e'_S\right|}{\left|S\right|} &\le \frac{\max\left\{\left|s_1\right|,\left|s_2\right|,\cdots,\left|s_m\right|\right\}}{\left|s_1+s_2+\cdots+s_m\right|}\cdot(m-1)\cdot\varepsilon'_M \\ &\le \frac{\left|s_1\right|+\left|s_2\right|+\cdots+\left|s_m\right|}{\left|s_1+s_2+\cdots+s_m\right|}\cdot(m-1)\cdot\varepsilon'_M\end{aligned} \tag{6.14}$$

Comparing Expression (6.13) and (6.14) with (6.3) and (6.4), we conclude that Theorem 2 holds. ■

Observing Expression (6.14), we note another efficient method to further improve error bounds: It is convenient to decrease ε'_M on FPGA-based solutions by using more fraction bits to represent/manipulate those aligned summands and corresponding fixed-point summations. On the contrary, this approach is painful on commodity computers because users have to simulate high-accurate floating-point operations (double-extended or double-double) by software subroutines [81].

Table 13. Errors for the New Summation Algorithm

Fraction bits	Maximum/Average absolute error (Condition number)	Maximum relative error (Condition number)
23	5.14e-7/9.03e-8(73.5)	0.143 (4.13e+6)
24	2.68e-7/4.94e-8(1.58)	0.0857 (4.13e+6)
25	1.08e-7/1.72e-8(1.98)	0.0857 (4.13e+6)
26	5.22e-8/2.77e-9(4.51)	0.0182(5.97e+6)
27	1.86e-8/1.78e-9(2.49)	6.85e-4(1.71e+5)
28	7.45e-9/5.06e-10(2.88)	1.08e-4(1.43e+5)
29	3.73e-9/1.36e-10(2.06)	7.65e-5(2.35e+5)
32	2.33e-10/2.24e-12(16.72)	2.13e-6(3.25e+4)
Single-precision	5.29e-7/4.02e-8(1.45)	0.0857(4.13e+6)

A MATLAB program is used in this work to demonstrate error properties of our new summation algorithm. We first create one million groups of single-precision floating-point operands as input data sets, each of which contains ten uniformly-distributed random summands ranging from -0.5 to 0.5. The group-alignment based summation is executed for each input data set and the worst case absolute and relative errors are recorded and compared with results produced by the sequential accumulation algorithm with standard single-precision arithmetic. Using double-precision arithmetic results as a reference, Table 10 lists the recorded maximum/average absolute errors and maximum relative errors for the new summation algorithm utilizing different fraction bits. Relative condition numbers of summations are also calculated to show their impacts on final results. Important observations are listed as follows:

- The same conclusion as Theorem 2 regarding the worst case/average absolute errors can also be drawn from the first column in Table 10. Furthermore, these errors decrease proportionally to the number of fraction bits we adopted. This observation correlated well with the error expression (6.13).

- Numerical stability theory [82] tells us that:

$$\left|Output\ \ relative\ \ error\right| \le Condition\ \ number \times \left|Input\ \ relative\ \ error\right| \qquad (6.15)$$

  We note that catastrophic cancellations happened in those listed worst cases in column two because of large condition numbers. Initial relative errors hidden in the input data set are magnified dramatically so that error digits in solutions are now much closer to the most significant digit. Here, we cannot observe a clear relation between errors and the number of fraction bits as in column 1. The reason is that these bad-conditioned cases are still far from the worst ones so that most details are hidden by the "less than or equal to" sign in Expression (6.14). Indeed, we can easily create a floating-point data set with all digits of their summations contaminated.
- Comparatively, condition numbers listed in column one are all moderate. An intuitive explanation to this coincidence is that when condition number of summation is small, most significant operands have the same sign so that the absolute error bound in expression (6.13) is tight. However when condition number of summation is large, results are much smaller than the largest operands. Correspondingly, those significant operands tend to have opposite signs and cancel others.
- Although the new summation algorithm has improved worst-case error bounds, it doesn't mean that this approach can always produce better results for every input data set. For example, the average absolute error for our algorithm with 23 fraction bits is about twice as bad as the conventional sequential accumulation approach using single-precision arithmetic. Indeed, it is a little unfair to compare these two cases because the conventional implementation of single-precision floating-point addition in general has three more guarding bits.

6.1.5 Implementation of Group-Alignment Based Summation on FPGAs

Using the simplest single-precision floating-point accumulator as an example, we introduce in detail our implementation of the group-alignment based summation algorithm on FPGAs. Extending this design to double-precision or extended-precision is

easy and straightforward with nearly the same computational performance if appropriate pipelining stages were inserted. An entry-level Virtex II Pro evaluation board [81] is used as the target platform. The software development environments are Xilinx ISE 7.1i and ModelSim 6.0 se. Figure 39 shows the hardware structure of the summation unit.

The following features distinguish this design from others [78] [79]:

- To be compatible with conventional numerical computing software, the inputs and outputs of our summation unit are all in floating-point representations. But almost all internal stages use fixed-point arithmetic to save hardware resources as well as pipelining stages.
- Floating-point operands are fed into the single input port sequentially at a constant rate. Two local feed-back paths connect outputs of the single-cycle fixed-point exponent comparator and the mantissa accumulator with their inputs respectively, and so will not result in pipeline stalls.
- With the help of two external signals marking the beginning/end of input groups or datasets, our design can always achieve the optimal sustained speed without the knowledge of summation length. Furthermore, multiple sets of inputs can be accumulated consecutively with only one pipeline stall between them.
- The synchronization circuit automatically divides a long input dataset into small groups to take advantage of the group-alignment technique. Grouping is transparent to exterior and will not cause any internal pipeline stalls.
- The maximum size of a summand group is set to 16 in our implementation so that the corresponding group FIFO can be implemented efficiently by logic slices. The size of a group can also be easily reduced or enlarged to achieve better performance for particular problems.

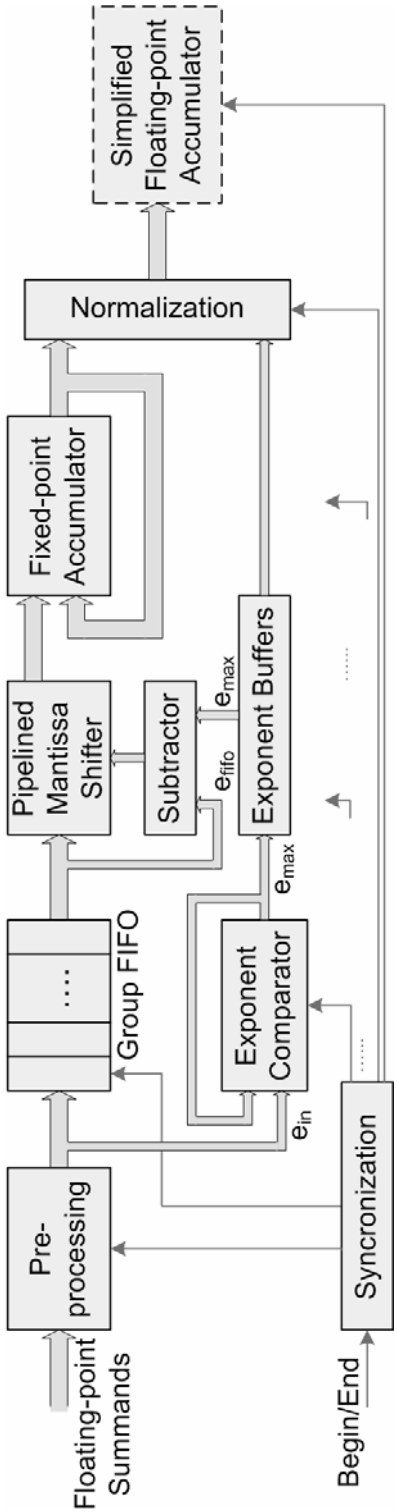

Figure 39. Structure of Group-Alignment Based Floating-Point Summation Unit

- Once the group FIFO contains a full summand group or the summation-ending signal is received, the synchronization circuit commands the exponent comparator to clear its content after sending the current value to the maximum exponent buffer. Starting from the next clock cycle, mantissas in FIFO are shifted out sequentially. Their exponents are also subtracted from the current maximum exponent one by one to produce differences for the pipelined mantissa shifter. While at the mean time, the next group of operands is moved in to fill vacant positions.
- The word-width of the barrel shifter is set to 34 bits (32 fraction bits) conservatively, so the fixed-point accumulator needs four more bits to prevent possible overflow. However, according to error analysis in Table 1, a 30-bit accumulator with 24 fraction bits is enough to achieve similar accuracy as the standard single-precision floating-point arithmetic.
- The single-cycle 38-bit group mantissa accumulator becomes the performance bottleneck of our design, preventing us from further improving the clock rate applied to the summation unit. Instead of using the costly Wallace-tree adder to remove the carry-chain from the accumulator's critical path [79], we simply disassemble the large fixed-point unit into two smaller ones. With their respective integer bits to prevent overflow, the resulting two 21-bit fixed-point accumulator now can work at a much higher speed.
- The normalization circuit accepts outputs from the fixed-point accumulator(s) and the exponent buffer, and converts them into normalized floating-point format. It also consists of Leading-Zero-Detector (LZD) and pipelined left shifter as standard floating-point adder. However, because the data throughput of this stage is at least half of all front-end circuits, a more economic implementation can always be achieved.
- For long summations with multiple summand groups, another group summation stage using conventional floating-point adder is required to accumulate all group partial-sums. Because its data throughput is just 1/16 of the front-end, pipelining inside the adder will not cause any data-dependency problem. Furthermore, the

foregoing normalization circuit and the floating-point adder can be combined to save a costly barrel shifter as well as other unnecessary logics.

- For applications where latency was an unimportant issue, On-chip block RAMs could also be used as FIFO to buffer a whole dataset instead of a small group of input data. Correspondingly, the synchronization circuit could be simplified considerably and the final floating-point accumulator is unnecessary.

Table 14. Comparison of Single-Precision Accumulators

	Group-alignment ①	Scheduling [78]	Delayed-addition [79]
Target device	Virtex II Pro	Virtex II Pro	Virtex-E
Area (Slices)	443 (716)	633 ($n=2^4$) ~900 ($n=2^{16}$) ②	1095 CLBs
Speed (MHz)	250	180 ($n=2^4$) ~160 ($n=2^{16}$)	150
Pipeline stages	14 (23)	20	5 ③
Latency (cycles)	n<16: 2n+12 (21) n>16: 44 (53)	$\le 3\cdot(n+20\log_2 n)$	5 + 46ns
Numerical accuracy	Proved	Guaranteed ④	Not guaranteed④

① Two numbers are listed at some places in this column for without and (with) the final group summation stage.
② One SRAM block is required for data buffering.
③ The final addition and normalization stage uses combinational logic, so is not pipelined.
④ The accuracy of [78] is guaranteed by the standard floating-point adder. In [79], the authors provided only simple numerical tests without rigorous proof.

Table 14 lists the performance of the new single-precision floating-point summation unit together with two other approaches proposed in [78] and [79]. They are

compared with each other based on sustained FLOPS performance, internal buffer requirements, latencies, etc. We observe that this new floating-point/fixed-point hybrid summation unit can provide much higher computational performance, less FPGA resource occupations, as well as more practical latency than previous designs. Furthermore, choosing more fraction bits for the fixed-point accumulator consumes negligible additional RC resources, but can significantly improve numerical error bounds of the summation.

Although the aforementioned summation algorithm can always provide similar or better absolute and relative error bounds than standard floating-point arithmetic, it still cannot avoid the occurence of catastrophic cancellation in some worst cases. Indeed, we can easily cook up floating-point data sets, where the initial relative errors hidden in inputs are magnified dramatically so that all digits of the final result are contaminated. The only way to completely eliminate inevitable cancellations is to use the “exact summation” approach [82]. Assuming that all floating-point inputs are represented exactly, rounding error happens when the word width of an accumulator is not enough to contain all effective binary bits of intermediate results. This method adopts an extreme solution to address this problem: It converts all floating-point inputs to ultra-wide fixed-point format so that fixed-point arithmetic can be used to reach an error-free solution. After that, a normalization stage is utilized to round the solution to appropriate floating-point format. A careful analysis shows us that nearly 300 binary bits is necessary to represent a single-precision floating-point number in fixed-point format, or over 2000 bits for double-precision cases. Even if such an ultra-wide register is acceptable, the underlying huge shifting/alignment stage and the unavoidable carry-chain of the fixed-point accumulator will pose a severe performance bottleneck to this approach. Indeed, to the best of our knowledge, there is no actual attempt to use this method in practice.

It is possible to construct an error-free floating-point summation unit on the proposed FPGA-enhanced computer platform. However, this unit would be much more consumptive and significantly slower than standard floating-point arithmetic. For some special cases where extremely accurate or even exact solutions are mandatory,

constructing such a rounding-error-free floating-point summation unit would be still worthwhile.

6.1.6 Accurate Vector Dot-Product on FPGAs

Given two column vectors of floating-point numbers $A = [a_1, a_2, \cdots, a_n]^T$ and $B = [b_1, b_2, \cdots, b_n]^T$, we want to accurately calculate their inner product:

$$C = A^T B = \sum_{i=1}^{n} a_i \cdot b_i \tag{6.16}$$

where the superscript "T" stands for the transpose of a vector or matrix. By accuracy, we mean better numerical error bound than the results produced by direct calculations using standard floating-point arithmetic.

We can easily observe that the only difference between summation and dot product is those element-wise multiplications, and so the group-alignment based summation technique we proposed above can also be applied here for accurate solutions of vector dot-product. However, these multiplications introduce a new problem. To expose this potential problem, let's consider the simplest case: the dot-product of two two-element vectors $A = [a_1, a_2]^T$ and $B = [b_1, b_2]^T$. Starting form Lemma 1, we have:

$$\begin{aligned}
&fl\big(fl(a_1 \times b_1) + fl(a_2 \times b_2)\big) \\
&= fl\big((a_1 \times b_1)(1+\varepsilon_1) + (a_2 \times b_2)(1+\varepsilon_2)\big) \\
&= \big((a_1 \times b_1)(1+\varepsilon_1) + (a_2 \times b_2)(1+\varepsilon_2)\big)(1+\varepsilon_3) \\
&= a_1 \times b_1 + a_2 \times b_2 + (a_1 \times b_1)\varepsilon_1 + (a_2 \times b_2)\varepsilon_2 + (a_1 \times b_1)\varepsilon_3 \\
&\quad + (a_2 \times b_2)\varepsilon_3 + (a_1 \times b_1)\varepsilon_1\varepsilon_3 + (a_2 \times b_2)\varepsilon_2\varepsilon_3 \\
&\approx a_1 \times b_1 + a_2 \times b_2 + (a_1 \times b_1)(\varepsilon_1 + \varepsilon_3) + (a_2 \times b_2)(\varepsilon_2 + \varepsilon_3)
\end{aligned} \tag{6.17}$$

After ignoring high-order rounding error terms in Equation (6.17), we can observe that numerical errors produced by multiplications ($\varepsilon_1, \varepsilon_2$) have similar magnitude as addition error (ε_3), and so should also be take into consideration for accurate solutions. Specifically, we cannot simply round the product of each pair of vector elements to standard floating-point format as we used to do on commodity CPU based general-

purpose computers. Some CPUs do provide so-called “Fused Multiply-Addition (FMA)” unit/instruction, where a floating-point accumulator is placed adjacent to the multiplier so that the rounding error introduced by multiplications could be called off. However, most software compilers do not yet support this function because it tends to complicate instruction scheduling, and may eventually slowdown the execution.

The group-alignment based floating-point summation unit as shown in Figure 38 could be easily extended to vector norm or dot-product unit by attaching a simplified floating-point multiplier to its input port. This multiplier accepts two standard floating-point operands at each clock cycle; normalizes them; and multiplies their mantissas. Then, the product and the sum of exponents are fed to inputs of following summation unit with all post-processing stages eliminated. For obtaining an accurate dot-product result, all effective bits of input mantissa products should be kept so that the numerical errors produced by multiplications ($\varepsilon_1, \varepsilon_2$) could be removed. In the mean time, the word-width of the barrel shifter and the fixed-point accumulator in the summation unit should also be extended correspondingly for better numerical error bound.

As we introduced before, the implementation of the Time Domain or Frequency Domain Finite Difference (FDTD or FDFD) computing engine could also profit from this technique by replacing the conventional costly floating-point adder tree with a group-alignment based summation unit. Moreover, the same technique can be applied to other linear algebra routines such as matrix-vector multiply, matrix-matrix multiply, etc. to efficiently decrease FPGA resource occupations and reduce pipeline stages without negative impact on computational performance or numerical accuracy. We will present our related works in following sections.

6.2 Matrix-Vector Multiply on FPGAs

The operations of floating-point matrix-vector multiply (GEMV) is defined as:

$$y_i = \sum_{j=0}^{n} A_{ij} \cdot x_j \tag{6.18}$$

Where A is a dense $n \times n$ matrix; x and y are two $n \times 1$ vectors.

After decomposing matrix A into n row vectors, the matrix-vector multiply can be treated as n dedicated vector dot-products. Correspondingly, the FPGA-based matrix-vector multiply engine can be constructed easily as a straightforward extension of the dot-product unit we proposed in Section 6.1. We already know that the main factor restricting the computational performance of pipelined summation or dot-product units is the contradiction between the long pipelines required for high throughput and data dependency among neighboring calculations. Specifically for the problem we considered, when the dimension of the matrix is larger than the depth of the pipelining stages of the FMA unit, adequate inherent low-level parallelism could be easily exploited so that simple scheduling can eliminate the potential data dependency problem. Suppose all elements of A, x, and y are saved in external memory (which is the case for most realistic numerical PDE problems). Because x is used as the only common column vector, there would be at least $\left(n^2 + 2n\right)$ memory accesses and $\left(2n^2\right)$ floating-point operations in total. The ratio between external memory accesses and floating-point operations is nearly two, which reveals the memory-bandwidth-bounded property of this subroutine.

The only work we can find that discusses this topic is in [7], where an FPGA-based matrix-vector multiply unit was proposed and its sustained performance was analyzed and compared with the same subroutine operating on contemporary general-purpose computers. External memory bandwidth of a typical FPGA-based system is, in general, millions of words per second, which is at the same level as the data throughput of fully-pipelined floating-point arithmetic units such as multiplier or adder in FPGA. A few

parallel-running arithmetic units could easily saturate all available external memory bandwidth, leaving considerable FPGA resources unused.

Here, we try to utilize these unused FPGA resources for some useful work so that the computations of matrix-vector multiply could be benefited. First, a floating-point FMA can be constructed based on the group-alignment based summation unit we proposed in Section 6.1. This new FMA unit also works in pipelined manner accepting two operands $(A_{ij}\ and\ x_j)$ at each clock cycle. All intermediate results $A_{ij} \cdot x_j\big|_{j=0\cdots n-1}$ for each y_i are buffered as one group of data using on-chip SRAM blocks. Correspondingly, the synchronization circuit of the summation unit is simplified significantly and the final floating-point accumulator is eliminated. The relatively long start-up latency cause by row buffering is leveraged because of the relatively large matrix size, so that the computational performance only drops slightly. Although this new approach doesn't alleviate the memory bandwidth bottleneck, it can evaluate y_i to higher accuracy at nearly the same speed by simply extending word width of the barrel shifter and the fixed-point accumulator. Comparatively, standard high-accurate floating-point arithmetic units are costly on FPGAs with considerably degraded performance. High-accurate result is always preferable when matrix A is bad-conditioned. Scientists [83] proposed pure software methods to achieve higher accurate results (double-extended, double-double, or quadruple precision) using only standard floating-point arithmetic (single or double precision) at the cost of 4~10 times of slowdown. Correspondingly, we can say that our new approach significantly improves the computational performance of high-accurate matrix-vector multiply in an indirect way.

Because the memory bandwidth limitation is always present, preventing us from further improving the sustained FLOPS performance of this subroutine, one of our choices is to construct an appropriate data buffering system to attain the most from the limited resources. Specifically, the data dependency existing in summations of consecutive products is the only issue that could be addressed by the buffering system. Assume the matrix is large and can only be stored in external memory. For small problems where in-chip RAM blocks could accommodate all elements of at least one

input/output vector, we have two choices to construct the data buffering system: The first one is to calculate matrix-vector multiply in row order just as Equation (6.18) implies. Here, Matrix entries are read into FPGA row by row and multiplied by the input vector x, whose elements are all buffered inside FPGA's RAM blocks. Multiple RAM blocks can support multiple operands accesses from x simultaneously so that a number of multipliers can work in parallel if the accumulative external memory bandwidth is wide enough for reading in the same number of operands from the matrix. The corresponding FPGA-specific hardware algorithm is listed in Algorithm 6. Figure 40 shows the diagram and dataflow of the FPGA-based hardware implementation.

Algorithm 6. Matrix-vector Multiply in row order

Input: $n \times n$ dense matrix A saved in external memory

$n \times 1$ input vector x saved in s SRAM blocks in FPGA chip. The number s block contains vector entries $i, i+s, i+2s, \cdots, i+(n/s-1)$.

Output: $y = A \cdot x$

For i = 1 to n

For j = 1 to n/s

Read in matrix entries $A_{i,(j-1)*s+1:j*s}$ from external memory

For k =1 to s do parallel

$p_k = A_{i,(j-1)*s+k} \cdot x_{(j-1)*s+k}$

End For

$q_j = \sum_{k=1}^{s} p_k$

End For

$y_i = \sum_{j=1}^{n/s} q_j$

End For

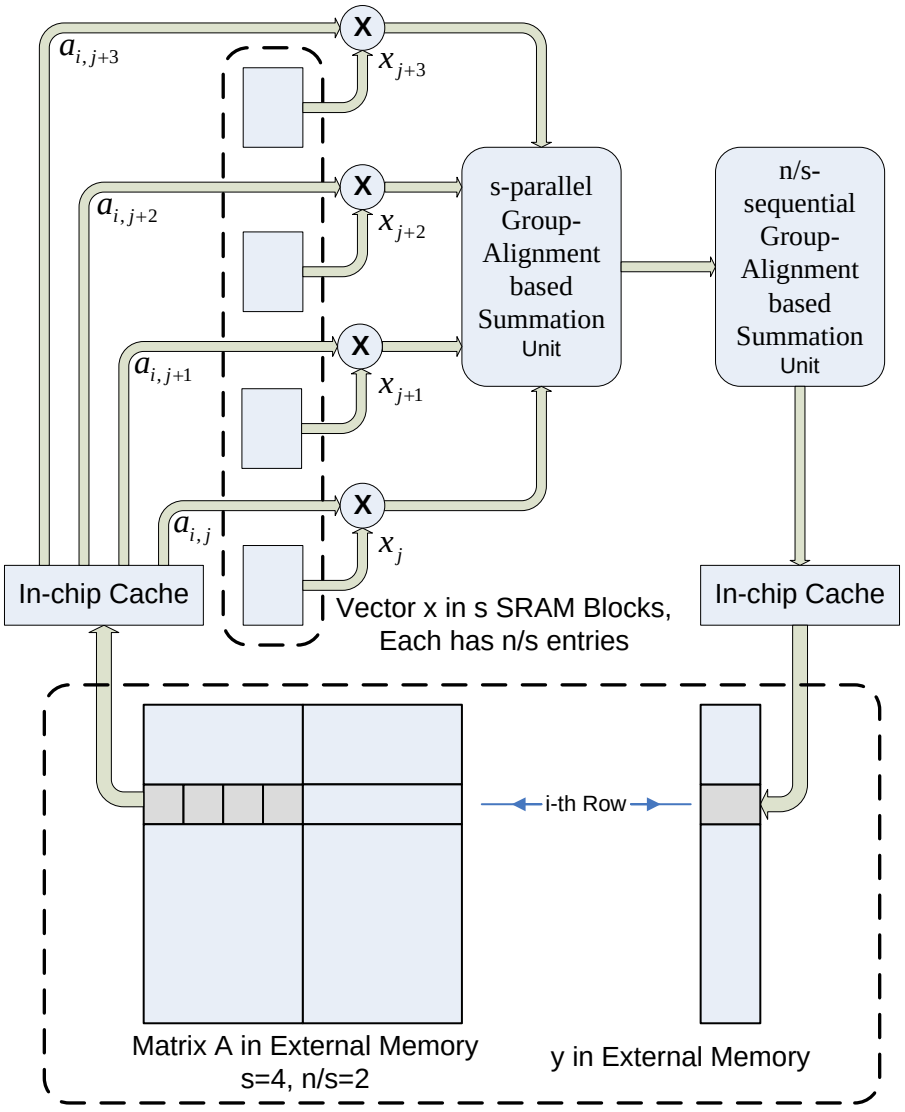

Figure 40. Implementation for Matrix-Vector Multiply in Row Order

We notice that the group-alignment based summation technique is applied twice here to address the data dependency problem: one in parallel followed by another in sequence. Although a little costly, this approach does simplify the design for the buffering circuit as well as internal data flow: the input vector buffer always works in read-only mode and there are no feed back data paths in this design.

The other data buffering scheme is based on the fact that matrix-vector multiply can also be represented as summation of n scaled column vectors of A as the following equation shows:

$$y = A \cdot x = \sum_{j=0}^{n} A_j \cdot x_j \tag{6.19}$$

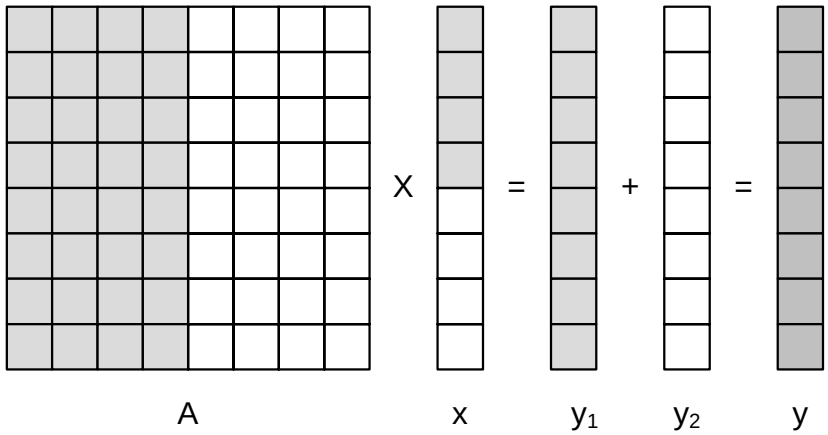

Figure 41. Matrix-Vector Multiply in Column Order

Figure 41 depicts the computational scheme of this approach. Here, all elements of the resulting vector y have their place inside FPGA's RAM blocks. The matrix is stripped into blocks of size $n \times s$, and the number of row entries in each block s is selected carefully so that all of them can be accessed simultaneously from external memory. Correspondingly, there are s parallel-running multipliers integrated inside the FPGA device. Although distributed registers are the best place to hold those vector entries, internal RAM block-based input buffer may still be necessary to hide access lags of external SDRAM modules. All product values generated at the same clock cycle together with the relevant partial sum value read from vector y are summed up to update the same y entry. The troublesome data dependency problem is eliminated easily by this simple scheduling because consecutive summations are now for different y entries. Compared with the previous approach, only one parallel group-alignment based summation unit is needed, but one data feed-back loop is introduced for updating the

output vector. On the whole, FPGA-based implementations for these two approaches will have similar computational performance with similar programmable hardware resources consumed. Algorithm 7 is the corresponding FPGA-specific hardware algorithm. The block diagram and data flow of its FPGA-based hardware implementation is depicted in Figure 40.

Algorithm 7. Matrix-vector Multiply in column order

Input: $n \times n$ dense matrix A saved in external memory

$n \times 1$ input vector x saved also in external memory

An in-chip dual-port SRAM as working space for the output vector y.

Output: $y = A \cdot x$

Set all entries in vector y to be zero

For j = 1 to n/s

Read in vector entries $x_{(j-1)*s+1,\, j*s}$ from external memory

For i = 1 to n

1. Read in matrix entries $A_{i,(j-1)*s+1:\, j*s}$ from external memory
2. For k =1 to s do parallel
3. $p_k = A_{i,(j-1)*s+k} \cdot x_{(j-1)*s+k}$
4. End For
5. $y_i = y_i + \sum_{k=1}^{s} p_k$

End For

End For

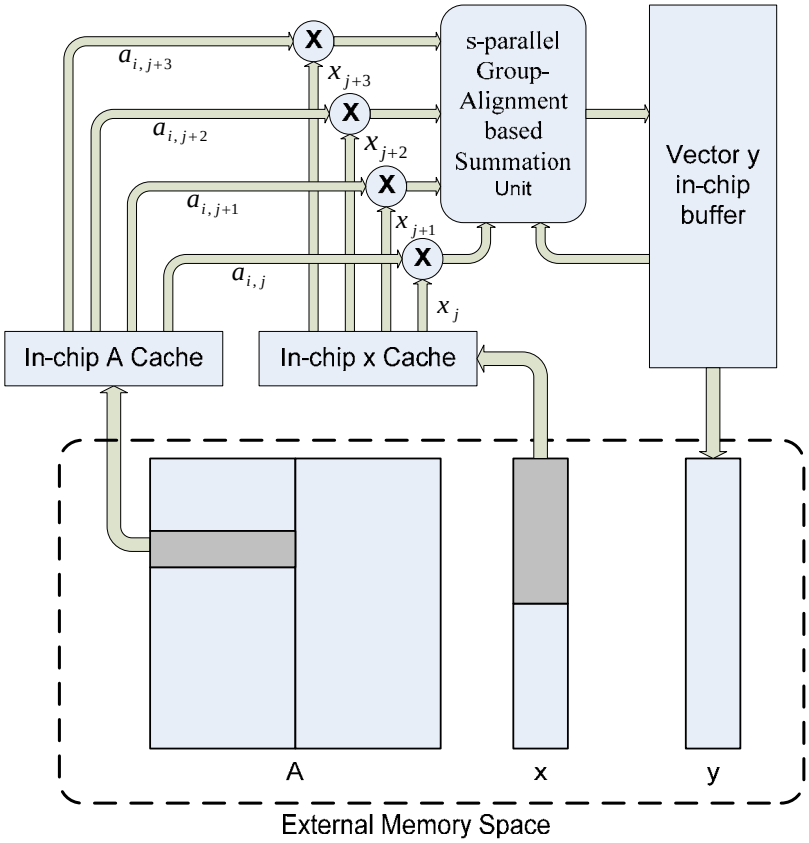

Figure 42. Implementation for Matrix-Vector Multiply in Column Order

The second approach can be easily extended to handle large matrix-vector multiply cases, where the input/output vector contains too many entries to be accommodated inside internal RAM blocks. The output vector y is now saved in external SDRAM modules together with the matrix. (Sometimes, we are lucky to have external SRAM modules integrated on board, which can be used to save the vector.) Every partial matrix-vector multiply related to one matrix strip requires access to this output vector twice in order to have all of its entries updated. As we introduced above, the computational performance of matrix-vector multiply is memory-bandwidth bounded, and so this memory access overhead would result in considerable performance degradation if the width of each matrix strip was narrow. For example, if we had only 4 row entries for each matrix strip, an additional 50 percent memory accesses would be

introduced, and so the sustained performance of the hardwired computing engine would be degraded by one third. Adopting a wide matrix strip would amortize this overhead and significantly improve the situation. However, the more matrix strip row entries, the larger parallel summation unit would be. On the other hand, this approach provides users with an effective way to maximize the utilization of FPGA resources for useful work.

6.3 Dense Matrix-Matrix Multiply on FPGAs

Given two $n \times n$ dense matrix A and B, the operations of matrix-matrix multiply $C = A \cdot B$ are defined as:

$$C_{ij} = \sum_{k=0}^{n} A_{ik} \cdot B_{kj} \tag{6.18}$$

In contrast to vector dot-product or matrix-vector multiply, matrix-matrix multiply is a computation-bounded subroutine. For the ideal case where FPGA's internal RAM blocks could accommodate all matrix elements of A, B, and C, there would be $\left(2n^2\right)$ memory accesses and $\left(2n^3\right)$ floating-point operations in total. So the ratio between external memory accesses and floating-point operations is proportional to n, which reveals excellent data reusability, especially when n is large. In theory, such computation-bounded subroutines could always put onboard FPGA devices into full play, and so achieve much higher sustained computational performance than commodity CPUs.

In reality, only a small portion of matrix entries could be saved in-chip. For these cases, block matrix-matrix multiply scheme is the most popular choice, although it may introduce considerable additional external memory accesses to deal with intermediate results. For example, if three were only enough in-chip memory spaces for buffering three $s \times s$ blocks of matrix entries, we need to read in $2 \times s^2$ matrix entries from A and B for every $2 \times s^3 + s^2$ floating-point computations. (Here, we ignore purposely the memory access for saving resulting matrix elements of C because this workload would be amortized if the matrix size is large.) So, the total number of external memory

accesses of this subroutine is $\left(\frac{n}{s}\right)^3 \cdot 2s^2 = \frac{2n^3}{s}$, which depends on s and is much larger than the ideal case.

A successful design of matrix-matrix multiply on FPGA-based platform implies building an appropriate internal data buffering subsystem utilizing FPGA's abundant in-chip SRAM blocks and distributed registers. Fully exploiting data/computation locality of the numerical subroutine is pivotal to its success. Specifically, large block size s is always preferred to eliminate additional external memory accesses. There has been previous research work, discussing FPGA-based implementation of matrix-matrix multiply. In [7], the authors investigated the trends in sustainable floating-point performance of some basic BLAS subroutines for CPU and FPGA. It mainly concentrated on theoretical peak performance analysis, but not on detailed implementation. In [84], the authors designed a two-dimensional processor mesh based on conventional systolic matrix-matrix multiply hardware algorithm. Each processor node has its own multiply-accumulate unit (MAC) and local memory space and is responsible for the computations of a consecutive block of matrix C. Elements of matrix A and B are fed into the computing engine from boundary nodes, and traverse internal processors via in-chip interconnection paths. The disadvantage of this design is that all boundary processing nodes import/export operands from/to the outside world so that the number of input/output ports it requires would be large. In [85] [86], the authors proposed a one-dimensional processor array to address this problem. Only the first processing node read matrix A and B from external memory. These values then traverse all inner nodes in the linear array to participate in all related computations. Once the procedure is finished, each processing node simply transfers its local results of matrix C to its neighbors. These results are finally written back to external memory via the first processing node. One common problem of these designs is that they all require complex control logics for coordinating communications and interactions among multiple processing nodes. For example, about 50 percent of FPGA's programmable hardware resources are consumed for this purpose in the implementation in [86].

In our work, we proposed a new solution for matrix-matrix multiply, which could achieve much simpler hardware implementation on an FPGA-enhanced computer platform. This new matrix-matrix multiply unit doesn't employ multiple processing nodes but does use a large computing engine with an array of standard floating-point multipliers followed by one parallel group-alignment based summation unit. A centralized data buffering subsystem is designed to read operands of matrix A and B from external memory, caching them in local memory, and distributing them to arithmetic units in correct sequence. Two crucial problems have to be addressed by the data buffering circuit: First, good scalability is required so that the block size s could be modified easily to ensure the computation-bounded nature of the underlying blocked matrix-matrix multiply schemes; Second, because of the parallel running of multiple arithmetic units, there are concurrent accesses to multiple operands in one data block. One large RAM module provides only one or two read/write ports, and so is inapplicable here. Multiple distributed small RAM blocks would be an appropriate choice. Only simple control logics are needed here to coordinate their operations. The FPGA-specific hardware implementation of matrix-matrix multiply is shown in Figure 43.

In this figure, s is simply set to 16. Each matrix block would have $16 \times 16 = 256$ double precision entries. The fully-pipelined computing engine contains 16 modified floating-point multipliers and a large parallel summation unit, and so can finish 16 multiplications and 16 additions at each clock cycle. If we set the computing engine to operate at 200MHz, its sustained computational performance would be $32 \times 200M = 6.4G$ FLOPS. By varying the value of s, we can easily change the computational performance as well as the size of the computing engine.

Multiple matrix C blocks are accommodated inside the in-chip output buffer, which can be efficiently constructed with FPGA's in-chip SRAM blocks. This output buffer circuit has two double word (64-bit) data ports with dedicated read/write logics and address pins. One of them is connected to an input port of the parallel summation unit for feeding in previous partial sums of C entries. The other one is for writing back those updated partial-sums produced by the summation unit. As we will see later,

concurrent read and write addresses to the same memory space will always have a fixed distance, and so will not introduce any conflicts.

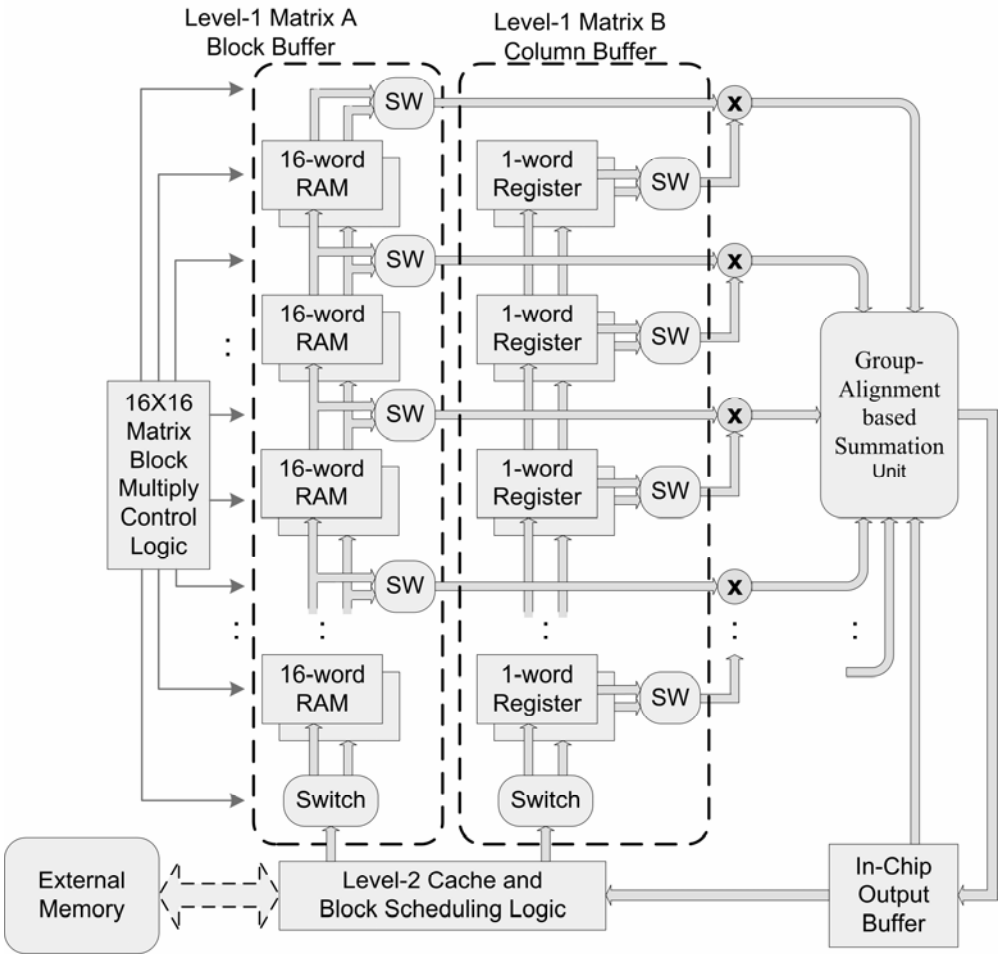

Figure 43. Blocked Matrix-Matrix Multiply

A two-level caching circuit is employed to efficiently buffer operands read from matrix A and B in-chip. 16 dedicated small RAM pieces constitute the first level cache for matrix A. There are 256 matrix entries (one 16×16 block) saved temporarily inside this caching circuit. So each RAM piece contains only 16 column entries and can be implemented efficiently with distributed register blocks in FPGA. This caching circuit

has two working modes: In the refresh mode, all of those small memory pieces are interconnected to form a cascaded FIFO with 16 levels; entries of a matrix block are read out from the second level cache in column order and pushed into the FIFO structure from its input port at the bottom. We need a total of 256 push cycles to update all entries. In computation mode, all these RAM pieces work independently and their access is controlled by a unique 4-bit addressing logic. At each clock cycle, 16 commonly-addressed entries buffered in these RAMs, who all come from the same row of the matrix block, are accessed simultaneously while providing operands of matrix A to 16 multipliers. The access of the matrix block will repeat for 16 rounds, with 16 clock cycles for each round. So in total, the computation mode also last for 256 cycles.

16 dedicated one-double-word registers are used to construct another 16-level cascaded FIFO for the first-level caching of 16 matrix B entries (one column of a matrix B block). They also have two working modes: the refresh mode and the computation mode. However, the switching speed of this caching circuit is 16 times faster than matrix A buffer. To hide the refresh cycles of both data buffers, two identical caching circuits are employed. They work in a swapping manner to overlap the refreshing and computation cycles. Once updated, the operands in matrix B buffers remain unchanged during the next 16 clock cycles providing another group of operands to multipliers. All 16 products together with the old partial-sum read out from the output buffer, are then fed into the group-alignment based parallel summation unit simultaneously as a group of summands. The summation result is written back to the output buffer via another data port. No data/computation dependency exists in this implementation because consecutive summations are for different C entries.

We already know that the computing engine needs 256 clock cycles to finish the computations of two 16×16 matrix blocks multiply. On the other hand, the 256-entry first-level matrix A cache updates its contents every 256 cycles, and in the same time period, the contents of the 16-entry matrix B cache have been changed for 16 times. In order to keep the computing engine operating at 200MHz, we need a memory channel that can afford a data transferring rate at a total of 400M double words per second

(3.2GByte/s), which corresponds to the memory bandwidth provided by a 400MHz DDR-SDRAM module. Fortunately, it is not necessary for these memory channels to be external. In this design, we introduced another level of large-capacity cache structure to buffer multiple matrix blocks in-core. A simple block scheduling circuit is employed to coordinate the multiplication of large matrices with multiple 16×16 matrix blocks. We can simply follow the ordinary block matrix-matrix multiply algorithm as shown in Figure 44. The special data buffering scheme we adopted here can ensure that the whole block a_2 and the first column of block b_3 would be ready in-core immediately after the multiplication of $a_1 \cdot b_1$ so that the computations of $c_1 \Leftarrow c_1 + a_2 \cdot b_3$ could start without any pipelining stalls. Once the final results of a block in matrix C are obtained, we have to save them back to external memory. If the size of the matrices is large enough, this overhead would be considerably amortized.

Figure 44. Blocked Matrix-Matrix Multiply Scheme

From a perspective outside of the buffering system, the block size of matrix-matrix multiply is now enlarged because of the existence of the level-2 cache; correspondingly, the requirement for external memory bandwidth would decrease proportionally. This second-level cache circuit has four data paths connected to the first level matrix A cache, matrix B cache, matrix C output buffer, and the external memory channel, respectively. The block scheduling circuit ensures the transfer rates of the first two data paths being fixed at 200M double words per second to guarantee the full speed operation of the computing engine. The data rate of the external memory channel will be determined by

the number of matrix blocks buffered in this second-level cache circuit. We can choose to build the caching circuits with in-chip RAM blocks or onboard external SRAM modules depending on the available hardware resources at hand. For example, it is straightforward to build an in-chip second-level cache with 8192 matrix A or B entries. The block size now becomes 64 and we need 800MByte/s external memory bandwidth to keep the computing engine operating at its full speed. Or, if we have a 400MHz DDR-SDRAM channel (3200MByte/s memory bandwidth) and enough FPGA resources on board, we are allowed to construct a much larger computing engine (64 floating-point multipliers together with a 64-parallel summation unit) up to 4 times more powerful than the aforementioned design. The sustained computational performance would be 25.6G FLOPS, which is much higher than any existing commodity CPU.

7. CONCLUSIONS

7.1 Summary of Research Work

In this research work, by proposing new hardware-reconfigurable computer architecture and designing FPGA-specific software algorithms, we considerably accelerated the executions of several representative numerical methods on FPGA-enhanced computers. We successfully demonstrated the impressive computational potential of the newly-proposed FPGA-enhanced computer system, thereby proving the feasibility of utilizing FPGA resources to accelerate computationally-demanding and data intensive numerical computing applications. The following topics had been investigated systematically in this work:

- Research on "Hardware Architecture Model of FPGA-Enhanced Computers for Numerical PDE problems"

  Targeted at computationally-demanding and data intensive numerical PDE problems, a new computer architecture model named FPGA-enhanced Computers was proposed together with detailed implementations as a single workstation as well as a parallel cluster system. Working in a hardware-programmable/application-specific manner, the resulting FPGA-enhanced computer system could be implemented economically with low-cost COTS components, and can therefore achieve much better price-performance ratio with much lower power consumption. Also, it is consistent with the prevailing PC-Cluster system and is scalable to a large parallel system containing abundant reconfigurable hardware and memory resources. Consequently, a wide range of numerical algorithms/methods could be accommodated on such a system.

- Research on "Accelerating PSTM Algorithm on the Proposed FPGA-Enhanced Computer Platform"

  Pre-Stack Kirchhoff Time Migration (PSTM) is one of the most popular migration methods in the seismic data processing field. It represents a class of numerical

algorithms/methods that require extraordinary computer arithmetic units that are relatively slow, or even unavailable, on commodity CPUs. Here, an application-specific Double-Square-Root (DSR) arithmetic unit was built on the proposed FPGA-enhanced computer platform to accelerate the evaluation of the algorithm's most time-consuming kernel subroutine without losing numerical accuracy. Because over 90 percent of CPU time is consumed by billions of iterations of the short kernel subroutine when operating on commodity CPUs, this new FPGA-based approach could operate more than 10 times faster than contemporary general-purpose computers, allowing people to produce a satisfying underground image much faster.

- Research on "High-accuracy Floating-point Summation Algorithms on FPGA-enhanced computers"

Floating-point summation is one of the most important operations in numerical computations. An FPGA-based hardware algorithm for accurate floating-point summation is proposed using the group-alignment technique. The corresponding fully pipelined summation unit is proven to provide similar, or even better, numerical errors than the standard floating-point arithmetic based sequential addition method. Moreover, this new design consumes much less FPGA resources, as well as pipelining stages, than other existent designs, and it achieves sustained working speed at one summation per clock cycle with only moderate start-up latency. This new technique can also be utilized to accelerate executions of other linear algebra subroutines as well as finite difference methods on FPGAs. The possibility of constructing an error-free floating-point summation unit on the RC platform is also investigated.

- Research on "Optimized Finite Difference Schemes with Finite Accurate Coefficients"

Based on maximum-order FD schemes whose coefficients are determined by cancelling as many lower-order Taylor expansion terms as possible, we proposed a new class of optimized finite accuracy FD schemes as well as heuristic algorithms to determine their FD coefficients. This new class of FD schemes has identical

computational workload and similar numerical accuracy as conventional high-order FD schemes, and would therefore be insignificant for commodity CPUs. However, its implementation on an FPGA-enhanced computer platform would be superior with much higher computational throughput and less FPGA resources consumption.

- Research on “Finite Difference Wave Equations Modeling on FPGA-enhanced Computers”

Adopting appropriate temporal and spatial FD schemes and applying results of aforementioned research works, the execution speed of realistic 2D or 3D seismic wave modeling problems is improved significantly on the proposed FPGA-enhanced computer platform. Efficient memory hierarchy and appropriate numerical algorithms are adopted to alleviate the memory bandwidth bottleneck of this specific numerical PDE problem.

- Research on “BLAS subroutines on FPGA-enhanced Computers”

The most time-consuming step of FEM is the solution of the large linear system equations generated from discretized PDEs. Basic Linear Algebra Subprograms (BLAS) are the standard toolkit necessary for users to solve linear equations. Our work aims to accelerate the executions of basic BLAS subroutines such as summation, dot-product, matrix-vector multiply, and matrix-matrix multiply on the FPGA-enhanced computer platform. Our efforts mainly concentrate on designing novel data buffering subsystem as well as suitable memory hierarchy to improve data reusability and save external memory access. By doing so, a wide range of scientific and engineering problems governed by partial differential equations could be accelerated on the proposed FPGA-enhanced Computer.

7.2 Methodologies for Accelerating Numerical PDE Problems on FPGA-Enhance Computers

In this section, we conclude conceivable methodologies of solving numerical PDE problems on an FPGA-enhanced computer. The essential purpose is to achieve higher sustained computational performance on FPGAs over commodity CPUs.

First of all, FPGA's computing power comes from the capability inherent in ASICs as efficient utilization of hardware resources. Unlike commodity CPUs, where a large portion of transistors are expended for providing program-controlled data flow, FPGA is capable of dedicating most of its in-chip programmable hardware resources for useful computations. By exploiting low level parallelism concealed in specific numerical methods/algorithms, a single FPGA device could accommodate a large computing engine consisting of tens, even hundreds, of similar or different arithmetic/function units. These hardwired units could be set to work in parallel for high accumulated performance or in a pipelined manner to achieve high data throughput. Furthermore, users could select to customize their own extraordinary arithmetic or function units for improved computational performance or hardware efficiency.

FPGA's In-System-Programmability (ISP) is pivotal to utilize its hardware resources for accelerating the solutions of numerical PDE problems. Such computing tasks are computationally demanding and data intensive. Their numerical solutions generally require a series of processing stages or multiple iterations with gradually improved simulation accuracy. Sometimes, initial trial-runs execute very rapidly, utilizing aggressive numerical methods. However, they are, in general, prone to convergence failure, and may even break down. Users have to seek the help of other robust but costly numerical methods. For example, we already know that seismic migration problems are governed by acoustic/elastic wave equations whose numerical solutions are in general time-consuming. Geophysicists may first try to attack a specific migration task using the relatively fast PSTM algorithm. After several iterations of inverse/forward procedures, if the migrated underground image is still unsatisfactory,

they are forced to resort to the robust but much more expensive Reverse Time Migration (RTM) algorithm, which is based directly on finite difference solutions of the original wave equations. However, if the parameters of underground media change abruptly, FD methods have to adopt excessive fine discretization steps for numerical stability, which, in turn, leads to unfeasible execution time. In these cases, FEM might be a relatively more efficient option because of its ability to follow complex boundaries and resolve minute geometrical features. Based on the results of our research work, all of these numerical methods can be accelerated effectively on the proposed FPGA-enhanced computer platform. Just as a large software package operating on general-purpose computers, users are now free to select different numerical algorithms/methods on the same hardware-programmable computer platform. The contents switching between different numerical methods on FPGAs costs only several seconds, thereby are negligible compared with the long execution time of realistic processing tasks.

Numerical methods/algorithms for PDE problems, in general, exhibit low FP-operation to memory-access ratio, require considerable memory space for intermediate results, and tend to perform irregular indirect addressing for complex data structures. These intrinsic properties inevitably result in poor caching behavior on modern commodity CPU-based general-purpose computers. Consequently, a significant gap always exists between their theoretical peak FP performance and the actual sustained Megaflops value. FPGA-enhanced computers are capable of reconfiguring memory hierarchy according to the requirements of specific problems. Because the clock frequencies applied to FPGAs and external memory modules are within the same range as hundreds of Millions Hz, we can treat all memory elements equally as a flattened memory space to simplify system architecture; or we can introduce complicated buffering structures or caching rules to further enhance data reusability and improve utilization of memory bandwidth.

There are mainly two error sources in numerical computations: the truncation error and the rounding error. Truncation error is the difference between the true result (for the actual input) and the result that was produced by algorithms/methods using exact

computer arithmetic. In most cases, truncation errors emerge due to numerical approximations such as truncating an infinite series, replacing derivative by finite difference, or terminating iteration before convergence. Numerical rounding error is the difference between the results produced by algorithms/methods using exact arithmetic and using finite-precision arithmetic. It is mainly due to inaccuracy in the representation of real numbers as well as the floating-point arithmetic operations on them. Numerical errors could be eliminated or at least significantly reduced by high-accuracy numerical algorithms/methods on general-purpose computers. However, the cost we pay for high-accuracy is more computational workload. For example, a numerical library called XBLAS consists of almost the same numerical subroutines as the BLAS library but uses increased floating-point working precision such as double-double, extended double, or quadruple precision. Subroutines in this library emulate high-accurate floating-point arithmetic using standard ones. Sometimes, they also have to compute correction terms in order to take into account the rounding errors accumulated during the computations, in other words, truncation terms that are normally ignored. Correspondingly, the execution speed of these XBLAS subroutines is, in general, tens of times slower than their siblings in BLAS. With the help of hardware-programmable FPGA resources, we can customize high-performance computing engines specified for high-order numerical methods so that truncation errors could be significantly reduced. Furthermore, we can construct our genuine high-accuracy floating-point arithmetic units to reduce numerical rounding errors with negligible speed penalties.

In summary, we believe and hope to convince others that the high computational potential of FPGA-enhanced computers would not only exercise a great influence on hardware architecture design of future computers, but also would have impact on numerical algorithms/methods when users try to take full advantage of FPGA's computational potential. We further boldly predict that such hardware-programmable resources would follow a similar path as floating-point arithmetic units: first working as an acceleration card loosely attached to a computer's peripheral bus, then coupled with

commodity CPU as coprocessor, and finally integrated into the same silicon chip with CPU cores, thereby becoming their indispensable component.

REFERENCES

[1] J. K. Costain, C. Coruh, Basic theory in reflection seismology, Elsevier Science, Amsterdam, Netherlands, 2004.

[2] Oz Yilmaz, S. M. Doherty, Seismic data analysis: Processing, inversion, and interpretation of seismic data, 2nd edition, Society of Exploration, Tulsa, OK, 2000.

[3] S. H. Gray, Y2K Review Article: Seismic migration problems and solutions, Geophysics, 66 (2001) 1622-1640.

[4] L. House, S. Larsen, J. B. Bednar, 3-D elastic numerical modeling of a complex salt structure, in: Expanded Abstracts of SEG 70th Annual Meeting, 2000, pp. 2201-2204.

[5] P. Moczo, M. Lucka, and M. Kristekova, 3D displacement finite differences and a combined memory optimization, Bulletin Seismological Society of America, 89 (1999) 69-79.

[6] S. Larsen, J. Grieger, Elastic modeling initiative, part III: 3-D computational modeling, in: Expanded Abstracts of SEG 68th annual meeting, 1998, pp. 1803-1806.

[7] K. D. Underwood, and K. S. Hemmert, Closing the gap: trends in sustainable floating-point BLAS performance, in: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 219-228.

[8] J. Makino and M. Taiji, Special-purpose Computers for Scientific Simulations: The GRAPE Systems, John Wiley & Sons, Hoboken, NJ, 1998.

[9] The GRAPE Project, GRAPE: A programmable multi-purpose computer for many-body simulations, <http://grape.astron.s.u-tokyo.ac.jp/grape/>.

[10] C. Cheng, J. Wawrzynek, and R. W. Brodersen, A high-end reconfigurable computing system, IEEE Design and Test of Computers, 22 (2005), 114-125.

[11] Starbridge, HC-62 board specifications, <http://www.starbridgesystems.com/>.

[12] Dini Group, Product Overview, <http://www.dinigroup.com/>.

[13] C. Petrie, C. Cump, M. Devlin, and K. Regester, High performance embedded computing using field programmable gate arrays, in: Proceedings of the 8th Annual Workshop on High-performance Embedded Computing, 2004, pp. 124-150.

[14] L. Gray, R. Woodson, A. Chau, and S. Retzlaff, Graphics for the long term: An FPGA-based GPU, <http://www.vmebus-systems.com/>, 2005.

[15] Cray, Cray XD1 datasheet, <http://www.cray.com/products/xd1/>.

[16] SGI, SGI RASC RC100 blade datasheet, <http://www.sgi.com/products/rasc/>.

[17] S.J.E. Wilton, Implementing Logic in FPGA memory arrays: Heterogeneous memory architectures, in: Proceedings of the IEEE International Conference on Field-Programmable Technology, 2002, pp. 142-149.

[18] C. Ebeling, D. C. Cronquist, P. Franklin, RaPiD-reconfigurable pipelined data path, in: Proceedings of the 6th International Workshop on Field-Programmable Logic, 1996, pp. 126-135.

[19] B. Fagin and C. Renard, Field programmable gate arrays and floating point arithmetic, IEEE Transactions on VLSI Systems, 2 (1994), 365-367.

[20] K. D. Underwood, FPGAs vs. CPUs: Trends in peak floating-point performance, in: Proceedings of the ACM/SIGDA 12th International Symposium on FPGA, 2004, pp. 171-180.

[21] P. Belanovic and M. Leeser, A Library of parameterized floating-point modules and their use, in: Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002, pp. 657-666.

[22] J. Liang, R. Tessier, and O. Mencer, Floating-point unit generation and evaluation for FPGAs, in: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003, pp. 185-194.

[23] A. A. Gaar, W. Luk, P. Y. Cheung, N. SHirazi, and J. Hwang, Automating customization of floating-point designs, in: Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002, pp. 523-533.

[24] 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 wrapping, in Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999, pp. 240-248.

[25] Maya B. Gokhale, Paul S. Graham, Reconfigurable computing: Accelerating computation with field-programmable gate arrays, Springer-Verlag, New York, 2005.

[26] Xilinx, Virtex-4 user guide, <http://www.xilinx.com>.

[27] V. Betz and J. Rose, Automatic generation of FPGA routing architectures from high-level descriptions, in: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2000, pp. 175-184.

[28] D. S. William. R. S. Austars, Towards an RCC-based accelerator for computational fluid dynamics applications, Journal of Supercomputing, 30 (2004) 239-261.

[29] R. N. Schneider, L. E. Turner, and M. M. Okoniewski, Application of FPGA technology to accelerate the Finite-Difference Time-Domain (FDTD) method,

in: Proceedings of the 10th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2002, pp. 97-105.

[30] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, Reconfigurable molecular dynamics simulator, in: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 197-206.

[31] Patterson and Hennessy, Computer Architecture: A Quantitative Approach. Third Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2000.

[32] C. He, M. Lu, and C. W. Sun, Accelerating seismic migration using FPGA-based coprocessor platform, in: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004, pp. 207-216

[33] D. Bevc, Imaging Complex structure with semi-recursive Kirchhoff migration, Geophysics, 62 (1997) 577-588.

[34] F. Audebert, 3-D pre-stack depth migration: Why Kirchhoff, Stanford Exploration Project, Report 80, 1994.

[35] C. Sun, and R.D. Martinez, Amplitude preserving 3D prestack time migration for V (z) media, in: Proceedings of the 64th Conference & Exhibition of the EAGE, 2002, pp.1124-1127.

[36] C. Sun, and R.D. Martinez, Amplitude preserving 3D prestack time migration for VTI media, First Break, 19 (2002) 618-624.

[37] F. Audebert, 3-D pre-stack depth migration: Why Kirchhoff?, Stanford Exploration Project, Report 80, 1994.

[38] D. Lumley, and B. Biondi, Kirchhoff 3D pre-stack time migration on the connection machine, Stanford Exploration Project, Report 72, 1991.

[39] J. E. Volder, The birth of CORDIC, Journal of VLSI Signal Processing, 25 (2000) 101-105.

[40] R. Andraka, A survey of CORIC algorithms for FPGA based computers, in: Proceedings of the 5th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 1998, pp. 191-200.

[41] K. Kota, and J. R. Cavallaro, Numerical accuracy and hardware tradeoffs for CORDIC arithmetic for special-purpose processors, IEEE Transactions on Computers, 42 (1993) 769-779.

[42] E. Antelo, M. Boo, J. D. Bruguera, and E. L. Zapata, A novel design of a two operand normalization circuit, IEEE Transactions on VLSI Systems, 6 (1998) 173-176.

[43] M. T. Taner and F. Koehler, Velocity spectra-digital computer derivation and application of velocity functions, Geophysics, 34 (1969) 859-881.

[44] C. Sun, H. Wang, and R. D. Martinez, Optimized 6th order NMO correction for long-offset seismic data, in: Expanded Abstracts of SEG 72nd Annual Meeting, 2002, pp. 2201-2204.

[45] W. C. Chew, Waves and fields in inhomogeneous media, IEEE Press, Piscataway, NJ, 1995.

[46] J. C. Strikwerda, Finite difference schemes and partial differential equations, Second Edition, Cambridge University Press, Cambridge, UK, 2004.

[47] I. R. Mufti, J. A. Pita, and R. W. Huntley, Finite-difference depth migration of exploration-scale 3-D seismic data, Geophysics, 61 (1996) 776-794.

[48] F. Bengt, Calculation of weights in finite difference formulas, SIAM Review, 40 (1998) 685-691.

[49] M. A. Dablain, The application of high order differencing for the scalar wave equation, Geophysics, 51 (1986) 54-66.

[50] E. Hairer, S. P. Norsett, and G. Wanner, Solving ordinary differential equations, Springer Press, New York, NY, 1991.

[51] R. P. Bordeling, Seismic modeling with the wave equation difference engine, in: Expanded Abstracts of Society of Exploration Geophysicists (SEG) International Exposition and 66th Annual Meeting, 1996, pp. 666-669.

[52] M. Bean, and P. Gray, Development of a high-speed seismic data processing platform using reconfigurable hardware, in: Expanded Abstracts of Society of Exploration Geophysicists (SEG) International Exposition and 67th Annual Meeting, 1997, pp. 1990-1993.

[53] J. R. Marek, M. A. Mehalic, and A. J. Terzouli, A dedicated VLSI architecture for Finite-Difference Time Domain (FDTD) calculations, in: Proceedings of the 8th Annual Review of Progress in Applied Computational Electromagnetic, 1992, 546-553.

[54] P. Placidi, L. Verducci, G. Matrella, L. Roselli, and P. Ciampolini, A custom VLSI architecture for the solution of FDTD equations, IEICE Transactions on Electronics, E85-C(2002) 572-577.

[55] R. N. Schneider, L. E. Turner, and M. M. Okoniewski, Application of FPGA technology to accelerate the Finite-Difference Time-Domain (FDTD) method, in: Proceedings of the 10th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2002, pp. 97-105.

[56] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, D. W. Prather, and M. S. Mirotznik, Hardware implementation of a three-dimensional finite-difference time-domain algorithm, IEEE Antennas and Wireless Propagation Letters, 2 (2003) 54-57.

[57] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport, An FPGA implementation of the two dimensional Finite Difference Time Domain (FDTD) algorithm, in:

Proceedings of the 12th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2004, pp. 213-222.

[58] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, P. F. Curt, and D. W. Prather, FPGA-based acceleration of the 3D Finite-Difference Time-Domain (FDTD) method, in: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 156-163.

[59] J. Virieux, P-SV wave propagation in heterogeneous media: velocity stress finite difference method, Geophysics, 51 (1986) 889-901.

[60] R. Clayton and B. Engquist, Absorbing boundary conditions for acoustic and elastic wave equations, Bulletin Seismological Society of America, 67 (1977) 1529-1540.

[61] G. Mur, Absorbing boundary conditions for the finite-difference approximation of time-domain electromagnetic field equations, IEEE transactions on Electromagnetic Computations, 23 (1981) 377-382.

[62] R. L. Higon, Absorbing boundary conditions for difference approximations to the multidimensional wave equation, Mathematical Computations. 47 (1986) 437-459.

[63] J. P. Berenger, A perfectly matched layer for the absorption of electromagnetic waves, Journal of Computational Physics, 114 (1994)185-200.

[64] I. Orlianski, A simple boundary condition for unbounded hyperbolic flows, Journal of Computational Physics, 21 (1976) 251-269.

[65] C. Cerjan, D. Kosloff, R. Kosloff, and M. Reshef, A nonreflecting boundary condition for discrete acoustic and elastic wave equations, Geophysics, 50 (1985) 705-708.

[66] D. R. Burns, Acoustic and elastic scattering from seamounts in three dimensions – a numerical modeling study, The Journal of the Acoustical Society in America, 92 (1992) 2784-2791.

[67] Xilinx, “ML401 Evaluation Platform User Guide”, <www.xilinx.com>.

[68] G. Marcus, P. Hinojosa, A. Avila, and J. Nolazco-Flores, A fully synthesizable single-precision, floating-point adder/substractor and multiplier in VHDL for general and educational use, in: Proceedings of the 5th International Caracas Conference on Devices, Circuits and Systems (ICCDCS), 2004, pp. 234-243.

[69] G. Chaltas and W. R. Magro, Performance analysis and tuning of LS-DYNA for Intel processor-based clusters, in: Proceedings of the 7th International LS-DYNA Users Conference, 2002, pp. 122-132.

[70] W. H. Press, B. P.Flannery, S. A.Teukolsky, and W. T.Vetterling, Linear programming and the simplex method, in Numerical Recipes in FORTRAN:

The Art of Scientific Computing, 2nd ed. 423-436,Cambridge University Press, Cambridge, UK, 1992.

[71] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna, Analysis of high-performance floating-point arithmetic on FPGAs, in: Proceedings of the 11th Reconfigurable Architectures Workshop, 2004, pp. 149-158.

[72] A. A. Gaar, W. Luk, P. Y. Cheung, N. SHirazi, and J. Hwang, Automating customisation of floating-point designs, in: Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002, pp. 523-533.

[73] E. Roesler and B. Nelson, Novel optimizations for hardware floating-point units in a modern FPGA architecture, in: Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002, pp. 637-646.

[74] 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 wrapping, in Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999, pp. 240-248.

[75] U. Kulisch, The fifth floating-point operation for top-performance computers, Universitat Karlsruhe, 1997.

[76] D. Goldberg, What every scientist should know about floating-point arithmetic, ACM Computing Surveys, 23 (1991) 5-48.

[77] D. Priest, Differences among IEEE 754 Implementations, <http://www.validgh.com/goldberg/>.

[78] L. Zhuo, G. R. Morris, V. K. Prasanna, Designing scalable FPGA-based reduction circuits using pipelined floating-point cores, in: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), 2005, pp. 147-156.

[79] Z. Luo and M. Martonosi, Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques, IEEE Transactions on Computers, 49(2000) 208-218.

[80] N. J. Higham, The accuracy of floating point summation, SIAM Journal of Scientific Computing, 14 (1993) 783-799.

[81] Xilinx, XUPV2P board user guide, <www.xilinx.com/univ/xupv2p.html>.

[82] U. W. Kulisch, W. L. Miranker, The arithmetic of the digital computer: A new approach, SIAM Review, 28(1986) 1-40.

[83] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, et. al., Design, implementation, and testing of extended and mixed precision BLAS, ACM Transactions on Mathematical Software, 18(2002) 152-205.

[84] J. W. Jang, S. Choi, and V. K. Prasanna, Area and time efficient implementation of matrix multiplication on FPGAs, in: Proceedings of the First

IEEE International Conference on Field Programmable Technology, 2002, pp. 203-202.

[85] K. Q. Li and V. Y. Pan. Parallel matrix multiplication on a linear array with a reconfigurable pipelined bus system, IEEE Transactions on Computers, 50(2001) 519–525.

[86] L. Zhuo and V. K. Prasanna, Scalable and modular algorithms for floating-point matrix multiplication on FPGAs, in Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004, pp.433-448.

VITA

Name:	Chuan He
Address:	Institute for Scientific Computation, Texas A&M University College Station, TX 77843-3404
Email Address:	chuanhe@gmail.com
Education:	B.S., Electrical Engineering, Shandong University, Jinan, China, 1995
	M.S., Electrical Engineering, Beijing University of Aeronautics and Astronautics, Beijing, China, 1998
	Ph.D. Electrical Engineering, Texas A&M University, College Station, TX, 2007