5-2008

High Performance Reconfigurable Computing for Linear Algebra: Design and Performance Analysis

Junqing Sun

University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Junqing Sun entitled "High Performance Reconfigurable Computing for Linear Algebra: Design and Performance Analysis." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Olaf O. Storaasli, Donald W. Bouldin, Jack Dongarra, Xiaorui Wang

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Junqing Sun entitled "High Performance Reconfigurable Computing for Linear Algebra: Design and Performance Analysis." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Gregory D. Peterson
Gregory D. Peterson, Major Professor

We have read this dissertation
and recommend its acceptance:

Olaf O. Storaasli

Donald W. Bouldin

Jack Dongarra

Xiaorui Wang

Accepted for the Council:

Carolyn R. Hodges

Carolyn R. Hodges, Vice Provost and
Dean of the Graduate School

High Performance Reconfigurable Computing for Linear Algebra: Design and Performance Analysis

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Junqing Sun
May 2008

ACKNOWLEDGEMENTS

I wish to thank the many people who helped me achieve my doctorate in Electrical Engineering. I would especially thank my major advisor, Dr. Gregory D. Peterson. He introduced me into this interesting field – computer architecture. Without his continual trust, support and instructive guidance, this work would not have been possible. I would like to thank Dr. Olaf O. Storaasli for his wisdom, insight, and broad knowledge in the field of scientific computing and matrix solvers. I also would like to thank Dr. Don Bouldin, Dr. Jack Dongarra and Dr. Wang for their exceptional support of my research and serving on my committee.

Special thanks go to my colleagues and friends - Yu Bi, Shaoyu Liu, Saumil Merchant, Akila Gothandaraman, Junkyu Lee and Depeng Yang - for sharing the knowledge and happiness in our lab. I also wish to thank Collin B. McCurdy and Richard Barrett of ORNL for useful discussions on computer architecture and matrix algorithms.

This work was partially supported by the University of Tennessee Science Alliance, and the ORNL Laboratory Director's Research and Development program.

ABSTRACT

Field Programmable Gate Arrays (FPGAs) enable powerful performance acceleration for scientific computations because of their intrinsic parallelism, pipeline ability, and flexible architecture. This dissertation explores the computational power of FPGAs for an important scientific application: linear algebra. First of all, optimized linear algebra subroutines are presented based on enhancements to both algorithms and hardware architectures. Compared to microprocessors, these routines achieve significant speedup. Second, computing with mixed-precision data on FPGAs is proposed for higher performance. Experimental analysis shows that mixed-precision algorithms on FPGAs can achieve the high performance of using lower-precision data while keeping higher-precision accuracy for finding solutions of linear equations. Third, an execution time model is built for reconfigurable computers (RC), which plays an important role in performance analysis and optimal resource utilization of FPGAs. The accuracy and efficiency of parallel computing performance models often depend on mean maximum computations. Despite significant prior work, there have been no sufficient mathematical tools for this important calculation. This work presents an Effective Mean Maximum Approximation method, which is more general, accurate, and efficient than previous methods. Together, these research results help address how to make linear algebra applications perform better on high performance reconfigurable computing architectures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

1.1 Motivation

In our time, the traditional Von Neumann computer architecture faces significant challenges that may result in new computing paradigms. CPU-centric computers are forced to invest more power and area on the cache hierarchy to bridge the widening gap between CPU and main memory performance. Meanwhile, heat dissipation and other problems caused by high clock rates make it increasingly difficult to continue the CPU frequency improvement rate. Due to these reasons, current computer architects struggle to fully utilize the exploding chip capacity brought by modern Integrated Circuit (IC) technology.

Matrix operations are widely applied in many applications such as the finite element method, linear system solvers and partial differential equation solvers. However, these applications usually cannot achieve good performance on traditional computers because most of the CPU time is spent on moving big matrices into and out of main memory rather than on computations.

Field Programmable Gate Arrays (FPGAs) show great potential for Reconfigurable Computing. With their rapid increase in gate capacity and frequency, FPGAs can now outperform microprocessors for both integer and floating point operations [6]. Many computationally intensive algorithms achieve significant speedup on FPGAs [7] [8]. We investigate the feasibility of utilizing FPGAs for linear algebra because of its potential importance in scientific computing.

1.2 Statement of Approach

1.2.1 Reconfigurable BLAS (RBLAS) Library

The BLAS (Basic Linear Algebra Subroutines) provide standard building blocks for performing basic vector and matrix operations. Because of their efficiency, portability, and

wide availability, BLAS are commonly used in the development of high quality linear algebra software such as LAPACK [1]. To explore the potential performance of linear algebra on reconfigurable computers, we implement BLAS kernels onto FPGAs which target the Cray XD-1 supercomputer at the Oak Ridge National Laboratory (ORNL).

Practical application of reconfigurable computing depends on efficient system integration to effectively utilize these high-speed accelerators to improve overall performance. Although many results from small FPGA-based systems are promising, overall performance is often limited by the I/O bandwidth [9]. The best way to integrate FPGA accelerators into a balanced computing system remains an open problem [10]. Our FPGA designs utilize deeply pipelined structures to maximize throughput. Due to frequent data movement in matrix operations, a data streaming architecture is used and control signals are simplified to reduce the overhead during system integration.

Lower-precision data requires less hardware resources and usually has higher performance (speed) on modern computer architectures. On the other hand, certain data precision is usually required by specific applications to obtain numeric convergence or result accuracy. Mixed-precision algorithms utilize lower-precision data formats for most computations, and higher-precision data format only when necessary. Mixed-precision algorithms can be applied to linear algebra to simultaneously achieve both higher performance and required accuracy [14]. Our work explores this approach using FPGAs, which offer more flexible data formats compared to traditional computers. We analyze floating point performance on FPGAs for different precision data formats, design mixed-precision architectures, and give performance analysis.

1.2.2 Performance Evaluation

Performance analysis is important in understanding computer efficiency and potentially to determine the best mapping of applications to reconfigurable resources. There are three broad classes of performance evaluation techniques: measurement, simulation, and analytical modeling [74]. Measurement is probably the most accurate approach, but the system to measure must be implemented and available. This technique also cannot predict

system performance or analyze different system configurations. Simulation provides visibility and controllability to the architecture simulated [75]. However, the very low performance is an unavoidable drawback. A big system can rarely be exhaustively simulated because of the exploding behavior states. Another problem for simulation is that general conclusions cannot be drawn from a single simulation because the performance is usually sensitive to collections of parameters. The large number of simulations that may be required for statistically significant results may take a very long time. Analytical modeling involves building a mathematic model for the system at the desired the level of details. The main advantage of analytical modeling is that it can allow exploration of the performance of a system before its construction. At the same time, a closed form analytic model greatly helps to adopt mathematic tools for performance analysis, such as sensitivity analysis, optimization, and load schedule.

Reconfigurable computing based on FPGAs has already shown great potential in accelerating scientific computations. However, such factors as long communication time can degrade overall system performance. Performance modeling provides a very important tool to predict execution time, decide optimal load mapping, and schedule in reconfigurable computing. We are interested in building a performance model for reconfigurable computing in a parallel environment which is common for large scientific applications.

1.3 Contributions

This dissertation proposes to use reconfigurable computing for high performance linear algebra computations. To achieve this goal, we develop high performance circuits and algorithms on FPGAs and analyze our designs by building accurate execution time models. This dissertation contributes both hardware/software implementations and theoretical derivations including:

- Development of an innovative FPGA architecture for sparse matrix vector multiplication with significant speedup over traditional CPUs.

- Development of high performance and cost efficient circuits for high performance linear algebra on FPGAs.

- Implementation of an innovative LU decomposition architecture on FPGAs with significant speedup over CPUs.

- The first to utilize mixed data format in sparse matrix vector multiplication on FPGAs and successfully achieve higher performance than single data format design.

- The first to propose hardware architectures for pivoting algorithm on FPGAs using HDL code.

- The first to utilize both a CPU and a FPGA for high performance linear direct solver by developing an innovative hybrid direct solver.

- The first to implement LU decomposition with pivoting on a Cray-XD1 supercomputer and give performance analysis.

- The first to explore the performance of FPGA based floating point linear direct solvers in different data formats and therefore point out the importance of using lower-precision data formats to obtain high performance.

- The first to develop mixed-precision direct solvers on FPGAs which achieves the high performance of lower-precision data formatting without any loss of accuracy.

- Development of an accurate performance models for LU decomposition on FPGAs. Extend performance models in previous work to new reconfigurable computing systems.

- Development and proof of mathematical theorems on properties of maximum random variables. Successfully utilize these theorems to improve the accuracy and efficiency of performance modeling. Point out potential applications of these mathematical results.

1.4 Outline of Documents

This chapter introduces motivations, approaches, and contributions of this dissertation. More detailed background of our work is described in Chapter 2. We introduce our high performance linear algebra design on FPGAs in Chapter 3. The performance of these designs is improved by using mixed-precision algorithms and architectures in Chapter 4. We implement our design on Cray-XD1 supercomputer and give performance analysis in Chapter 5. Chapter 6 develops performance models for reconfigurable computers and analyzes our design and architectures. In chapter 7 we derive mathematical tools for maximum random variables and use them for improve our performance models. We conclude this dissertation and point out future work in chapter 8.

2 Background and Related Work

This chapter introduces previous work related to this dissertation. First of all, we introduce floating point operations on FPGAs from which we can determine the performance of floating point linear algebra. Second, we are interested in the development of parameterized linear algebra subroutines on FPGAs. Some related FPGA designs for linear algebra are discussed. Third, we introduce a mixed data format algorithm and implementation on CPUs which will be extended to our FPGA designs for high performance in following chapters. To analyze and optimize our design, we build performance models for reconfigurable computing. Therefore, we introduce some related backgrounds and point out an important problem affecting the accuracy of parallel computing models. Finally, we describe the FPGA development environment and give conclusions.

2.1 Linear Algebra on FPGAs

2.1.1 Related Work

Floating Point IP Cores

Floating point data format is widely used in scientific computing. Previous work shows that the peak floating-point performance of FPGAs has surpassed that of CPUs and will soon have an order of magnitude advantage [6]. To exploit the floating point advantage of FPGAs, many researchers and commercial vendors provide floating point IP cores on FPGAs. Xilinx has included pipelined floating point operators in its ISE tools [18]. The data format and pipeline depth can be parameterized when configuring the operators. Some operators, such as multipliers, can be built both from combinational logic slices and more efficient embedded circuits, such as the built-in 18x18 multipliers for Virtex II and DSP48 for Virtex 4 FPGAs. Other floating point IP cores can also be found in academic research groups [32] and [33].

Sparse Matrix Vector Multiplication (SpMxV)

Sparse matrix-vector multiplication (SpMxV), $y = Ax$, is one of the most important computation kernels in scientific computing, such as iterative linear equation solvers, least square and eigenvalue solvers [2]. In this computation kernel, matrix *A* is a large sparse matrix and *x* is a dense vector. To save storage and computational resources, usually only the nonzero elements of matrix *A* are stored and computed. Pointers are necessary to store the sparsity structure but also degrade memory operation efficiency. This is because the vector 'x' is addressed by pointers during computation and possibly loses spatial locality in the cache-memory hierarchy. Furthermore, utilizing pointers requires additional load operations and memory traffic. Despite numerous efforts to improve SpMxV performance on microprocessors [3], [4], [5], these algorithms rely heavily on the matrix sparsity structures and the computer architectures, typically resulting in degraded performance on irregular matrices.

Several FPGA designs for SpMxV have been reported before. Zhuo and Prasanna designed an adder tree based SpMxV implementation for double precision floating point that accepts any size of matrices in general compressed row storage (CRS) format. ElGindy and Shue proposed SpMxV on FPGAs for fixed point data [19]. DeLorimier and DeHon arranged the processing elements (PEs) in a bidirectional ring to compute the equation $y = A^i x$, where A is a square matrix while i is an integer. The design they proposed reduces the I/O bandwidth requirement greatly by sharing the results between PEs. Because local memories are used to store the matrix and intermediate results, the matrix size is limited by the on-chip memory [12]. El-kurdi et al proposed a streaming architecture for finite element method matrices [13].

Matrix Factorization

Matrix factorization is widely used to solve linear equations, while LU decomposition is the most commonly used method for matrix factorization. For some common data processing algorithms, wireless sensor networks require efficient LU decomposition running on resource-constraint senor nets [35].

Significant previous work addresses this important computational kernel on hardware. Daga described a block LU decomposition algorithm and corresponding architecture [36]. Govindu developed circular linear array architecture on FPGAs and achieved a 10% - 60% reduction in energy over that of a traditional CPU [36]. Those two designs assume the matrix is non-singular and no pivoting is needed. For a matrix of size $n \times n$ this design requires n PEs, with each PE consisting of a multiplier and an adder. One of the PEs is specialized for the division computation and has just a divider. To avoid the data dependencies and fill the deep pipelines of floating point units, multiple matrices are required to be interleaved in the FPGAs and operated alternatively. Zhuo et al improved this design by increasing parallelism through more PEs and achieved higher GFLOPS performance than a 2.2 GHz AMD Opteron processor by using a Virtex-II Pro FPGA [38]. Wang developed parallel LU factorization for power systems [39]. Kim built a systolic array architecture for LU decomposition which needs $n^2/2$ PEs for a $n \times n$ matrix. Each PE has two multiplier-subtractor units [40]. All these previous designs assume that target matrices are positive definite and no pivoting is required. Although pivoting will complicate control logic, it increases the numeric stability of LU decomposition. Therefore, we will consider hardware architecture for pivoting. Turkington et al proposed to use high level language for LU decomposition algorithm with pivoting [87]. Handel C used in [87] directly maps high level codes to FPGA hardware without considering specific hardware architecture. This approach brings great convenience but also loses significant performance compared to HDL based hardware design.

Because of the importance of linear algebra in scientific computing and embedded systems, it is important to develop hardware accelerators for linear algebra subroutines. Previous work has shown the potential of using FPGAs. However, many problems still left unsolved, such as high performance architectures and algorithms, matrix storage optimization for FPGAs, system interfaces, high performance algorithms, and performance evaluation.

2.2 Mixed-Precision Algorithms

2.2.1 Introduction

Iterative refinement for the solution of linear equations has been extensively studied to improve the accuracy of linear systems' solutions [42]. As shown in Figure 2-1, once the equation at step 1 is solved, the solution can be refined through an iterative procedure. In each of the iterations, the residual is computed based on the solution at the previous iteration (step 4); a correction is computed as in step 5 by using the computed residual; and finally this correction is applied in step 6 for the updated solution.

The common use of iterative refinement consists of performing all arithmetic operations with the same precision (either single or double precision floating point on traditional CPUs). Langou et al investigated the application of mixed-precision, iterative refinement where the most computationally expensive steps, 1 and 5, are performed in single precision floating point and steps 4 and 6 are performed in double precision floating point [14]. Strzodka and Göddeke explored similar algorithms for iterative solvers [43], [44], and [45]. The error analysis for mixed-precision iterative refinement shows that this approach can achieve the same accuracy as full double precision arithmetic provided that the matrix is not too badly conditioned [14].

$1: x_0 \leftarrow A^{-1}b$
$2: k = 1$
$3:$ **repeat**
$4: \quad r_k \leftarrow b - Ax_{k-1}$
$5: \quad z_k \leftarrow A^{-1}r_k$
$6: \quad x_k \leftarrow x_{k-1} + z_k$
$7: \quad k \leftarrow k + 1$
$8:$ **until** *converge*

Figure 2-1: Iterative Refinement Technique for Linear Equations

2.2.2 Performance on Traditional CPUs

Previous work reveals that on many current processors, the performance of 32 bit floating point arithmetic may be significantly higher than 64 bit floating point arithmetic due to many factors [41]. First of all, many processors increase their throughput by using vector instructions. For example, the Intel IA-32/IA-64 and AMD Opteron families have the SSE2 instruction set; the Motorola, Freescale, and IBM PowerPC has the AltiVec unit. For SSE2, a vector unit can complete four single precision operations every clock cycle but only two for double precision [14]. Secondly, data movement is cut in half for single precision data compared to double. This helps performance by reducing memory traffic across the bus and enabling larger blocks of the user's data to fit into cache.

In mixed-precision iterative refinement algorithms, the computationally expensive steps (1 and 5) are performed very fast in single precision arithmetic while the steps requiring double precision accuracy (4 and 6) are typically less computationally demanding. Langou et al explored the single/double mixed-precision iterative refinement algorithm and achieved promising results on multiple architectures. There are limitations to the success of this process, such as when the conditioning of the problem exceeds the reciprocal of the accuracy of the single precision computations. In that case, the double precision algorithm should be used.

Single/double mixed-precision iterative algorithms take advantage of the higher performance of single precision arithmetic on hardware while achieving the accuracy of double precision. FPGAs have flexible data formats. Shorter formats can result in higher frequency, lower memory/bus requirements, and reduced energy consumption. Exploring mixed-precision iterative algorithms is desirable for both high performance computing and resource-constrained embedded systems.

2.3 Hybrid System Performance Modeling

2.3.1 Performance Modeling

Execution time modeling plays an important role in understanding system performance. Given certain computation loads and resources, the execution time of each processor can be modeled as a random variable while the overall system time is determined by the last processor completing its task [52]. For example, Peterson and Chamberlain built a model for networked workstations [52], [64]. The overall execution time consists of three parts: parallel work, serial work, and overhead. Smith extended this work to consider the impact of reconfigurable computing devices in shared, high performance reconfigurable systems [75].

2.3.2 Mean Maximum Estimation

In modeling parallel applications, the execution time for each processor can be represented by variables X_i [52]. Due to the effects of synchronization, estimation of the system execution time depends on calculating the expectation of maximum value (EMV) $E(\max_{i=1}^{N} X_i)$ [52]. Unfortunately, the solution in closed form usually cannot be derived for this term. This problem becomes much more challenging for heterogeneous environments, where the execution time for different processors has different distributions. Due to its importance in parallel computation modeling, many researchers have tried various kinds of methods for this problem.

Monte Carlo (MC) methods can be used to compute EMV with any initial distributions. However, it has no analytical expression and the computation load is unaffordable for most of the evaluations. The use of order statistics is first suggested for analyzing parallel program performance by Weide [65]. For independent identically distributed (i.i.d.) random variables with known distribution functions, extreme theory [66] [67] can approximate the distribution of extremes. The approximation becomes exact as the number of random variables increases. The drawback is that to derive the mean maximum by using extreme theory is usually difficult. Further, extreme theory cannot derive asymptotical extreme distribution functions for many distributions.

Agrawal [57] evaluates the performance of synchronous logic circuits simulation by applying a Binomial distribution to determine the number of events at each processor. The number of active gates at each time stop which needs to be simulated for each processor is random distributed. Order statistics are used to calculate the expectation of the execution time when the number of gates is equally distributed to each processor. In this case, the processor loads are independent and identically distributed binomial random variables. For the imbalanced case of uneven work distribution, processors are divided into different subsets. The processors in the same subset are identically loaded, so the order statistics can be applied to calculate the expectation of execution time for each subset. Because there is no analytical method for the non-identical random variables, the maximum subset execution time is considered as the overall execution time [57].

Despite all the previous efforts above, this problem of calculating EMV is still unsolved after decades. First of all, current methods cannot accurately compute the expectation of the maximum variables (EMV) for heterogeneous initial distributions. Secondly, even when the initial random variables are i.i.d., current methods cannot cover all the commonly applied distributions in parallel computing or are not accurate enough. MC simulation can be general and accurate enough for all distributions. However, its expensive computational requirements are usually unacceptable in large scale parallel computation performance evaluation. Furthermore, MC simulation cannot provide an analytical form for this problem, which is important for sensitivity analysis and optimization of the execution time.

This dissertation presents an innovative approximation method for this problem, where EMV is calculated by very convenient functions. Compared to previous work, this approach is more general, accurate, and computationally efficient.

2.4 Development Environment

2.4.1 Software Environment

Optimal FPGA development requires knowledge of electric circuits. After an algorithm is analyzed and converted into logic blocks by the developers, hardware description

languages, such as VHDL and Verilog, can be used to describe the logic. With the help of tools, the VHDL or Verilog scripts can be compiled, synthesized, and then mapped to hardware logic units.

VHDL and Verilog provide detailed control over the circuit design, but also require hardware expertise, which is usually not familiar to software engineers. Due to the potential of FPGAs in scientific computations, several high level languages were developed for the convenience of high-level users. For example, SRC's IMPLICIT+EXPLICIT™ ARCHITECTURE has both implicit (CPU) and explicit (FPGA) computation engines [31]. It allows programmers to use both C and FORTRAN. The compiler generates a unified executable to run on the CPU and FPGA. The compiler extracts parallelism and generates pipelined logic initiated in the FPGA chip. It also generates all the required interface code to manage the movement of data to and from the FPGA, and to coordinate the CPU with the logic running in the FPGA [31]. VIVA is a graphical language developed by Starbridge Systems [30]. Programmers can easily describe an algorithm by placing and connecting computation unit icons in a graphic environment. VIVA provides an extensively optimized library for different scientific computations [30]. Other commercial vendors such as Mitrion [78] and Xilinx [18] also provide tools to convert high level languages to hardware.

2.4.2 Reconfigurable Computers

Many supercomputer vendors have noticed the potential power of FPGAs and developed machines by utilizing FPGAs. The Cray-XD1 has up to 6 FPGA chips on each chassis as application accelerators [16]. The SGI RASC technology based on FPGAs enabled dramatic application acceleration compared to traditional servers [29]. The Hypercomputer from Starbridge Systems uses FPGAs as the computation engine and achieves competitive performance with traditional supercomputers [30]. Other companies that provide reconfigurable computer architectures include SRC [31], DRC [80], XtremeData [79], and Nallatech [28].

2.5 Conclusions

This chapter introduces previous work and problems of high performance reconfigurable computing for linear algebra. The consistently improving FPGA capacity and development environment make FPGAs very attractive for computational intensive computations. Although many efforts have been delivered, how to utilize FPGAs for high performance linear algebra is still an unsolved problem. In the next chapter, we will introduce our FPGA design for some linear algebra subroutines. Further performance improvement and analysis will also be introduced in following chapters.

3 Reconfigurable Processor Design for Linear Algebra

Because of the parallelism inherent in most many matrix algorithms, reconfigurable accelerators can achieve higher peak performance than microprocessors. However, due to the frequent memory movement in matrix operations, especially sparse matrix operations, the system performance is heavily affected by memory bandwidth and overheads in real applications. Therefore, effectively integrating FPGA accelerators to computation systems is important for the overall system performance. In this chapter, we introduce our reconfigurable matrix computation design. System performance is optimized with both matrix algorithms and hardware architectures.

3.1 SpMxV

We introduce an innovative SpMxV solver for FPGAs (SSF). Because the hardware does not need to change for different matrices, the initialization time is minimized and the system integration complexity is reduced. The storage format plays an important role in SpMxV and affects the performance of optimization algorithms. We use the common format, Compressed Row Storage (CRS), for our FPGA design [15]. Our design requires the multiplicand vector x to be stored in the FPGA local memory. Large matrices and vectors are divided into sub blocks. In contrast to traditional Block CRS (BCRS) format, our matrix storage format is optimized for FPGA accelerators. As explained later, this format is compatible with algorithms using BCRS but reduces requirements for both I/O bandwidth and computational resources.

Because floating point adders are usually deeply pipelined to achieve high frequency, accumulating floating point data is normally difficult in digital design. We propose an accumulation circuit for SpMxV. By taking the advantage of the data flow, we design an innovative summation circuit which has low resource requirements and simple control logic.

3.1.1 SpMxV on FPGAs

In general, the SpMxV computation $y = Ax$ is defined as:

$$y_i = \sum_{j=0}^{N} a_{i,j} x_j \text{ , } (0 \le i \le M) \tag{3-1}$$

where A is an $M \times N$ matrix, while y and x are $M \times 1$ and $1 \times N$ vectors, respectively. For efficiency, most sparse matrix algorithms and storage formats only operate on nonzero elements. For each nonzero element, there are two floating point operations (one add and one multiply). By convention, we assume A has n_{nz} nonzero elements. All the elements of A and x in storage have to be moved into the FPGA, while computed results for y have to be moved out of the FPGAs. Because of the pointers used in storage formats, the indices for matrix A also need to be moved into FPGA local memories. Suppose there are n_p pointers needed. If we assume data sizes for A, x, y, and pointers are the same, the total I/O requirement is at least:

$$n_{IO} = n_{nz} + m + n + n_p \tag{3-2}$$

Because of the loss of locality and limited memory size, matrix and vector data may have to be moved multiple times on traditional microprocessor-memory architectures. In our SSF design, I/O time is hidden by overlapping with computations to reduce the overall time. The time used to preload the data onto the FPGA is denoted as T_{Init}, which also includes hardware initialization and data formatting. We denote T_{sync} as the time for the FPGA to synchronize with the host, and $T_{overhead}$ for other overheads. The overall time spent on FPGA accelerators is thus

$$T = \max(T_{comp}, T_{IO}) + T_{init} + T_{sync} + T_{overhead} \tag{3-3}$$

In equation (3-3), the computation time T_{comp} is the only part doing matrix multiplication operations. However, SSF cores also have tremendous I/O demands. To improve the

overall performance, we need to overlap the I/O operations with computations as much as possible. At the same time, synchronization and overhead needs to be minimized.

3.1.2 Sparse Matrix Storage Format

The CRS format makes no assumptions about the sparsity structure of the matrix and has no unnecessary elements stored [15]. In the CRS format, 3 vectors are needed: the "*val*" vector stores subsequent nonzeros of the matrix in row order; the integer vector "*col*" stores the column indices of the elements in the "*val*" vector; while the integer vector "*len*" stores the number of nonzero elements of each row in the original matrix. As an example, consider the matrix A defined by

$$A = \begin{pmatrix} 2 & 0 & -3 & 0 & 0 \\ 0 & -1 & 0 & 0 & 6 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 5 & 8 & 0 & 6 & 0 \end{pmatrix}$$

The CRS format for this matrix is then specified by the arrays given below:

Val:	2, -3, -1, 6, 1, 9, 5, 8, 6
Col:	0, 2, 1, 4, 0, 1, 0, 1, 3
Len:	2, 2, 1, 1, 3

In our design, the multiplicand vector needs to be loaded into FPGAs. The maximum matrix size that can be fit into FPGA chips is restricted by the on-chip memory size. Big matrices need to be divided into sub-matrices. Our matrix division format is shown in Figure 3-1. The matrix is divided into stripes along the rows. Each stripe is then divided to sub-matrices (shown in dashed lines). The sub-matrices having only zeros are neither stored nor computed. We refer to this format as Row Blocked CRS (RBCRS).

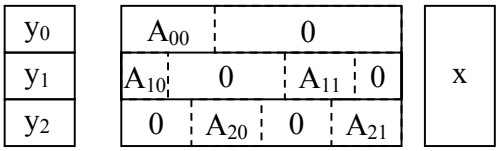

Figure 3-1: Row Blocked Compressed Row Storage (RBCRS)

During the computation, sub-matrices in the same stripe are assigned to the same FPGA accelerator. Note that the elements required from vector x will differ for each stripe based on the sparsity structure. The vector x is kept in the FPGA off-chip memory, and part of it (x_j) is loaded before computing $A_{ij}x_j$. Note that the result $A_{ij}x_j$ is not sent out after being computed, but is stored in the FPGA and added with the result from the next sub-matrix vector multiplication in the same row. For example, the result of $A_{20} \times x_0$ will be stored in the FPGA to add with the result of $A_{21} \times x_1$. After all the matrices in a row are computed, the result y_2 is read out. This approach saves I/O bandwidth and computational resources.

3.1.3 Framework and Basic Design

This section introduces our basic design for the SSF and the framework when used in software applications. In the basic design, we discuss the design for integers. Because the integer adders have latency of one clock cycle, the accumulation circuit can be built with a simpler pipelined structure. The summation circuit can also be simply implemented by using an adder. The basic design can also be made to support floating point data, but at a lower performance. The design for deeply pipelined floating point operators is more complicated because of the read after write hazard discussed in the next section.

For all the designs, we assume the matrices and vectors are too large to be accommodated in the FPGA on-chip memory. The algorithm may be executed by multiple FPGAs working in parallel. Without loss of generality, we assume each PE computes the sub-matrices in one stripe. To illustrate the matrix mapping in practical implementations, we briefly introduce the structure on the Cray XD-1.

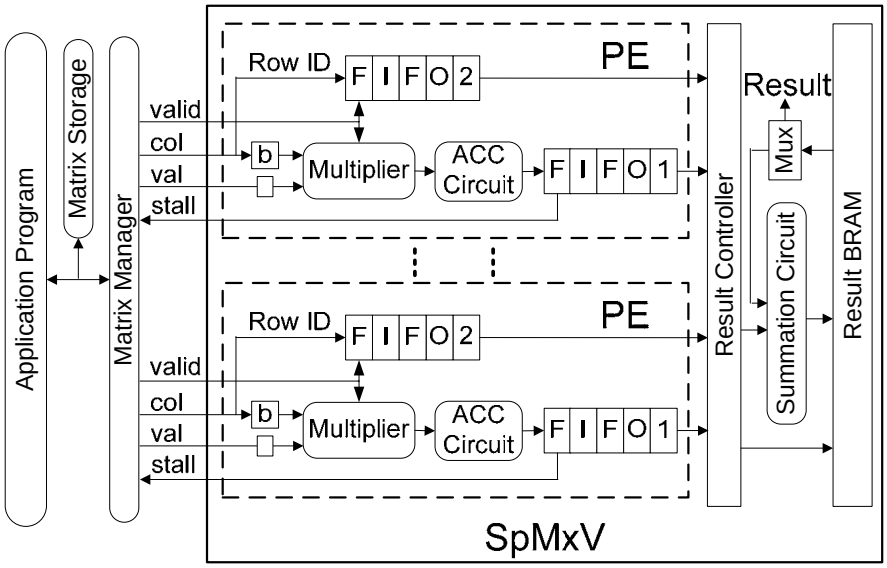

Figure 3-2: Data Path and Framework of SSF

Basic Design and Interfaces

Figure 3-2 shows the basic design of our SSF core and the framework for applications. The application program stores the matrix in Blocked CRS format. The matrix manager feeds sub-matrices to the SSF core in CRS format and reads back the values of y_i. The application program may read back all the $y_i s$ from different FPGAs to determine the result y.

In our design, each PE is a pipeline consisting of a multiplier, adder, and result adder. FIFO1 is used as a buffer for intermediate results. The data for "*val*" and "*col*" are imported into the PE synchronously. The multiplicand vector x_j is preloaded into the FPGA and addressed by "*col*". Because there is a one clock cycle latency to read data from Block RAM (BRAM), a buffer is inserted for "*val*" before the multipliers. An illustration of the signals is shown in Figure 3-3. At the end of "*col*", there is one data for Row ID in the sub-matrix. It is used to address the result BRAM and stored into FIFO 2. Zeros are inserted when there is a stall signal or when waiting for the I/O to feed the next rows. The

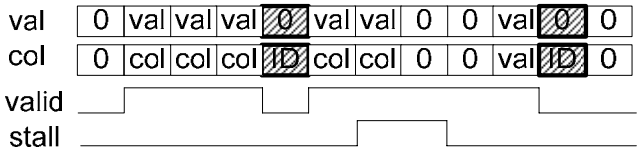

Figure 3-3: Signals for Processing Elements (PEs)

signal "*valid*" is set when "*val*" and "*col*" data are being imported. It is also used to control the components: multiplier, adder, FIFO1, and FIFO2.

Suppose row i of a sub-matrix is being imported. The PE computes n_{nz} values, one for each nonzero for the row, and stores the resulting data into FIFO1. The summation circuit adds the results from the PEs with the data in the result BRAM addressed by data read from FIFO2. Note that the data in the result BRAM are from previous sub-matrices.

To maximally utilize the data input bandwidth, all the components in the PEs are synchronized with the pipelined data by using the signal "*valid*". Some intermediate signals are produced to tell when the components have valid inputs and outputs. Most of these signals can be produced by adding appropriate delays to the signal "*valid*". For example, the "*input valid*" signal for a multiplier is produced by adding one clock cycle delay to the signal "*valid*". The "*write enable*" signal for FIFO1 is set for one clock cycle when the result for one row is accumulated. The "*stall*" signal is set if FIFO1 is close to full. When a stall is issued, zeroes are inserted as inputs while the "*valid*" signal does not change. If a row is being imported, the multipliers and accumulators operate on inserted zeros and will have no affect on the results. Note that the data already in the pipelines of multipliers and ACC circuits still need to be computed and stored into FIFO1. Therefore, FIFO1 needs to have certain free space when a stall signal is issued. The size of the free space should be bigger than half of the total pipeline stages of an adder and a multiplier.

The Result Controller checks the "*empty*" signal of all the FIFOs. If the FIFO is not empty, the result will be read out and added to the corresponding value in Result BRAM. The row

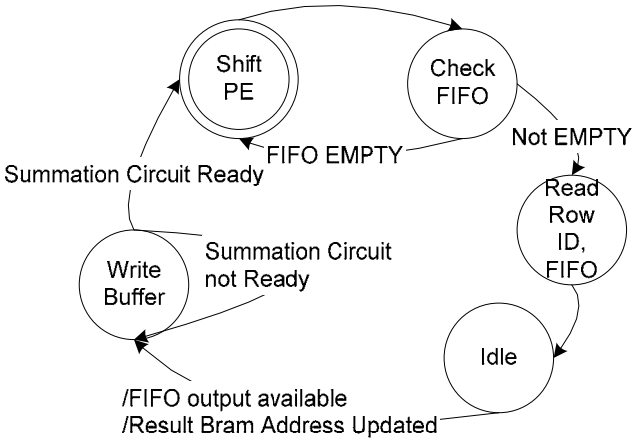

Figure 3-4: State Diagram of Result Controller

ID is read out at the same time as the data address. The purpose of using the adder is to sum the results from all sub-matrices in the same stripe as explained in section 2.1. The result BRAM will be read and cleared when all the sub-matrices in the same line (for example A_{2i}) are computed.

In the basic SSF design, it takes 7 clock cycles for the Result Controller to read from one PE. The state diagram of the Result Controller is shown in Figure 3-4. The result of a row is stored in FIFOs and read out by the Result Controller. A stall signal is issued when a FIFO becomes full. To avoid the shared summation circuit becoming the bottle neck, its operation time should be able to be overlapped by I/O or computation time. As discussed later, multiple summation circuits can be used in parallel to increase the throughput.

The structure of our design on the Cray XD-1 is shown in Figure 3-5. The Matrix Manager feeds the data to the SSF core. Sub-matrices and vectors are loaded to the QDR memories on different blades. During the execution of applications, the host processor sends the matrices/vectors addresses and the "*start execution*" request to the FPGA through the RapidArray Transport (RT) interface IP core [16]. The host continues its execution after it receives the acknowledge signal from the FPGAs. Each FPGA then starts to operate on

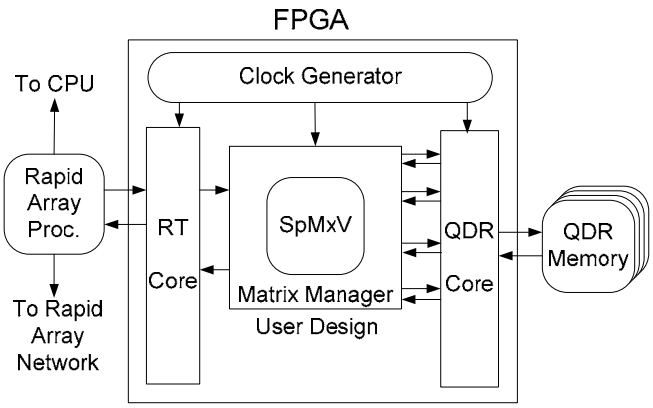

Figure 3-5: Structure on Cray XD1 FPGA [16]

its sub-matrices. The results are then written to QDR memories and a completion signal is sent back to the processor node. When completion signals are received from all the FPGAs, the host node retrieves the final matrix results.

3.1.4 Complete Design

Pipelined ACC Circuit

Pipelined floating point operators can be used to improve the frequency of our design. However, the accumulator cannot be simply built as in the basic design because of read after write data hazards. The dataflow for a 5-stage pipelined floating point adder is shown in Figure 3-6. The hashed blocks are inserted zeros, which come when the valid signal is zero (invalid). The second row shows the outputs of this circuit. There are three problems in this circuit:

1. The output is not accumulated into one data as in the integer design. For example, the first row has 6 numbers with a summation of 21. The circuit gives 5 outputs we should use (2, 3, 4, 5, and 7).
2. The data is added to the output of previous rows. For example, 8, 9 and 10 are added to 3, 4 and 7.
3. To solve the first problem, we can use 5 registers to store the last 5 outputs of each row. However, these registers will have results from previous rows when the current row is

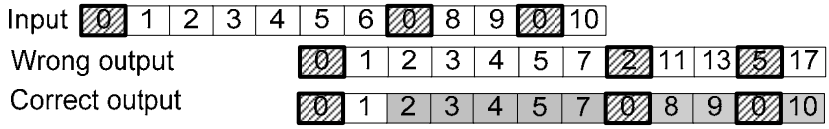

Figure 3-6: Data Flow for Pipelined ACC Circuit

short. For example, if 5 registers are used to store the outputs from the second row, the data should be captured when 13 (the correct output is 9) comes out. However, 5, 7 and 2 are also stored because the data stream to be accumulated is too short.

To solve the data hazards mentioned above, we design an ACC circuit with one pipelined floating point adder. One of the inputs, (a), is connected to the output of multiplier and works as the input for the ACC circuit. The last 5 outputs of the adder are stored in 5 registers to work as the output of the ACC circuit. The correct outputs from our design are also in Figure 3-6, where the blocks in grey are data stored in registers. For example, 8 and 9 are stored in two registers as the output of 8 and 9 in the first line. The other 3 registers have just zeros.

Adder Tree

For pipelined adders with L clock cycle latency, L outputs will be stored into FIFO1 to add with the data in the result BRAM. One way to solve this problem is to add these L outputs by an adder tree. Suppose L is equal to 4, we need to add 4 data from the FIFO and 1 data from the result BRAM. For these 5 inputs, an adder tree with 3 levels and 4 adders are needed as shown in Figure 3-7. If the number of inputs is not a power of two, shifters with latency L can be used in an adder tree to take the place of adders to save resources.

For our design with double precision data, 12 outputs from the FIFO and 1 data from the result BRAM need to be added. We use 12 floating point adders to build the adder tree, which costs 25% of the total slices of a Xilinx XC2VP70 FPGA. The data flow of the adder tree used in our design is shown in Figure 3-8.

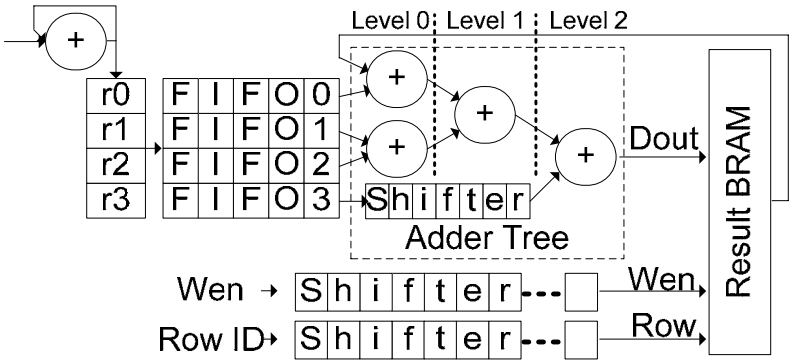

Figure 3-7: Adder Tree Used for Pipelined Adders

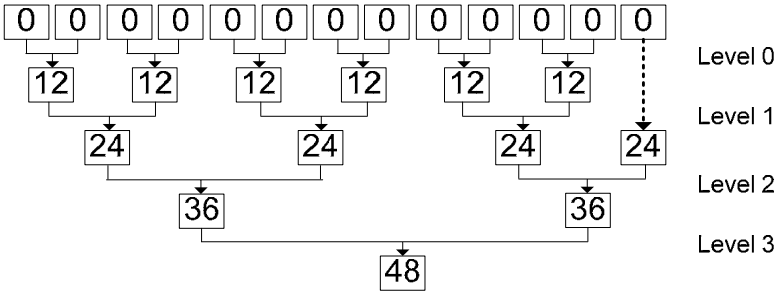

Figure 3-8: Data Flow for Adder Tree

In Figure 3-8, the rectangles represent data. The numbers in rectangles are the clock cycles when that data is available. The dashed line is a FIFO with a latency of 24 clock cycles. The final result comes out 48 clock cycles after the inputs are available, so it is very important to capture the output at the right clock cycle. We input the row ID and write enable signal to two shifters with depth of 48 at clock 0. They will come out with the result at clock 48 to be used as the address and write enable signal for the result BRAM.

Reduced Summation Circuit

Because of the large adder tree, we propose a reduced summation circuit as shown in Figure 3-9. The idea is to reduce the number of adders by importing just two data each clock cycle. The data coming out first is stored in a buffer and computed with the next. By inserting a certain number of buffers between the adders and taking advantage of the data flow, we designed a summation circuit for this function without control logic. For our double precision design, 4 adders and 7 buffers are used in total. 16 registers are used to store the data from the FIFO, the result Block RAM, and 3 zeros to fill the pipeline for correctness. This will be explained later in the data flow.

The data flow here is more complicated than in the adder tree, as shown in Figure 3-10. The data in a row are added by the same adder in serial, while buffers are used to delay the intermediate data for the appropriate time. For example, the datum on clock 12 should be added to that on clock 13, so a buffer needs to be added before adder 1. We can see that the data on clock cycle 18 does not have a counterpart for the addition operation. We pad with zeros to obtain the correct sum. The shaded rectangles are inserted zeros. Figure 3-11 shows that the final result can be captured 55 clock cycles after the data is available in the buffer. In our design, a "*Write Enable*" signal for the result BRAM is stored to a shifter with length of 55 at clock 0. When the "*Write Enable*" comes out of the shifter, the final result will also be ready.

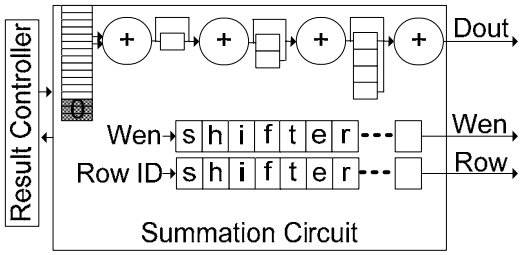

Figure 3-9: Reduced Summation Circuit

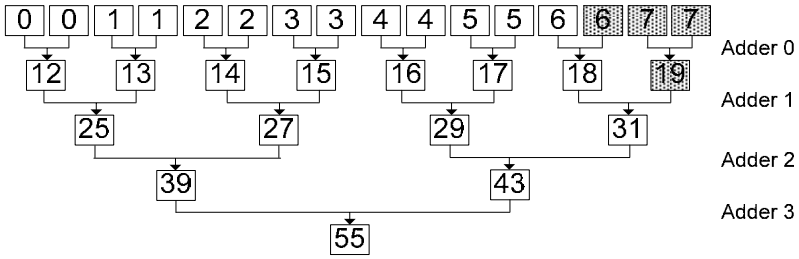

Figure 3-10: Data Flow for Summation Circuit

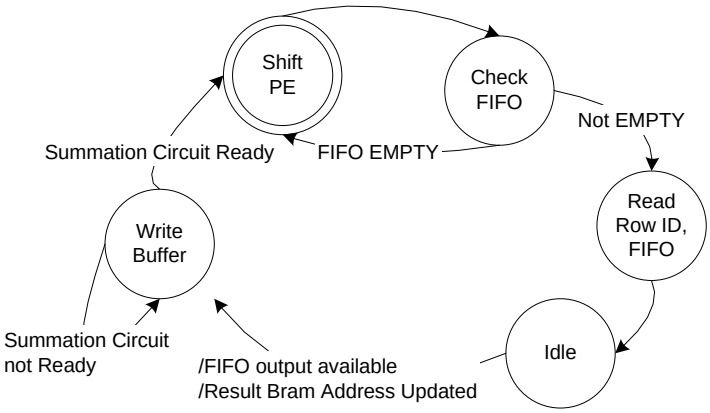

Figure 3-11: Result Controller for Summation Circuit

The Result Controllers of these two circuits are very similar. Because of their long latency, the Result Controller does not wait for the result and write to the result BRAM. Instead, we insert the row ID and write enable signals to be written into shifters at clock cycle 0. If the three outputs of these two circuits are connected to corresponding pins of the result BRAM, the data should be written automatically.

The Result Controllers for the summation circuit and adder tree have just 5 states. However, the summation circuit needs 8 clock cycles to compute 16 data for each PE, so the time on each PE is 8 clock cycles. On the other hand, the adder tree can compute for each PE per clock cycle, but is slowed down by the Result Controller. The Result Controller for the adder tree can be further improved by adding control logic behind the FIFOs in each PE. Table 3-1 compares the summation circuit and adder tree.

3.1.5 Implementation Results and Comparison

We implemented our SSF design by using Xilinx ISE and EDK 8.1 [18]. ModelSim and Chipscope [18] are used for verification and debugging. For mathematic operations, we use Xilinx IP cores which follow the IEEE 754 standard and that can also be customized [18]. Considering the limited size of the FPGAs, we use a summation circuit for the floating point design. The BRAM size for x_i, y_i are 1024. The adders and multipliers are

Table 3-1: Comparison of Adder Tree and Summation Circuit

Design	Number of Adders	Latency (clock cycles)
Adder Tree	12	48
Summation	4	55

Table 3-2: Characteristics of SSF on XC2VP70-7

Design	64 bit Integer	Single FP	Double FP
Achievable	175MHz	200MHz	165MHz
Slices	8282 (25%)	10528 (31%)	24129 (72%)
BRAMs	36 (10%)	50 (15%)	92 (28%)
MULT18X18	128 (39%)	32 (9%)	128 (39%)

provided by Xilinx [18]. To compare our results with previously reported designs [11], we target the Xilinx XC2VP70-7, which is similar to the devices our platforms have. The characteristics are summarized in Table 3-2.

The slice usage and the frequency of our design are dominated by the mathematic operators, while the effect from control logic is almost negligible. If high speed floating point operators are used, the speed of our design can be improved accordingly. Our design can easily adapt to different data formats by simply replacing IP cores. The only change for the control logic is the latency of operators and the interface width, which are defined as a variable in the VHDL. Our design is deeply pipelined. Ignoring I/O bandwidth limitations and communication overheads, two floating point or integer operations (one addition and one multiplication) can be done per clock cycle by each PE.

Previously reported work describes an implementation that achieves 2340 MIPS at 28.57 MHz frequency by using 3 multipliers [19]. However, that design is for fixed point data. The closest related work is [11], which develops an adder-tree-based design for double precision floating point numbers. A reduction circuit is used in their design to sum up the

Table 3-3: Double Floating Point Design Comparison with [11]

Design	Design in [11]	SSF
Frequency	160Mhz	160Mhz
Adders	7+7 (Reduction Circuit)	12
Multipliers	8	8

floating points. Because the frequency is mostly dependent on the floating point operations for both designs, the achievable speed is similar in these two designs if the same mathematical IP cores are used. When 8 multipliers are utilized, both designs achieve a peak performance of 16 floating point operation per clock cycle. Table 3-3 compares our design with the data reported in [11] when 8 multipliers are used for each. Their design uses high performance floating point cores with clock latencies of 19 for the adder and 12 for the multiplier. The number of adders depends on the size of the reduction circuit, which changes with different matrices. For the test matrices in [11], the size of reduction circuit is 7. Our approach accepts any input matrices with no hardware changes required. There is no a priori analysis on the matrix or extra hardware initialization time needed for our design. For the tree-based design [11], zeros need to be padded when the number of nonzero in a row is not a multiple of the number of multipliers. To reduce the overhead caused by zero padding, [11] uses a technique called merging. As the PE number increases, the tree based design will face a choice between high overhead and complicated control logic [11]. Our design scales very easily and without increased overheads.

3.1.6 Potential Improvements

Parallelism: Reducing Summation Circuit Latency

In our design, the summation circuit is shared by all the PEs to add the data from the FIFOs and the result BRAM. When the design scales up, care must be taken that it will not become the bottleneck of the whole pipeline. That is, the time the result adder uses to

transport data should be overlapped by communication or computation time. There are 3 Result Adder circuits discussed here: reduced summation circuit, adder tree, and a one-clock-cycle latency adder, which take 8, 5, and 7 clock cycles to operate on each PE. We analyze this problem by considering the reduced summation circuit because it takes the most time. We compare the time the I/O and the summation circuit needs to transport data when each PE has 1 row. For a design with 8 PEs, the time needed by the result adder to transport data is $8 \times 8 = 64$ clock cycles.

The communication time is decided by the I/O bandwidth and matrix sparsity. On the Cray XD-1, the peak speed for the bus between FPGA chip and QDR II RAM is 1.6GB/s in each direction [16]. Suppose the matrix sparsity is 1% and sub-matrix size is 1000 by 1000. Then on average, there are 10 double precision floating point data (8 Bytes) for "*val*" and 10 integer pointer data (2 Bytes) for each "*col*", that is 100 Bytes per row. Even assuming the I/O bandwidth can be fully utilized with no other communication overheads, the communication time for the double precision floating point design is at least $100 \times 8 \times 165MHz / 1.6G \approx 83$ clock cycles for a design with 8 engines. The overhead for 8 PEs needs extra $10 \times 8 \times 165MHz / 1.6G = 8$ clock cycles and results in a total of 83+8=91 clock cycles.

If more PEs are implemented, the time spent by both the result adder and I/O operations increases linearly. Therefore, the I/O is the bottleneck instead of the adder tree under the conditions above. If the sub-matrix size increases, the time spent on I/O will increase accordingly. Therefore, the summation circuit has less possibility to become the bottleneck. If faster I/O is used, the time for I/O will be smaller and may not overlap the time for the summation circuit. Multiple summation circuits can work in parallel to increase the throughput until is the time on summation circuit can be overlapped by the communication or computation time.

Using Mixed Data Format for SSF

Given that I/O time is the performance bottle neck, reducing data transfer time will improve overall performance. We try to increase the performance of SSF by applying

Table 3-4: Comparison on 64 bit and 32/64 bit Mixed Integer SSF

Design	32/64 bit Mixed	64 bit
Achievable Frequency	183Mhz	175Mhz
Slices	3475 (10%)	8282 (25%)
BRAMs	20 (6%)	36 (10%)
MULT18X18	32 (9%)	128 (39%)
Multiplier Latency	4 cycles	6 cycles
I/O Bandwidth Requirement	8.8GB/s	14GB/s

shorter data formats as much as possible. The potential impact on performance is explained here by a simple example. Suppose 32-bit integers can provide sufficient resolution for the matrices and vectors given. The output data could be bigger and 64 bit integers are needed. Instead of using 64 bit data for both input and output data, we can use two different data formats: 32 bits for input and 64 bits for the output. Table 3-4 shows a mixed data format design has higher frequency, lower latency, and less I/O bandwidth and resources.

3.1.7 Performance

Performance Model for SSF Accelerator

In our design, the time for moving "*val*" and "*col*" into the FPGA is overlapped with the computation time. When a sub-matrix is being computed, the multiplicand vector x_i for the next matrices can be loaded. The I/O time on $x_i (i \geq 1)$ can be overlapped, so it is not counted here. The time for initialization and synchronization should also be counted, so the total time spent by the SSF core is

$$T = \max(T_{comp}, T_{IO}) + T_{init} + T_{sync} + T_{overhead} \quad (3\text{-}4)$$

In equation (3-4), the real computational work only contributes T_{comp} to the total time. To increase overall performance, we need to overlap the communication time and reduce the initialization and synchronization time besides reducing T_{comp}.

T_{comp} is determined by the frequency and number of computational engines. We assume F floating point operations are executed per second. The communication time is limited by the host memory bandwidth and by the I/O bus speed. Suppose the bandwidth for each I/O bus is B_{IO} and that the matrix *A* and vector *y* are transported by separate I/O buses. To compute a nonzero element, both its value and pointer have to be moved into FPGA. The time spent on the FPGA accelerator is thus

$$T = \max\left(\frac{2n^*_{nz}}{F}, \frac{n^*_{nz} \times (val\ datawidth + col\ datawidth)}{B_{IO}}\right) + T_{init} + T_{syn} + T_{overhead} \qquad (3\text{-}5)$$

Where n^*_{nz} is the total number of nonzero elements for all sub-matrices assigned to a FPGA accelerator.

To minimize equation (3-4) and (3-5), we have discussed several approaches to accelerate the computation: increasing the frequency and number of PEs to improve F; optimizing the matrix mapping to reduce I/O operations; making the design general to all different matrices so no hardware initialization or preparation on inputs is required; designing a simple interface which only needs a start signal and matrix/vector address; and not requiring any participation of the host during the computation.

We still need to discuss the block RAM size. The effect from the block size of x_i is a double edged sword. The overheads in our design mainly come from the one clock cycle control signal between rows. Therefore increasing the block size of x_i reduces the ratio of overheads by having more nonzero elements in each row. However, it also results in a longer initialization time for loading x_0. The result BRAM size determines the number of rows of sub-matrices, which affects the number of nonzero elements of sub-matrices.

Under certain sparsity, the I/O time to move sub vectors x_j can be overlapped with a big enough result BRAM size.

For very larges matrices, many sub-matrices will be assigned to a FPGA. T_{init} in our design comes from loading x_o and can be ignored in that case. The synchronization time with hosts is also just a function call, so we can also neglect T_{sync} for simplicity. For the double precision design, the data width is 8 Bytes. Because of the limited size of sub matrices, the pointer width is 2 Bytes. So equation (3-5) becomes:

$$T < \max(\frac{2n_{nz}^*}{F}, \frac{10n_{nz}^*}{B}) \tag{3-6}$$

If unlimited resources are assumed, F is also infinite. Then the achievable MFLOPS performance is limited by B.

$$MFLOPS = \frac{2n_z}{T} < \frac{2n_z}{10n_z / B} = B/5 \tag{3-7}$$

The floating point operations F take advantage of both the frequency and capacity of FPGAs and result in 4 times improvement every two years [6]. However to build a balanced system, the number of PEs is limited not just by chip capacity but also I/O bandwidth. For double precision floating point as discussed before, the maximum number of PEs that can be supported by the I/O bandwidth is a function of the I/O bandwidth B and the frequency f_{FPGA}:

$$\text{Number of PE } \leq \frac{B/5}{2f_{FPGA}} = \frac{B}{10f_{FPGA}} \tag{3-8}$$

Equation (3-8) shows that the number of PEs required for a computation system is constrained I/O bandwidth. If the I/O bandwidth of a system is 1GB/s and SSF runs at 100 MHz, only 1 PE is required to achieve the best performance.

Comparison with Previous Work

To the best of our knowledge, the work in [11] reports the highest previous performance for SpMxV on FPGAs. Given the same size design as shown in Table 3-2, we have similar peak performance and I/O requirements. However, our design does not need to change the hardware for different matrices, so the initialization and synchronization time is shorter. We also do not suffer from either high overheads or very complicated control logic when the system scales. For large matrices, the results from the design [11] are just for sub-matrices and need to be summed up for the final result. Our design allows storing the immediate result in the FPGA and computes the final result without this additional I/O operation requirement.

Comparison with Microprocessors

In our design, the overhead mainly comes from the one clock cycle between continuous rows. The number of these overhead clock cycles is decided by the total number of sub-rows. The initialization time is for preloading sub vector X_0. Both of these overheads can be found precisely in simulation. The synchronization time is affected by the interface and API between host and FPGA chip. Our design needs a few synchronization signals, such as "*start*", "*complete*" and "*start addresses*" of the matrices/vectors. The synchronization time is neglected at this point. We test our design on matrices from different fields as shown in Table 3-5. All these matrices come from Tim Davis' Matrix Collection [21]. They are roughly ordered by increasing irregularity. The percentage of overheads in the test matrices is shown in Figure 3-12.

We compare the performance of our design with microprocessors. Our design utilizes 8 PEs at 165 MHz frequency. The required memory bandwidth is 13.2 GB/s, which can be provided by current technology. For example, BenBLUE-V4 provides 16GB/s memory bandwidth [28]. We take a conservative performance estimation by deducting 40% off the peak performance for control overhead of the high speed memory interface [6], [11]. The achievable percentage of performance is shown in Figure 3-13.

Table 3-5: Test Matrices [21]

ID	Matrix	Area	Size (N)	Nonzeros (Nnz)	Sparsity (%)
1	Crystk02	FEM Crystal	13965	968583	0.5
2	Crystk03	FEM Crystal	24695	1751178	0.29
3	Stat96v1	linear programming	5995 x 197472	588798	0.05
4	nasasrb	Structure analysis	54870	2677324	0.09
5	raefsky4	Buckling problem	19779	1328611	0.34
6	Ex11	3D steady flow	16614	1096948	0.4
7	rim	FEM fluid mechanics	22560	1014951	0.2
8	goodwin	FEM fluid mechanics	7320	324784	0.61
9	dbic1	linear programming	43200 x 226317	1081843	0.01
10	Rail4284	Railways	4284 × 1092610	11279748	0.24

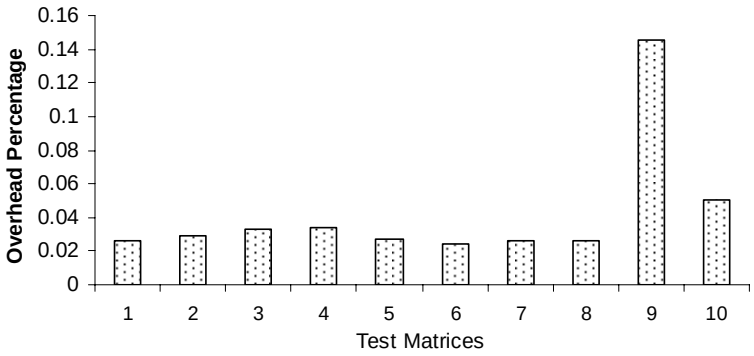

Figure 3-12: Overhead Percentage

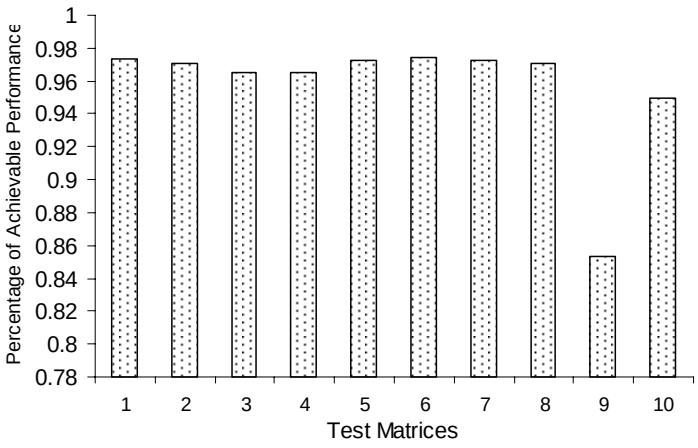

Figure 3-13: Percentage of Achievable Performance

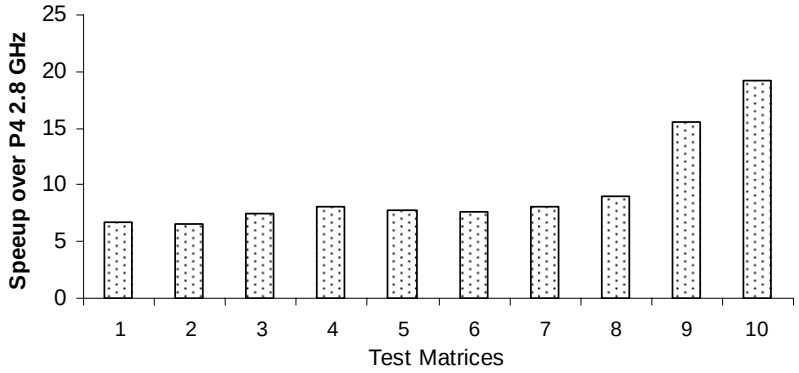

Figure 3-14: Speed Up of Our Design over 2.8 GHz Pentium 4

To test the software performance on a microprocessor, we use OSKI, which has achieved significant speedups by using techniques such as register and cache blocking [20]. The machine is a dual 2.8GHz Intel Pentium 4 with 16KB L1, 512KB L2 Cache and 1GB memory.

The speedup of our design over the 2.8 GHz Pentium 4 is shown in Figure 3-14. Our design performs better than the Pentium 4 on matrices with irregular sparsity structures. This is because the overhead of our design depends on the number of nonzero elements per row of sub-matrices but is not affected by their sparsity structure.

The high performance of our SSF design relies in the reduced overhead, deep pipeline, and a parallel architecture. First, SSF fully controls data required for computations and therefore avoids the high penalty of cache misses in traditional CPUs. Second, SSF uses a deeply pipelined architecture and maximally reduce idle pipeline stages. Third, multiple PEs are implemented in SSF to achieve parallelism.

3.2 Matrix Factorization

3.2.1 LU Decomposition Design on FPGAs

LU decomposition is a widely used matrix factorization algorithm. It transforms a square matrix A into a lower triangular matrix L and an upper matrix U with A=LU. The elements of A, L, and U can be denoted as $a_{x,y}$, $l_{x,y}$, and $u_{x,y}$, respectively. As shown in the following steps, the Dolittle algorithm for LU does the elimination column by column from left to right. It results in a unit lower triangular matrix and an upper triangular matrix which can use the storage of the original matrix *A* [15]. This algorithm requires $2n^3/3$ floating point operations.

Step 1: Column Normalization. The elements $a_{x,0}$ in the first column below the diagonal element $a_{0,0}$ are divided by $a_{0,0}$.

Step 2: Sub-matrix Modification. The product of $l_{x,0}$ and the row vector $a_{0,x}$ (also $u_{0,x}$), is computed and subtracted from each row of the sub-matrix $a_{x,y}$, where $(1 \le x, y \le n-1)$.

Step 3: Steps 1 and 2 are recursively applied to the new sub-matrix generated in step 2. During the k^{th} iteration, $l_{x,k}$ and $u_{k,y}$ $(k+1 \le x, y \le n-1)$ are generated. The iterations stop when $k = n-1$.

When matrix *A* is not positive definite, columns could be divided by a small number or even zeros, and cause inaccuracy. To avoid this problem, partial pivoting is applied to swap columns in sub-matrices. We first assume matrix *A* is positive definite and no pivoting is needed to compare to current LU design on FPGAs. Partial pivoting will be discussed in the next section.

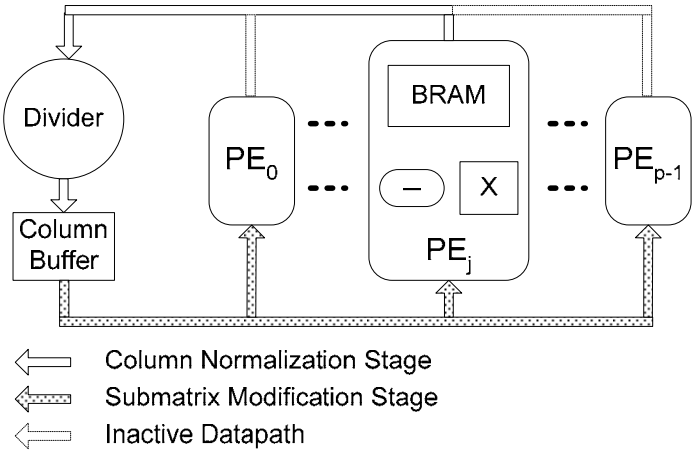

Figure 3-15: Base Diagram for Direct LU Decomposition

As shown in Figure 3-15, our design mainly consists of a divider, a column buffer, and p PEs. In each PE there is a multiplier, an adder, and local memory. The maximum number of PEs and their local memory size are limited by available resources of the FPGA chip.

The process to complete LU decomposition by our design has 4 stages: *matrix input*, *column normalization*, *sub-matrix modification*, and *completion*. As shown in the LU algorithm, stage 2 "*column normalization*" and 3 "*sub-matrix modification*" are executed iteratively until the sub-matrix becomes a scalar. To fully fill the deep pipelines of floating point units, a streaming architecture is used. Initially matrix A is stored in the PEs' local memory. In each stage, data flow out of the memory, through the arithmetic engines for computation, and finally return back to the memory. According to the LU decomposition algorithm, the data path configurations are different for different stages. To maximally reuse these expensive floating point units and memory, high speed switches are used to change the connection between these components for different stages.

At the "*matrix read stage*", the input data and address ports of the PE local memory are connected to the PEs' local memory input ports. These local memories appear to the host as a big memory block by address mapping. The matrix is striped to PEs by columns with

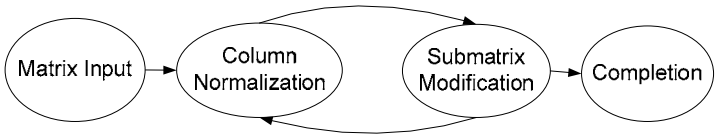

Figure 3-16: LU Design Operation Stages

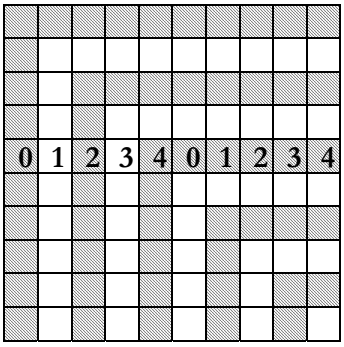

Figure 3-17: Matrix Mapping on FPGAs

column $n/p+j$ stored in PE j as shown in Figure 3-17. Because the sub-matrices become smaller and smaller in the iterative stages, such a storage format ensures that the sub-matrices are evenly distributed among the PEs for parallel computation. Without loss of generality, we assume the matrix size is an integer multiple of PE number.

During the "*column normalization" stage*, the column "col_0" flows out of its local PE storage, through the divider, to compute $a_{k,0}/a_{0,0}$ $(0<k\le n-1)$. The computed results l_0 are stored in the column buffer and the appropriate PE's local memory at the same time. In the "sub-matrix modification" stage, $a_{x,y}-l_{x,0}a_{0,y}$ needs to be computed for $1\le x,y\le n-1$. The data flow configuration inside a PE is shown in Figure 3-18. l_0 flows out of column buffer and trough all the multipliers in the PEs simultaneously. At the same time, its address flows through the PEs' local memory to address $a_{x,y}$ and $a_{o,y}$ by

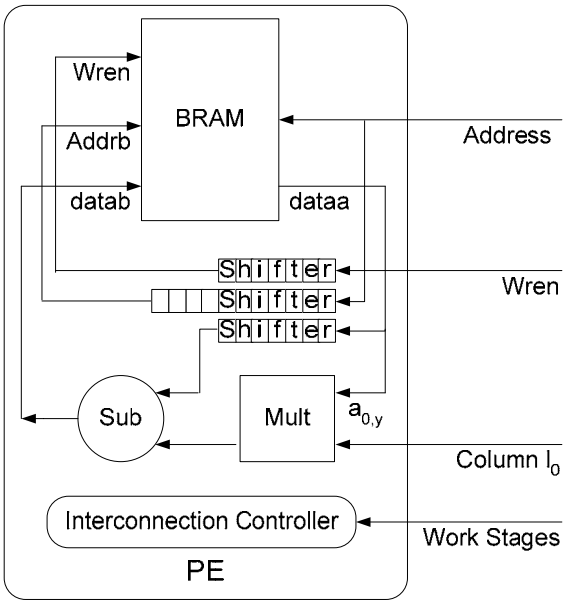

Figure 3-18: PE Data Flow Configuration at Sub-matrix Modification Stage

inserting proper delays. All PEs perform sub-matrix update simultaneously in this stage. In each clock cycle, one floating point addition and multiplication are executed for each PE. Because multiple columns are stored in one PE, column l_0 should also circulate multiple times until all the columns are updated.

Design analysis: If the few overheads due to the control flow are not counted, the proposed design completes LU decomposition in approximately $n^3/3p$. Just n words are needed for the storage besides the original matrix's own space.

Proof: In iteration k ($0 \le k < n-1$), the sub-matrix size is n-k. The clock cycles needed by the divider to compute $l_{k,x}$ ($k \le x < n-1$) is $(n-k-1)$, while that for multiplication and subtraction is $2(n-k-1)^2$. So the total divider operation is $\sum_{x=1}^{n-1} x = n(n-1)/2$ and that for multiplication and subtraction is $\sum_{x=1}^{n-1} 2x^2 = 2n^3/3 - n^2$. Because the multiplication and subtraction are overlapped and computed by p PEs in parallel, so the total time is $n(n-1)/2 + (2n^3/3 - n^2)/2p \approx n^3/3p$. The addition and multiplication floating point

operations for LU decomposition are in order of $O(2n^3/3)$. More accurate execution time analysis for LU decomposition is discussed in chapter 6.

3.2.2 Pivoting

When zeroes exist on sub-matrix diagonals, elements will be divided by zeros in the LU decomposition algorithm discussed above. At the same time because computers have to use certain precisions, relatively small values on sub-matrix diagonals will possibly cause big accumulated numeral errors [81]. Pivoting is a process performed on a matrix to increase numerical stability. The element $a_{0,0}$ used in "*column normalization*" stage is called a pivot element. The row having the pivot element is called the pivot row. There are numerous pivoting methods discussed in the literature. We list some of them here to give a general idea.

(1) Trivial Pivoting. The trivial pivoting strategy is as follows. Locate the first row *j* below 0 in which $a_{j,0} \neq 0$ and then switch rows *j* and 0. This will result in a new element $a_{0,0} \neq 0$, which is a nonzero pivot element.

(2) Partial Pivoting. The partial pivoting strategy is as follows. If $a_{0,0} = 0$, locate row *j* (*j>0*) that has the maximum absolute value in column 0 and then switch rows *j* and 0. This will result in a new element $a_{0,0} \neq 0$, which is a relatively big pivot element. In partial pivoting, only row permutations are employed. The strategy is to switch the largest entry in the pivot column to the diagonal.

(3) Total Pivoting. The total pivoting strategy is as follows. Locate row *j* (*j>0*) and column *k* (*k>0*) where element $a_{i,j}$ has the biggest absolute value. Then first switch rows 0 and *j* and second switch column 0 and *k*. This will result in a new pivot element $a_{0,0} \neq 0$. This is also called "*complete pivoting*" or "*maximal pivoting*." Here both row and column permutations are permitted. The strategy is to switch the largest entry in the part of the matrix that we have not yet progressed to the diagonal.

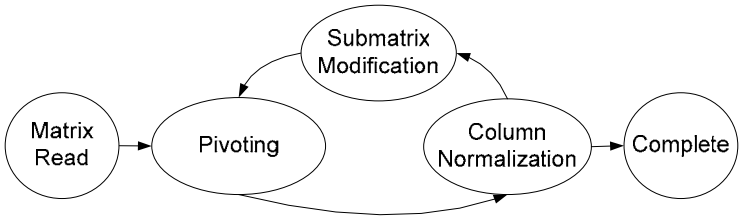

Figure 3-19: LU Design Operation Stages with pivoting

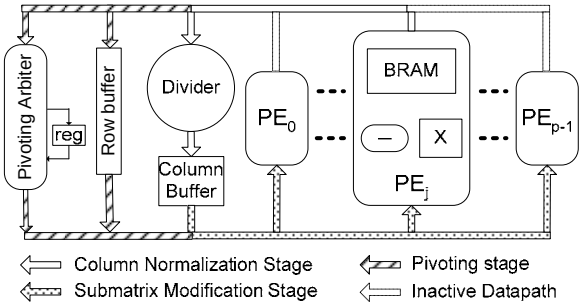

Figure 3-20: Base Diagram for Direct LU Decomposition with pivoting

Compared to other strategies, partial pivoting effectively reduces numerical errors without large computational overheads. Therefore it is employed in the hardware design here. As shown in Figure 3-19, pivoting is performed after a new matrix is imported or each time when a sub-matrix is completely updated. A column normalization operation is executed after pivoting to avoid dividing column 0 by zeroes or relatively small numbers.

The system diagram with partial pivoting is shown in Figure 3-20. During the "*pivoting*" stage, the first sub-matrix column is streamed out from the PEs to the pivoting arbiter, which compares the pivot element with other values in this column. If the pivot element is the biggest value in the column, no pivoting is required. Therefore the state machine transfers to the next stage "*column normalization*" directly. If the pivoting arbiter finds a value in column 0 bigger than the pivot element, a pivoting operation has to be performed. The biggest element in column 0 is the new pivot element, while the row that has the new pivot element will be the new pivot row. Because matrix A is stored in the PEs' local

memories by column, a row of matrix A is distributed in all PEs. The values in the old and new pivot row are exchanged in all PEs simultaneously. The pivoting buffer is used to temperately store the old pivot row when exchanging two rows.

3.2.3 Implementation Results

We implement the LU design on Xilinx FPGA XC2VP50-7 FPGA, which is used on the Cray-XD1 supercomputer as an application accelerator. Table 3-6 and Table 3-7 give implementation results with and without pivoting, respectively. When the same number of PEs and same size of maximum matrix size are implemented for double and single precision, the latter costs less than half of the slices, BRAMs, and embedded 18x18 multipliers. By using a similar number of total slices, a Xilinx XC2VP50 FPGA can accommodate 8 PEs for double, 16 PEs for s31e8, and 32 PEs for s16e7. At this case, s31e8 and s16e7 can hold larger matrix sizes than double precision. The achievable frequencies are tested from a Cray-XD1 supercomputer. The specific design for the Cray-XD1 and execution time performance will be discussed in Chapter 5.

3.3 Hybrid Direct Solver

LU decomposition is widely used for direct solution of linear systems. Suppose matrix A is factored to a lower triangular matrix L and upper triangular matrix U. The linear system becomes $LUx = b$. It is equivalent to solve two linear equations $Ly = b$ and $Ux = y$. Since L and U are triangular matrices, y and x can further be solved by forward and backward substitutions.

Table 3-6: LU Decomposition Implementation with Pivoting on XC2VP50-7

Design	Double (s52e11)	Single (s23e8)	s31e8	s16e7
Number of PEs	8	8	16	32
Maximum size	128	128*	128	256
Achievable Frequency	120MHz	135MHz	130MHz	140MHz
Slices	21044 (89%)	9091 (38%)	20356 (86%)	20907 (88%)
BRAMs	84 (36%)	42 (18%)	84 (36%)	130 (56%)
MULT18X18	128 (55%)	32 (13%)	64 (27%)	32(13%)

- A larger matrix size can be accommodated

Table 3-7: LU Decomposition Implementation without Pivoting on XC2VP50-7

Design	Double (s52e11)	Single (s23e8)	s31e8	s16e7
Number of PEs	8	8	16	32
Maximum size	128	128*	256	256
Achievable Frequency	120MHz	135MHz	130MHz	140MHz
Slices	20422(86%)	7737 (32%)	19070(80%)	19575(82%)
BRAMs	68 (29%)	34 (14%)	148 (63%)	97(41%)
MULT18X18	128 (55%)	32 (13%)	64 (27%)	32(13%)

- A larger matrix size can be accommodated

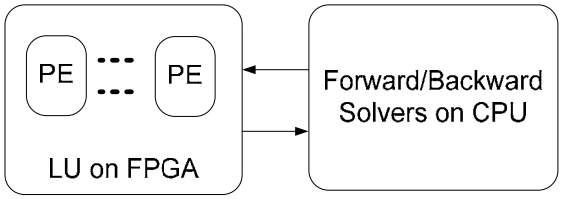

Figure 3-21: A Hybrid Structure for Direct Solver

$$
\begin{aligned}
& y_0 = b_0 / l_{0,0} \\
& y_1 = (b_1 - l_{1,0} y_0) / l_{1,1} \\
& \cdots \\
& y_{n-1} = (b_{n-1} - \textstyle\sum_{j=0}^{n-2} l_{n-1,j} y_j) / l_{n-1,n-1}
\end{aligned}
\tag{3-9}
$$

$$
\begin{aligned}
& x_{n-1} = y_{n-1} / u_{n-1,n-1} \\
& x_{\mathbf{n}\text{-}2} = (y_{n-2} - u_{n-2,n-1} x_{n-1}) / u_{n-2,n-2} \\
& \cdots \\
& x_0 = (y_0 - \textstyle\sum_{j=n-1}^{1} u_{0,j} x_j) / u_{0,0}
\end{aligned}
\tag{3-10}
$$

Equations (3-9) and (3-10) could be implemented on FPGAs, but complicated control logic is required to achieve fine parallelism. Furthermore, the division operation of each iteration cannot be parallelized and also requires an expensive floating point divider. On the other hand, the computational complexity for equations (3-9) and (3-10) is $O(n^2)$, while that for the LU decomposition is $O(n^3)$. Therefore, we propose to explore LU decomposition on FPGAs but leave the forward and backward substitutions on the CPU as shown in Figure 3-21.

3.4 Conclusion

The first design for our RBLAS library is an innovative SpMxV FPGA design with overall system performance addressed. First, we introduce an improvement for traditional BCRS, which results in lower I/O requirements and less overhead. Secondly, we propose an

efficient multiplication accumulation circuit for pipelined floating points by taking advantage of the data flow. Compared to previous work, our design has higher peak performance, lower memory requirements, better scalability, and does not need to reconfigure hardware for different matrices.

The second design is an *LU* decomposer on FPGAs. To maximally utilize the floating point units, flexible interconnections are implemented by using high speed switches. During each stage, the data is streamed out of memory and then through floating point units. The results are computed and flowed back to the memory. For non positive definite matrices, the LU algorithm requires partial pivoting which involves complicated control logic design. This is the first HDL design implementing partial pivoting architecture for LU decomposition on FPGAs.

Based on the LU design, we propose a hybrid structure for a direct solver. The LU decomposition has a computation complexity of $O(n^3)$, so it is mapped to FPGAs for fined parallel computation. The forward and backward substitutions have just $O(n^2)$ computational complexity but require expensive floating point units and complicated control logic, so are left in the CPU.

4 Mixed-Precision Linear Solver on FPGAs

Floating point linear equation solvers are widely used in scientific computations such as the finite element method (FEM) and partial differential equation solvers. For the purpose of algorithm convergence and accuracy, a double precision data format is often used in software codes for these algorithms. Recently research has targeted to accelerating these applications on FPGAs and achieved promising results. To achieve speedup via parallelism, FPGA designs require multiple floating point units to be implemented. However due to their very high resource cost, current FPGAs can only accommodate a very limited number of double precision floating point units.

Shorter formats on FPGAs usually result in higher frequency and lower resource consumption. Meanwhile using smaller data sizes also helps to reduce the bus traffic. Therefore, it makes a lot of sense to use shorter and shorter formats for higher performance when the accuracy allows. For example, fixed point data are widely used in digital signal processing to take the place of floating point units. The problem for these approaches is that the accuracy is usually decreased. Therefore, analysis must be performed to guarantee that the lower-precision data format is accurate enough for certain applications. We propose to explore mixed-precision data algorithms on FPGAs, which can achieve higher performance by adopting lower-precision data formats without losing accuracy [14].

4.1 Mixed-Precision Algorithm for Direct Solver

4.1.1 Iterative Refinement

Suppose matrix A can be factorized as $PA = LU$ with partial pivoting, where L is a lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix used for pivoting. The direct solver with iterative refinement is shown in Figure 4-1, where refinement loops are taken to improve the accuracy based on the available solution. Demmel pointed out that the iterative refinement process is similar to Newton's method

```
Factorize A to LU: PA=LU.
Sove LUx=Pb
while (r is too big & maximum loop not reached)
        r=b-Ax
        Solve Ly=Pr
        Solve Uz=y
        x=x+z
end
```

Figure 4-1: Direct Solver with Iterative Refinement

applied to $f(x) = b - Ax$. If all the computations were done exactly, it would be done in one step [42].

The idea of this mixed-precision algorithm is that the factoring $PA=LU$, and the triangular solver $LUx = Pb$ are computed in lower-precision; while the residual and updating of the solution will be computed in higher-precision. This approach was analyzed by Wilkinson [46] and Moler [47], who showed that this algorithm produces a computed solution correct to the working precision, provided matrix A is not too ill–conditioned. Demmel [48] pointed out that the behavior of the method depends strongly on the accuracy with which the residual is computed.

The potential performance gain of using the mixed-precision algorithm lies in that the computation on factorization is $O(n^3)$ and dominates the runtime of the algorithm in Figure 4-1. The other steps, including triangular solver, residual computation, and the solution update, are just $O(n^2)$. Furthermore, shorter data formats usually reduce the memory bandwidth requirement.

4.1.2 Error Analysis

Previous work addressed error analysis of iterative refinement techniques. Higham derived error bounds for fixed precision iterative refinement [82]. For single/double mixed-precision iterative refinement executing the refinement in double precision arithmetic, [82] gives error bounds in single precision. Stewart gives an error analysis of

iterative refinement [49]. Langou et al derived the results from [49] and give error bounds in double precision for a single/double mixed-precision algorithm with iterative refinement performed in double precision arithmetic [14]. The result in [14] reveals that a mixed-precision algorithm can achieve the same accuracy as with higher-precision, provided that the matrix is not too badly conditioned.

Data formats utilized in our design are much more flexible than single and double precision, so we extend the results of [14] for iterative refinement methods performed in general high/low mixed-precision arithmetic. We consider mixed-precision iterative refinement algorithms in Figure 2-1 which execute steps 3 and 5 in higher-precision ε_{high} but the other steps in lower-precision ε_{low}. If the matrix A is not too-ill conditioned with respect to the lower-precision arithmetic, that is $\psi_k(n)\kappa(A)\,\varepsilon_{low}<1$, from the results in [14], we have

$$\frac{\|b - Ax_{k+1}\|}{\|A\| \cdot \|x_{k+1}\|} \leq \alpha_B \cdot \frac{\|b - Ax_k\|}{\|A\| \cdot \|x_k\|} + \beta_B \tag{4-1}$$

where α_B and β_B are of the form,

$$\alpha_B = \psi_B(n)\kappa(A)\varepsilon_{low} \tag{4-2}$$

and

$$\beta_B = \rho_B(n)\varepsilon_{high} \tag{4-3}$$

$\psi_k(n)$, $\psi_B(n)$ and $\rho_B(A)$ are small functions of *n* explicitly defined in [14]. α_B is depends on $\kappa(A)$ and $\varepsilon_{low,}$ which are the condition number of the matrix A and the implemented lower-precision. α_B indicates the convergence rate. β_B depends on the higher-precision used ε_{high}, and determines the limiting accuracy of the algorithm. At convergence, the following exists:

$$\lim_{k\to\infty}\frac{\|b-Ax_k\|}{\|A\|\cdot\|x_k\|}=\beta_B(1-\alpha_B)^{-1}$$
$$=\frac{\rho_B(n)}{1-\psi_B(n)\kappa(A)\varepsilon_{low}}\varepsilon_{high} \tag{4-4}$$

This indicates that the same normwise accuracy is achieved for the mixed-precision algorithm as for the higher-precision.

4.2 Performance on FPGAs

Given the fact that FPGAs have much more flexible data formats than traditional processors, it is valuable to find out the data formats optimal for both accuracy and performance. More specifically, simpler and shorter data formats help to increase the frequency and reduce resource cost and bus bandwidth requirements. On the other hand, using lower-precision for LU factorization might require more refinement iterations and may even fail to converge. As an example, we test the convergence and iteration loops of a mixed-precision direct solver using double precision (s52e11) and a customized format (s16e8). The refinement stops either when the solver achieves the accuracy of the double precision algorithm or there are more than 30 iterations. The latter is considered to be a failure of convergence. Table 4-1 shows the results tested on 100 random matrices. When the problem size increases, we observe that more iteration loops are required. Note that for large problems, the refinement takes a very small percentage ($O(n^2)/O(n^3)$) of the overall time, so a small increase in the number of iterations will have little performance impact. In Table 4-2, we tested the number of iterations required for different data formats for same matrices shown in Table 4-1. The data precision is decided by the mantissa, so the exponent is not listed in Table 4-2. The number of iterations increases from right to left and from top to bottom, with convergence failures at the lower left.

Table 4-1: Average Refinement Iterations for a Customized Format (s15e7)

Problem size (n)	Average condition number	Average iterations	Variance
128	913	4	0.24
256	1818	5.1	0.48
512	4017	6.1	3.36
1024	6196	6.3	5.16
2048	9407	9.3	12.21
4096	22425	13.3	22.6

Table 4-2: Average Refinement Iterations for Different Data Formats

Mantissa Bits / Problem Size	12	16	23	31	48	52
128	8.9	4	2	1	1	0
256	11.1	5.1	2.1	1	1	0
512	19.7	6.1	2.5	1	1	0
1024	28	6.3	2.6	1	1	0
2048	-	9.3	3	1.3	1	0
4096	-	13.3	3.1	1.43	1	0

Modern FPGAs utilize embedded circuits for higher performance. One example is the embedded DSP48 blocks in the Xilinx Virtex 4 FPGA families. Because each DSP48 can be configured as an 18 by 18 multiplier (including sign bit), data formats wider than 18 bits require multiple embedded units. Therefore, designs using embedded multipliers might result in significant resource savings by selecting suitable data formats, such as those highlighted in Table 4-3. All frequency reports here come from Xilinx place and route tools, with the Place and Route Effort Level high. If we assume all the DSP48s are configured as multipliers, the GFLOPs performance can be computed by multiplying the number of multipliers available and the frequency. Table 4-4 and Table 4-5 show the characteristics of implementing one multiplier or one adder by using slices. To compute the FPGA GFLOPs performance for adders and multipliers, we assume only 70% of the slices can be configured as multipliers or adders, the rest are used for other circuits and routing. This is a reasonable assumption according to previous linear algebra designs on FPGAs [27].

We find that the FPGA GFLOPs performance increases significantly by using shorter data formats. One reason is that shorter data formats reduce the resource cost and therefore more floating point operators can be implemented. At the same time, shorter formats reduce the memory space and bus bandwidth. This is crucial to linear algebra applications, which usually require frequent data movements. Finally, using shorter formats also reduces the latency of floating operators, which is also an important factor for the performance of linear algebra design.

4.3 A Reconfigurable Mixed-Precision Direct Solver

The direct solver we proposed can be used for the mixed-precision algorithm as shown in Figure 4-2. A lower-precision version of the matrix A is moved from the CPU main memory to the FPGA for LU decomposition. The CPU computes the solution using lower-precision LU matrices but computes the residual and updates the solution in higher-precision.

Table 4-3: Characteristics of a Multiplier on XC4LX160-10 (Using DSP48s)

Data Formats	DSP48s	Frequency (MHz)	Latency	GFLOPs
s52e11 (double)	16/96	237	21	1.42
s51e11	16/96	238	21	1.43
s50e11	**9/96**	245	19	2.61
s34e8	9/96	289	14	3.08
s33e8	**4/96**	292	9	7.01
s23e8 (single)	4/96	339	9	8.14
s17e8	4/96	370	9	8.88
s16e8	**1/96**	331	6	31.78
s16e7	1/96	352	6	33.79
s13e7	1/96	336	6	32.26

Table 4-4: Characteristics of a Multiplier on XC4LX160-10 (Using slices)

Data Formats	Slices	Frequency (MHz)	Latency	GFLOPs
s52e11 (double)	1392/67584	184	9	6.25
s51e11	1368/67584	184	9	6.36
s50e11	1326/67584	191	9	6.81
s34e8	656/67584	199	8	14.35
s33e8	644/67584	207	8	15.21
s23e8 (single)	388/67584	286	8	34.87
s17e8	274/67584	265	7	45.75
s16e8	237/67584	283	7	56.49
s16e7	233/67584	257	7	52.18
s13e7	185/67584	343	7	87.71

Table 4-5: Characteristics of an Adder on XC4LX160-10

Data Formats	Slices	Frequency (MHz)	Latency	GFLOPs
s52e11 (double)	778/67584	235	12	14.29
s51e11	772/67584	239	12	14.65
s50e11	754/67584	245	12	15.37
s34e8	531/67584	278	12	24.77
s33e8	510/67584	268	12	24.86
s23e8 (single)	380/67584	287	11	35.73
s17e8	314/67584	278	11	41.88
s16e8	301/67584	309	11	48.57
s16e7	293/67584	266	11	42.95
s13e7	244/67584	287	10	55.65

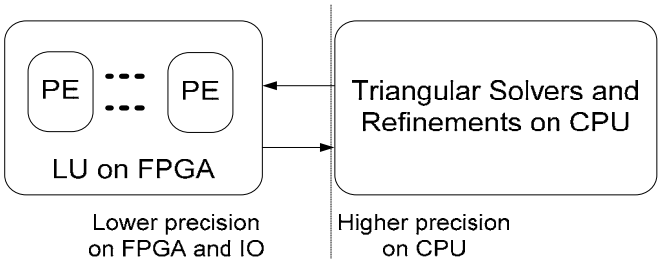

Figure 4-2: Structure for Mixed-Precision Direct Solver

4.4 Performance Summary

The execution time of our design in Figure 4-2 consists of four parts: the time for the LU decomposition, iterative refinement, forward/backward triangular solver, and communication. As we discussed before, the clock cycles required for our LU design is $n^3/3p$, where p is the number of PEs. The frequency f of the LU decomposition design depends heavily on the data formats. The communication time is associated by the data movements between the FPGA and the CPU main memory, so it is determined by the matrix size (n^2), data width (w), and bus bandwidth (B_{bus}). Finally, the time for iterative refinement depends on the number of iterations (I_{ref}) and the time for each loop (T_{ref}). So the total time can be described as:

$$\begin{aligned} T_{mixed} &= T_{LU} + T_{comm} + T_{tri} + T_{refinement} \\ &= \frac{n^2}{3pf} + \frac{2n^2}{B_{bus}} w + T_{tri} + I_{ref} T_{ref} \end{aligned} \tag{4-5}$$

Using a smaller data format could significantly increase the number of PEs and the frequency and reduce the data width, so the first two terms will be greatly decreased. On the other hand, the number of refinement iterations would likely increase as shown in Table 4-2. Because T_{ref} is relatively small, the impact from the third term is dominated by the first two terms. The computation for the triangular solver is $O(n^2)$ which is much less than

for the LU decomposition. The architecture for this mixed-precision solver on the Cray-XD1 supercomputer will be introduced in Chapter 5.

5 Design on Cray XD1

This chapter describes the implementation our hybrid direct solver on the Cray-XD1 supercomputer which utilize FPGAs as application processors. First, we introduce a general architecture and development background for the Cray-XD1 supercomputer. Second, both hardware and software implementations of our hybrid direct solver on the Cray-XD1 supercomputer are introduced. The performance of the hybrid solver is also tested and compared to CPUs.

5.1 CRAY XD1 Introduction

5.1.1 Architecture Overview

The Cray XD-1 supercomputer incorporates reconfigurable computing devices as accelerators to deliver significant speedup of targeted applications [16]. The basic architectural unit of the Cray XD1 system is the Cray XD1 chassis, which can contain one to six compute blades. Each compute blade includes two 64-bit AMD Opteron processors configured as a two-way symmetric multiprocessor (SMP) that runs Linux. 1 to 8 GB DDR can be assigned to each compute processor. FPGAs can be adopted as coprocessors by adding an expansion module on the compute blade. Processors, FPGAs, and memory within a chassis and between chasses are linked by a high-speed switch fabric called the RapidArray interconnect. Besides the main memory, each FPGA module contains four QDR II SRAMs as high-speed storage. The programmable clock enables the user to set the speed of the FPGAs [50]. The Cray XD1 machine at ORNL (Tiger) has 12 chasses containing 144 Opteron processors and 6 Xilinx XC2VP50-7 FPGAs .

5.1.2 RapidArray Interconnect

The high-bandwidth, low-latency RapidArray interconnect is the central organizing construct of the Cray XD1, which enables the system to avoid bus bottlenecks and shared-resource contention. The Cray RapidArray Transport (RT) core provides the

RapidArray fabric interface to an FPGA design. To facilitate different applications, the RT core has two interfaces: fabric request and user request. The fabric request interface issues read/write requests from the rest of the Cray system to the user logic, while conversely the user request processes requests from the user logic. Currently the Cray XD1 only supports access to the local processor. The RT interface provides a 64-bit interface at a maximum speed of 200 MHz, which yields a bandwidth of 1.6GB/sec for simultaneous transmit and receive. For applications with heavy data movement, the RT core provides data bursts, which can be up to 64 bytes per request [50], [51].

The FPGA is accessible via a 128 MB region of the HyperTransport I/O address space. Any HyperTransport read/write from the SMP to this region is directed to the RT interface of the FPGA which passes them on to the user logic. The Cray XD1 provides API functions for processors to communicate with FPGA applications. More specifically, it supports both SMP-initiated requests and FPGA-initiated requests. The SMP can initiate requests in two ways: I/O mapped access and read/write functions. The main difference lies in that the I/O mapped access takes advantage of "*write combining*", which improves the performance of write accesses from the SMP to the FPGA by combining multiple write accesses into a single HyperTransport packet [50].

5.1.3 HDL Development Flow

The Cray XD1 uses standard development processes and tools for FPGA development. FPGA IP cores are used to provide the interface between user applications and Cray System. As mentioned before, the RT core provides the interface between user application and the RapidArray, while the QDR core is used for connecting the user application and the QDR II SRAMs. These IP cores need to be integrated with the user design during the FPGA implementation process. The binary file from place and route needs to be converted to a Cray-proprietary format file by adding frequency and other information before downloading to the FPGAs. A typical application on the Cray XD1 is illustrated in Figure 5-1. The top-level VHDL file contains several logic components: user application, RT core, QDR core, and a user-programmable clock generator.

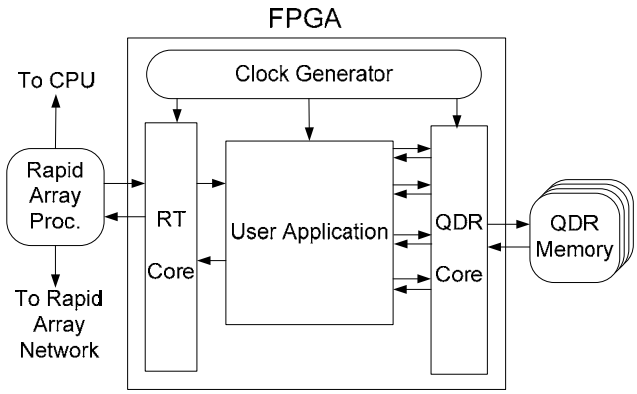

Figure 5-1: Cray XD1 FPGA Organization [50]

5.2 Hybrid Direct Solver on Cray XD1

5.2.1 Hardware Architecture

As shown in Figure 5-2, the hybrid solver top level architecture consists of the RT Client, Register Interface, and LU Interface block. QDR memory is not used in this application, so QDR core is disabled to save resources and power. The RT Core is a standard IP block provided by Cray to enable communication with other devices over the RapidArray fabric. The Register Interface block provides a set of readable and writeable interface registers, which are used to communicate between host and LU decomposition kernels. The LU Interface block contains all function units for LU decomposition. The LU Interface block appears to the host processor as a large block of memory. Appropriate internal BRAMs are mapped to the User Interface ports by internal control logic according to different operation stages. For example, all PEs' local BRAMs are combined as a big memory block when the matrix is transported from main memory to the FPGA local memory. Addresses of PEs are properly arranged so that the input matrix is mapped into all PEs by columns as shown in Figure 3-17. The Decoder block in the LU Interface is used to interpret signals between the Register Interface and LU Interface.

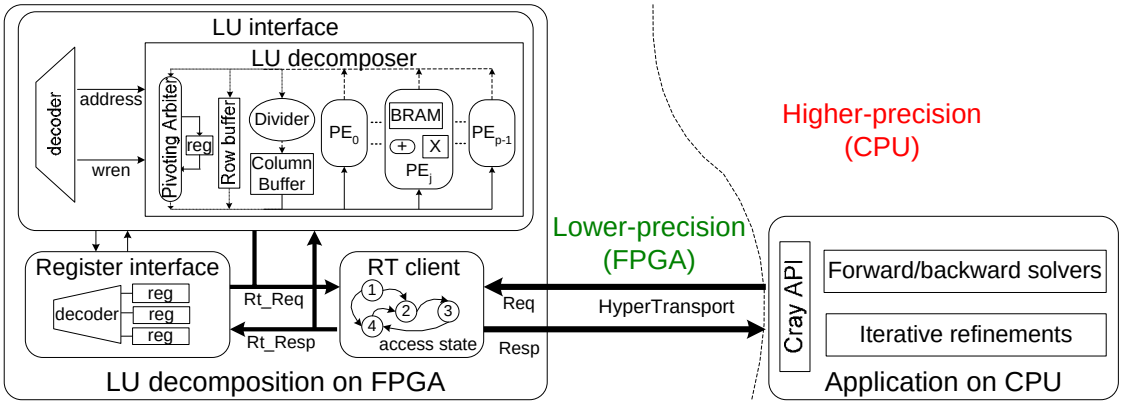

Figure 5-2: Hybrid Mixed-Precision Direct Solver on Cray-XD1

For our direct solver design, the original matrices are located in the processor main memory. The complete matrix is moved into the FPGA for LU decomposition and then moved back to the main memory after the required operations are completed.

5.2.2 System Hierarchy

The hybrid solver is co-designed in C and VHDL. C is used for the host programs, while VHDL configures the hardware for the FPGA accelerator. The Cray FPGA API library is utilized to communicate between the C program and FPGA kernel. The file hierarchy is shown in Figure 5-3. The top level of the software program is a hybrid solver, which has LU decomposition and forward/backward solvers in double precision. When a matrix is assigned to the FPGA accelerator, the hybrid solver calls FPGA interface functions to communicate with the FPGA hardware. Test matrices are stored in separate files and can be loaded by the hybrid solver's I/O functions. The software also provides functions to record matrix solver and performance analysis results such a number of iterations or execution time.

The final binary file to configure the FPGA accelerator is "*top.bin.ufp*", which combines the FPGA configure file "*top.bin*" and Cray configuration file "*ufphdr*". The file "*ufphdr*" provides Cray Part Number and FPGA frequency information. The top level logic design is in "*user_app.vhd*"which includes several components: LU Interface, Register Interface,

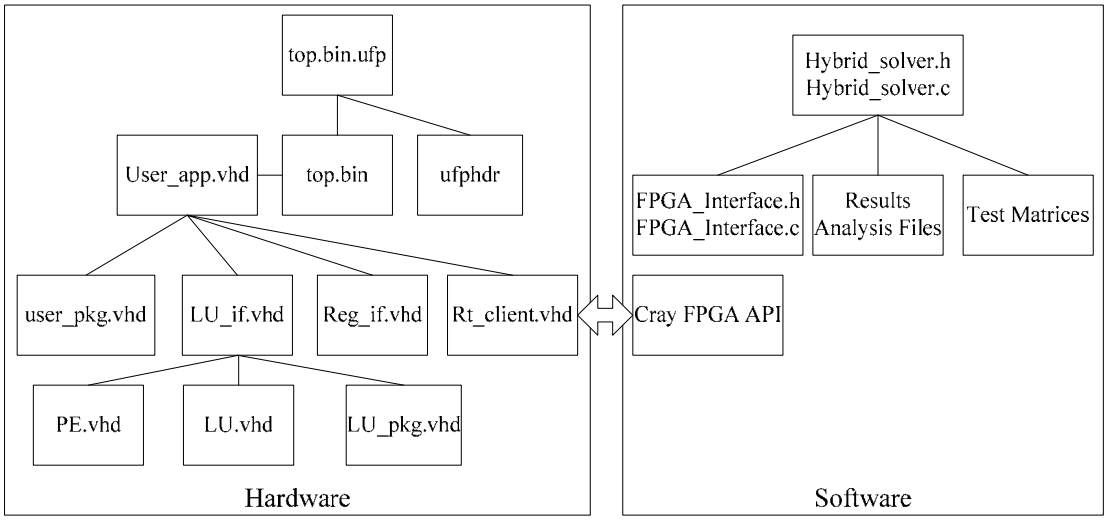

Figure 5-3: Hybrid Mixed-Precision Direct Solver on Cray-XD1

and RT Client. The system uses a parameterized design. All Cray parameters are included in "user_pkg.vhd", while parameters for LU decomposer are included in "LU_pkg.vhd".

5.2.3 Implementation Results and Performance Comparison

The hardware implementation results for LU decomposition are listed in Chapter 3. Other logic circuits for Cray IP cores total around 5% extra slices. No previous FPGA designs for LU decomposition have considered mixed-precision data formats, so we just compare our double precision design with previous work. In [39] LU decomposition is implemented on multiple processors on a FPGA, and its architecture is very different from ours. The architecture in [40] limits the problem size by the number PEs, and cannot scale to big matrices. [38] improves the design of [36], and implements the LU algorithm using circuits as with our work. In both [38] and our work, the matrix size is not limited by the number of PEs but by BRAM size. Block LU decomposition algorithms can be used for large matrices which exceed the FPGA on-chip BRAM size. If we target our work onto Xilinx XC2VP100 to compare our results to [38], 18 PEs can be implemented. Therefore our design is very similar to [38] as far as resource cost. However, unlike [38] our design implements pivoting algorithm which requires some additional slices.

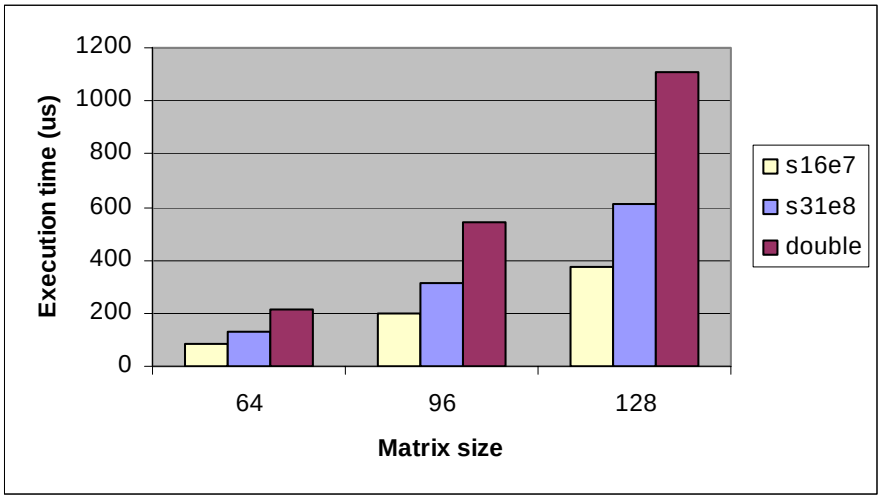

Figure 5-4: Performance comparison of LU design

Our work accelerates the performance of direct solvers by mapping LU decomposition onto FPGAs and taking advantage of the high performance of lower-precision arithmetic. Therefore we first test the performance of our LU decomposition designs with different data formats. As shown in Figure 5-4, the LU decomposition execution time for lower-precision designs is much less than for higher-precision designs.

The test matrices here are randomly generated with all elements following a Gaussian distribution as shown in Table 4-1 and Table 4-2. As shown in equation (4-5), the execution time for our mixed-precision solver consists of four components: LU decomposition, iterative refinement, forward/backward triangular solver, and communication. The average execution time for randomly generated matrices is shown in Figure 5-5. As expected, the time for both LU computation and communication is reduced rapidly for lower-precision arithmetic. On average, this approach requires 1 refinement iteration for s33e8 and 4 iterations for s16e7 format. The execution time for backward/forward solvers and iterative refinement occupies a small portion of the complete direct solver algorithm, but appear relatively long in Figure 5-5. The reasons are that the

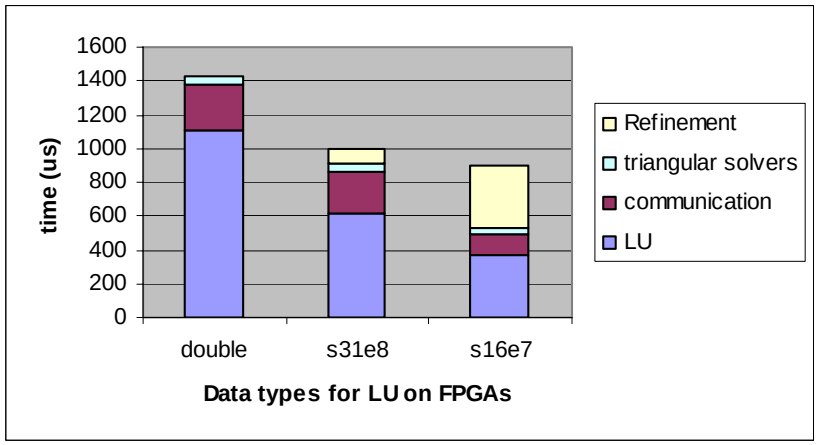

Figure 5-5: Execution Time for Mixed-Precision Direct Solvers

time for LU decomposition is significantly reduced by using our FPGA accelerator. The time on iterative refinements will become relatively small when matrix sizes increase.

Finally, we compare the performance of our design to software executing on CPUs. For software, we implement the LU decomposition algorithm in C. As shown in Figure 5-6, our double precision LU decomposer achieves 2x speedup over 2.2GHz Opteron processors. Lower-precision designs have higher performance by taking advantage of both more parallelism and higher frequency. The LU decomposer using s16e7 data format achieves about 8x speedup over software. By taking advantage of the high performance of the lower-precision LU decomposer, our mixed-precision direct solver achieves roughly 3x speedup over CPUs. The performance of the lower-precision design s16e7 is about 3 times faster for LU decomposition and 1.6 times faster for matrix solver than for the double precision design.

For large matrices, the execution time of the triangular solver and iterative refinement will require a smaller percentage in Figure 5-5. Previous work also shows that a FPGA-based LU decomposer achieves higher performance for larger matrices. For example, design 2 in [38] achieves 2GFLOPs for 100x100 matrices, but the performance increases to 4GFLOPs for 1000x1000 matrices. According to Amdahl's law [85], the high performance of our

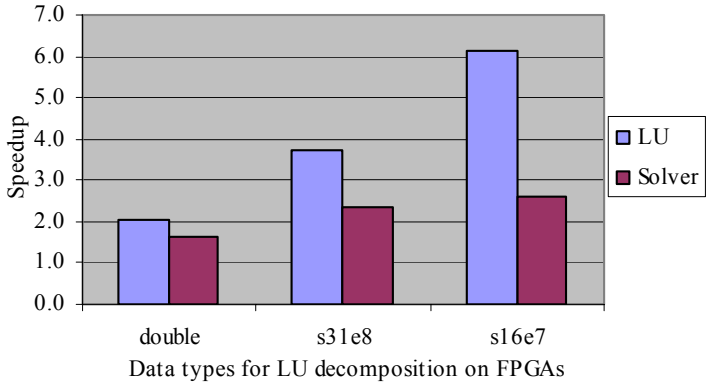

Figure 5-6: Speedup of LU and direct solver over a 2.2GHz Opteron

lower-precision LU design will make more impact on the overall performance as problem size increases. Therefore, we expect even higher speedup of our mixed-precision design for large matrices.

5.3 Conclusion

This chapter introduces the Cray XD1 architecture and the implementation of our hybrid direct solver design. Our experimental results show that the FPGA based LU decomposer design has higher performance than a 2.2 GHz Opteron processor. Due to the large size of double precision floating point units, we cannot achieve high parallelism on FPGAs for them due to resource constraints. On the other hand, our lower-precision LU decomposition design has much higher performance. Test results show that mixed-precision design on FPGAs can achieve significantly higher performance without losing accuracy.

.

6 Performance Evaluation

Due to power consumption, heat dissipation, and other reasons, it is increasingly difficult for the IC industry to keep up with Moore's Law. Therefore combining parallel clusters with FPGA application processors for high performance computing has gathered wide interest. For example, Cray supercomputers integrate computation blades by using fast interconnections. FPGA application processors can be adapted to Opteron processor based blades by adding expansion modules.

This chapter introduces FPGA application accelerators for high performance computing systems and gives performance analysis. First of all, the execution time of algorithms mapped on FPGAs is investigated. A clock cycle accurate analytic model is also introduced for the execution time on FPGAs. Due to the difficulty in developing FPGA application accelerators, an analytic model brings great convenience by enabling designers to analyze and predict the performance of FPGA applications on various platforms. Secondly, the framework of FPGA-enhanced computing system is introduced. For reconfigurable computers, the overall performance is affected by factors such as the attributes of microprocessors, FPGAs, memory, and interconnects. These factors are investigated by building a reconfigurable computing system performance model. Finally, we extend this model to parallel computing systems. Our performance model brings an important tool to optimize program development, predict the performance, and investigate high performance computer architectures.

6.1 Performance Metrics

To compare reconfigurable computing systems to traditional computers, speedup is an important metric. The basic definition of speedup is the execution time of applications on a serial processor over that of the investigated computing systems. For the heterogeneous parallel computing systems discussed in this dissertation, we define speedup as the shortest time of programs on a single microprocessor over that on a parallel computing system. If

the execution time on a single processor is R_{serial} and that on a parallel computing system is $R_{parallel}$, the speedup can be described by the following equation.

$$Speedup = \frac{R_{serial}}{R_{parallel}} \tag{6-1}$$

Heterogeneous parallel computers shorten parallel execution time by using a combination of multiple microprocessors and FPGAs. Even inside an FPGA application processor, multiple processing elements (PEs) are usually implemented. Using more parallel processing units reduces the computation load on each unit but also increases resource cost and parallel overhead. It is valuable to evaluate the speedup brought by each processing unit. Efficiency is another important metric for parallel computing systems, and is define as speedup over the number of processing units *p*.

$$Efficiency = \frac{speedup}{p} = \frac{R_{serial}}{R_{parallel} \cdot p} \tag{6-2}$$

6.2 FPGA Performance Analysis

In general, applications mapped onto hardware consist of serial and parallel parts. FPGA accelerators speed up the parallel parts of algorithms by employing parallel multiple processing units (PEs). The number of PEs is usually limited by hardware resources. Our deeply pipelined architecture also allows many FPGA applications to overcome the performance of CPUs with much higher frequency. For example, the LU factorization design in this work using 8 PEs at 120MHz has higher performance than a 2.2 GHz Opteron CPU. One important reason for FPGA application processors to achieve higher performance is that the FPGA design utilizes deeply pipelined architecture and therefore has less idle cycles. A pipeline cannot achieve peak performance unless all the pipeline stages are filled. This is the "*latency of pipelines*". For an *L* stage pipeline, the latency is also *L*. In common parallel architectures for FPGA application accelerators, the total cycles

of the critical path consists at least 3 parts: serial time c_{serial}, parallel time $c_{parallel}$, and pipeline latency $c_{latency}$. In practice, there are other overheads such as control logic cycles, and register/BRAM latencies. We include these overheads in $c_{overhead}$. Since FPGAs are usually used as accelerators for microprocessors, data and control signals have to be transferred between the host and FPGAs. The clock cycles for communication can be represented by c_{comm}. Therefore, the clock cycles of FPGAs can be represented as:

$$C_{FPGA} = c_{serial} + c_{parallel} + c_{latency} + c_{comm} + c_{overhead} \tag{6-3}$$

The central logic of a hardware design is commonly implemented as state machines. A large design usually has many states. For an application that has *S* states with deterministic length tasks, the total cycles are the summation of cycles for all the states.

$$C_{FPGA} = \sum_{i=1}^{S} (c_{serial,i} + c_{parallel,i} + c_{latency,i} + c_{comm,i} + c_{overhead,i}) \tag{6-4}$$

Now we analyse the clock cycles required by an application. Suppose the application needs $C_{task,i}$ clock cycles if parallelism is not considered. $C_{task,i}$ has parallelizable and non prrallelizable (serial) parts.

$$C_{task,i} = C_{serial,i} + C_{parallel,i} \tag{6-5}$$

For a specific application, the execution time is decided by both the total clock cycles and frequency. If *p* PEs can be implemented for the parallel tasks and the frequency is *f*, the total execution time becomes:

$$R_{FPGA} = \frac{\sum_{i=1}^{S} (C_{serial,i} + \frac{C_{parallel,i}}{p} + c_{latency,i} + c_{comm} + c_{overhead,i})}{f} \tag{6-6}$$

6.2.1 Performance Modeling for LU Factorization on FPGAs

Our FPGA-based LU decomposer accelerates the LU factorization algorithm by employing a deeply pipelined architecture and multiple parallel PEs. Chapter 5 introduces a matrix decomposer on the Cray-XD1 supercomputer which requires the complete input matrix to be fit into the FPGA on-chip memory. Some applications might require larger matrices. In this case, input matrices can be stored in the QDR memory located beside the FPGA. Considering the limited resources on current platforms and the difficulty of developing reconfigurable accelerators, it is very valuable to predict the performance of this design for bigger matrices before hardware development. A performance model for this design helps to predict performance for different inputs and optimize future hardware architecture.

Due to the dynamic sub-matrix sizes and various matrix operations in the LU factorization algorithm, the FPGA-based LU decomposer requires complicated hardware logic. Figure 3-19 gives the main stages for the LU decomposer with pivoting. These stages are divided into multiple sub-states for hardware state machines. For example, the "*column normalization*" stage in Figure 3-19 is divided into two sub-states "*column normalization*" and "*normalization delay*". The former state normalizes the column by a floating point divider, while the latter fills idle cycles and therefore avoids data hazards during the time caused by the floating point divider. Assuming the floating point divider has a latency of $L_{divider}$, the total clock cycles for the complete "*column normalization*" stage for a *k* by *k* sub-matrix is $k + L_{divider}$. If the input matrix has a size of *n* by *n*, the LU decomposition algorithm has *n-1* iterations, in which *n-1* sub-matrices with size *n* to 2 are processed. Considering overheads introduced by BRAM operations, L_{BRAM}, the total number of clock cycles for column normalization is:

$$\begin{aligned} c_{normalization} &= \sum_{k=2}^{n} (k + L_{divider} + 3L_{BRAM}) \\ &= (n^2/2 + 7n/2 - 4) + (n - 1)L_{divider} \end{aligned} \tag{6-7}$$

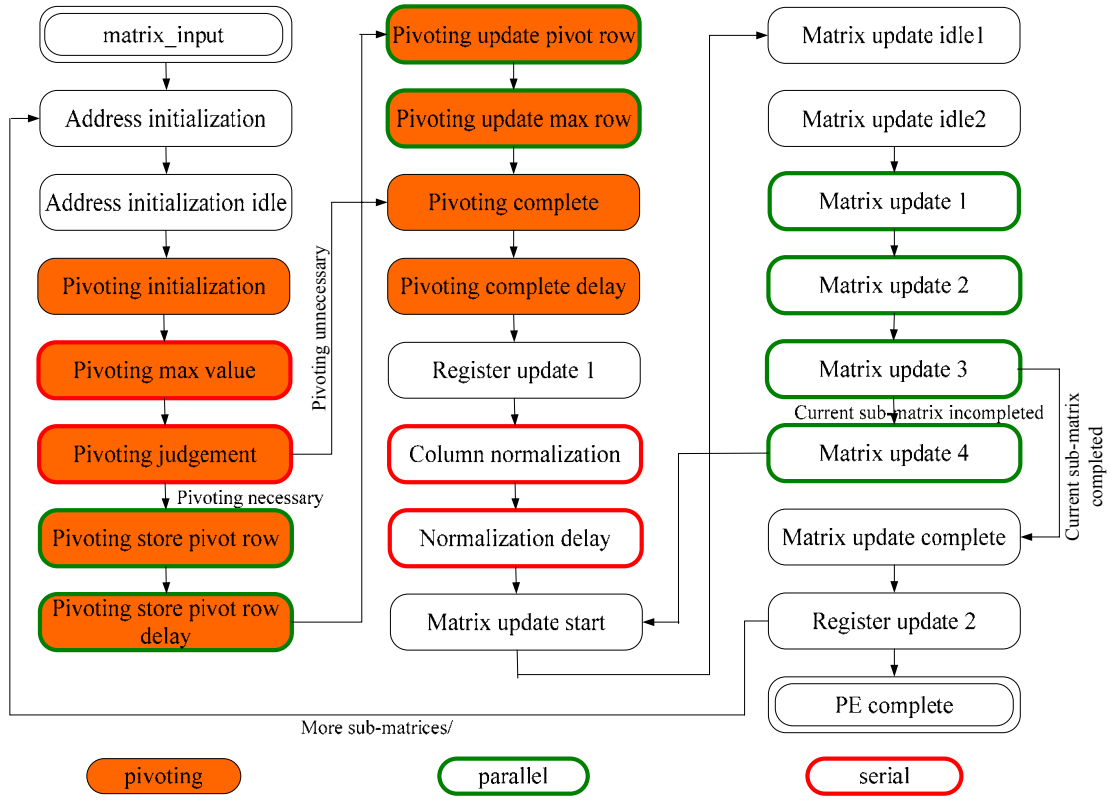

Figure 6-1: Complete States for LU Factorization

As introduced in chapter 3, the "*sub-matrix update*" stage is to update the value of $a_{x,y}$ by $a_{x,y} - l_{x,0}a_{0,y}$, where $a_{x,y}$ is the element in the sub-matrices and $l_{x,0}$ is the element in the normalized column. A sub-matrix in this design is updated by column simultaneously by all PEs. Because multiple columns are stored in a one PE's local memory as shown in Figure 3-17, the data stream in the column buffer needs to flow through the PEs multiple times with one column updated each time. For a sub-matrix with size *k*, *k* values are updated for each column. The column stream needs to circulate $\lfloor (k-1)/p \rfloor + 1$ times, where *p* is the number of PEs and $\lfloor (k-1)/p \rfloor$ is to calculate the integer part of *(k-1)/p*. For an *n* by *n* matrix, there are *n-1* sub-matrices totally with size reducing from *n* to 2. The states "*Matrix update start*", "*Matrix update idle1*", and "*Matrix update idle2*" are to initialize address registers and insert idle clock cycles for reading BRAMs. Each of these 3 states costs 1 clock cycle. Considering BRAM latencies, the total clock cycles for the sub-matrix update computation part is:

$$c_{update\ \ compute} = \sum_{k=2}^{n} (k + 3 + 3L_{BRAM})([(k-1)/p] + 1) \tag{6-8}$$

The "*Sub-matrix update*" stage has the most computations among four stages in Figure 3-19. It is divided into 8 sub-states for parallelism in Figure 6-1. As shown in Figure 3-17, the data path is deeply pipelined in the "*sub-matrix update*" stage. When a column is fed into the pipeline, the output will come out after the delay of the pipeline. The depth of this pipeline is equal to the latency of a floating point multiplier and adder, which can be represented by $L_{mult} + L_{adder}$. Our design hides this latency between iterations of columns. But the "*Matrix update complete*" state has to wait for $L_{mult} + L_{adder}$ clock cycles to avoid data hazards. There is also a 1-clock-cycle overhead due to control logic. "*Matrix update start*", "*Matrix update idle1*", "*Matrix update idle2*", and "*Matrix update complete*" need to be executed in *n-1* iterations for a matrix decomposition algorithm as shown in Figure 6-1. Therefore the clock cycles for the "*sub-matrix update*" stage totals:

$$
\begin{aligned}
c_{update} &= c_{update\ \ compute} + (n-1)(L_{mult} + L_{adder} + 1) \\
&= \sum_{k=2}^{n} (k+6)([(k-1)/p]+1) + (n-1)(L_{mult} + L_{adder} + 1)
\end{aligned} \tag{6-9}
$$

Pivoting has 9 total states. For a sub-matrix of size *k*, the "*pivoting maximum value*" state costs *k* clock cycles to find maximum value and 1 clock cycle overhead. The function of "*Pivoting store pivot row*", "*Pivoting update pivot row*", and "*Pivoting Update max row*" states is to exchange 2 rows in all PEs simultaneously by using temporary buffers and requires *3n/p* clock cycles. The other 4 states take 1 clock cycle each. Note that pivoting might not be executed depending on the results from the "*Pivoting judgment*" state. Assume the probability to execute pivoting is p_{pivot}, the total number of clock cycle for pivoting for all sub-matrices are:

$$
\begin{aligned}
c_{pivoting} &= ((3n+1)(n-1)/p)\,p_{pivot} + 3(n-1) + \sum_{k=2}^{n}(k+1) \\
&= ((3n+1)(n-1)/p)\,p_{pivot} + 3(n-1) + \sum_{k=2}^{n}(k+1)k \\
&= ((3n^2-2n-1)/p)\,p_{pivot} + (n^2/2 + 9n/2 - 5)
\end{aligned} \tag{6-10}
$$

According to Figure 6-1, there are still some states not counted. The "*Address initialization"*, "*Address initialization idle*", "*Register update 1*" and "*Register update 2*" cost 1 clock cycle per iteration. Considering BRAM read latency, the total time for LU decomposer on FPGA with frequency *f* is:

$$
T_{LU} = (c_{pivoting} + c_{normalization} + c_{update} + 4(n-1))/f \tag{6-11}
$$

Model Validation and Performance Prediction

We validate the clock cycle accurate performance model in equation (6-11) by comparing with our test results on the Cray-XD1 supercomputer. Figure 6-2 shows the execution time predicted by our performance model agrees remarkably with real test results. This performance model is very valuable to predict the performance of our design for different matrices and on different platforms.

For large matrices which cannot fit in FPGA on-chip memories, we propose to use QDR memory on the Cray-XD1 supercomputer. The Cray-XD1 supports 4 QDR memory banks.

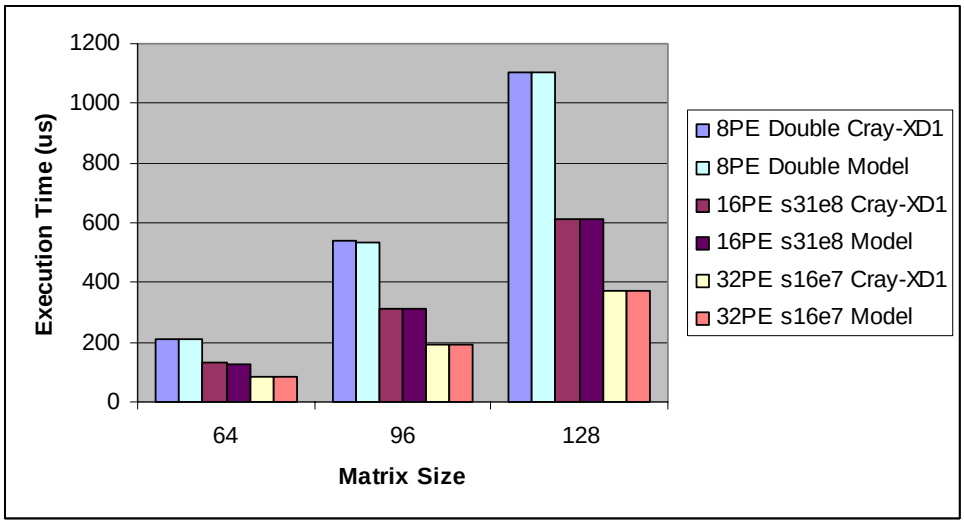

Figure 6-2: Test and Model Performance

Each QDR memory can be used as PE local memory for a single or multiple PEs. For future architectures, each PE should have a separate QDR memory bank for the purpose of high I/O bandwidth. We compare LU execution time speedup from our model with software codes on an Opteron processor of Cray-XD1 supercomputer in Figure 6-3. For software, we use C.

Linear solvers take advantage of the high performance lower-precision LU decomposition and increase the accuracy by iterative refinement. The required refinement iteration loops are listed in Table 4-2. We plot the GFLOPs performance of linear solvers in Figure 6-4. It is easy to see that mixed-precision solvers achieve higher speedups for large matrices. This is because LU decomposition dominates the execution time for large matrices.

One advantage of our LU decomposition design over previous work [36] [38] [39] [40] is that our work implements the pivoting algorithm in hardware which greatly improves the numeric properties of LU decomposition algorithms. For non-positive-definite matrices, pivoting must be implemented to prevent matrix entries from being divided by zeros. As shown in Figure 6-1, the pivoting algorithm costs almost half of the states in our design.

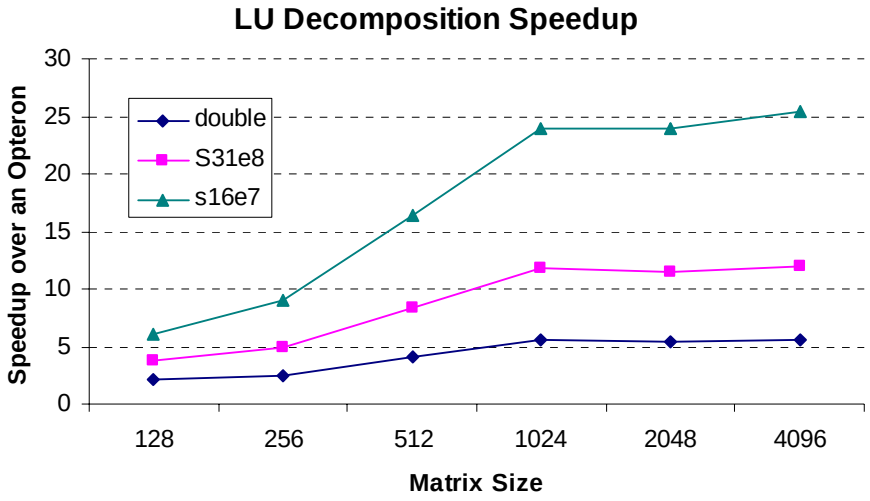

Figure 6-3: LU Performance Comparison

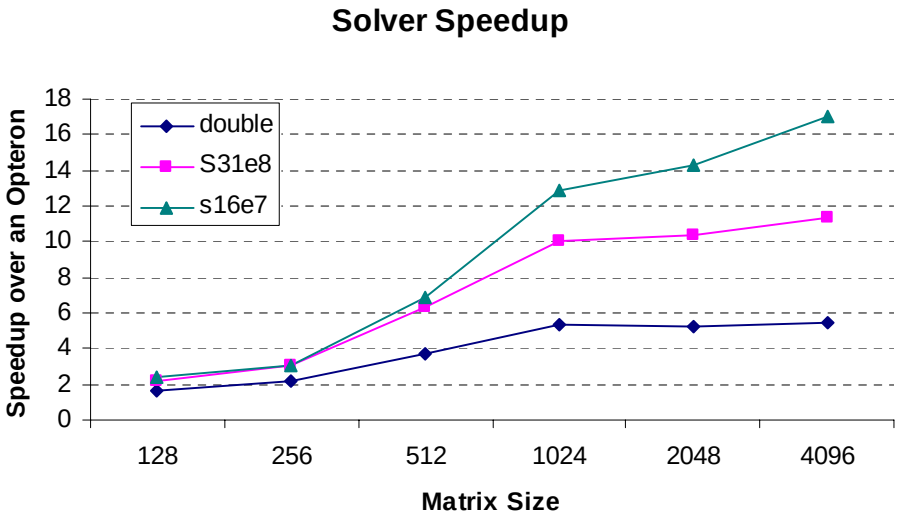

Figure 6-4: Solver Performance Comparison

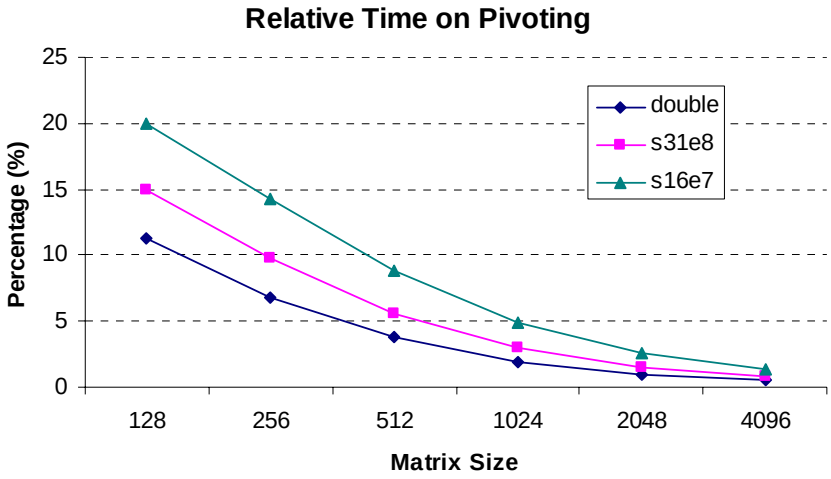

Figure 6-5: Relative Time on Pivoting

Figure 6-5 gives the relative execution time for pivoting. We observe that the percentage of time on pivoting decreases with matrix size. This is reasonable because the computational complexity of pivoting is $O(n^2)$ while that of the complete LU decomposition is $O(n^3)$. We also notice that the pivoting algorithm costs a higher percentage of time for lower-precision data formats. The reason is that lower-precision designs have more PEs, but compared to other parts of the LU decomposition algorithm, the pivoting algorithm can not take good advantage of parallelism. An accurate estimate of the relative time required for pivoting can be derived by equations (6-10) and (6-11).

6.3 Reconfigurable Single Node Model

For FPGA-enhanced computers, we start our performance analysis with a single reconfigurable computing (RC) node running a synchronous iterative algorithm (SIA). Restricting the analysis to a single node helps us to investigate the interactions between hosts and FPGA application processors before expanding to a parallel computing analysis.

As shown in Figure 6-6 (a), we assume the program segment we are interested has I similar iterations as shown in Figure 6-6 (a). A reconfigurable node could have multiple microprocessors and FPGA hardware accelerators. The program kernel to be accelerated can be parallelized and assigned to both microprocessors and FPGAs as shown in Figure 6-6 (b). Smith proposed a similar block diagram for reconfigurable nodes [76]. Because of the new multi-core technology and its wide application on supercomputers, we consider a reconfigurable node with multiple microprocessors.

For an iteration i, the time for initialization and reconfiguration can be denoted as $t_{init,i}$ and $t_{conf,i}$; the communication time is denoted as $t_{comm,i}$; the serial time cannot be accelerated is $t_{serial,i}$; the accelerated program kernel is run both on *m* microprocessors for time $t_{sw,j,i}$ $(1 \le j \le m)$ and n FPGAs for time $t_{hw,j,i}$ $(1 \le j \le n)$ respectively. We include $t_{init,i}$, $t_{conf,i}$ and other overheads in $t_{overhead,i}$. The execution time of the SIA is determined by that of the critical path, so the runtime, R_{RC}, for I iterations is:

$$R_{RC} = \sum_{i=1}^{I}(t_{serial,i} + \max(\max_{1\le j\le m}(t_{sw,j,i}), \max_{1\le j\le n}(t_{hw,j,i})) + t_{comm,i} + t_{overhead,i}) \quad (6\text{-}12)$$

Since all the iterations are similar in SIA, we are interested in a typical iteration. The parallel time on hardware and software can be described by random variables [53]. The time spent on serial execution, communication, and overheads can be represented as t_{serial}, t_{comm}, and $t_{overhead}$. Now R_{RC} becomes the expectation of Equation (6-12).

$$\begin{aligned} R_{RC} &= I(E[t_{serial}] + E[\max(\max_{1\le j\le m}(t_{sw,j}), \max_{1\le j\le n}(t_{hw,j}))] + E[t_{comm}] + E[t_{overhead}]) \\ &= I(t_{serial} + E[\max(\max_{1\le j\le m}(t_{sw,j}), \max_{1\le j\le n}(t_{hw,j}))] + t_{comm} + t_{overhead}) \end{aligned} \quad (6\text{-}13)$$

Time $t_{sw,j}$ in equation (6-13) is decided by the computational load and microprocessor computation capability. The former can be deterministic or stochastic depending on specific applications, while the latter is affected by such factors as microprocessor speed, I/O, and memory.

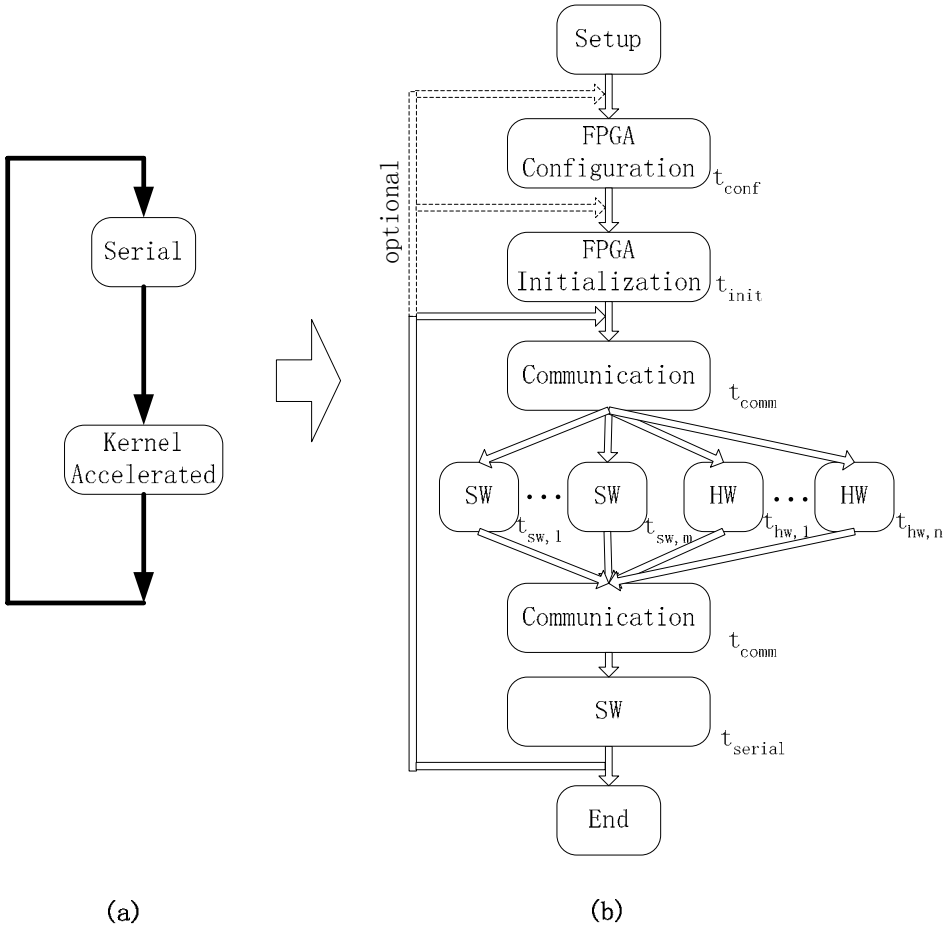

Figure 6-6: Synchronous Iterative Algorithm on a Single RC Node

In a shared computing environment, software execution time is also affected by background load which can be described by a parameter [76]. The software execution time $t_{sw,j}$ can be modeled as a random variable, whose parameters can be decided by tests on platforms, simulation, or analytical modeling [76]. Time $t_{hw,j}$ is determined by the tasks and the FPGA application accelerator performance. Because the FPGA application processor is usually a dedicated system, $t_{hw,j}$ is usually deterministic for deterministic tasks.

On a single processor, the execution time is the summation of the serial time t_{serial}, total software time $\sum_{j=1}^{m} t_{sw,j}$, and total hardware time $\sum_{j=1}^{n} t_{hw,j} \cdot \sigma_j$. Note that the execution times for the same algorithm on hardware and software are different. σ_j represents hardware speedup over software for algorithms mapped on FPGA *j*. Now the speedup of the reconfigurable computing system over a single processor is defined as the execution time on a single processor over that on the reconfigurable computing systems:

$$\begin{aligned} Speedup_{RC} &= \frac{R_1}{R_{RC}} \\ &= \frac{t_{serial} + \sum_{j=1}^{m}(t_{sw,j}) + \sum_{j=1}^{n}(t_{hw,j}) \cdot \sigma_j}{t_{serial} + E[\max(\max_{1 \le j \le m}(t_{sw,j}), \max_{1 \le j \le n}(t_{hw,j}))] + t_{comm} + t_{overhead}} \end{aligned} \tag{6-14}$$

6.4 Reconfigurable Parallel Computing Model

We now expand our analysis to parallel computers which utilize multiple nodes for high performance. The system diagram is shown in Figure 6-7. Computational tasks are divided into multiple nodes which are enhanced by FPGA application accelerators. We still consider SIA algorithms for the multiple node analysis, and assume each node has similar tasks. The total execution time is equal to the last RC node to finish its tasks plus the communication time and serial software time which cannot be divided into multiple nodes. For a system with *p* nodes, the execution time is:

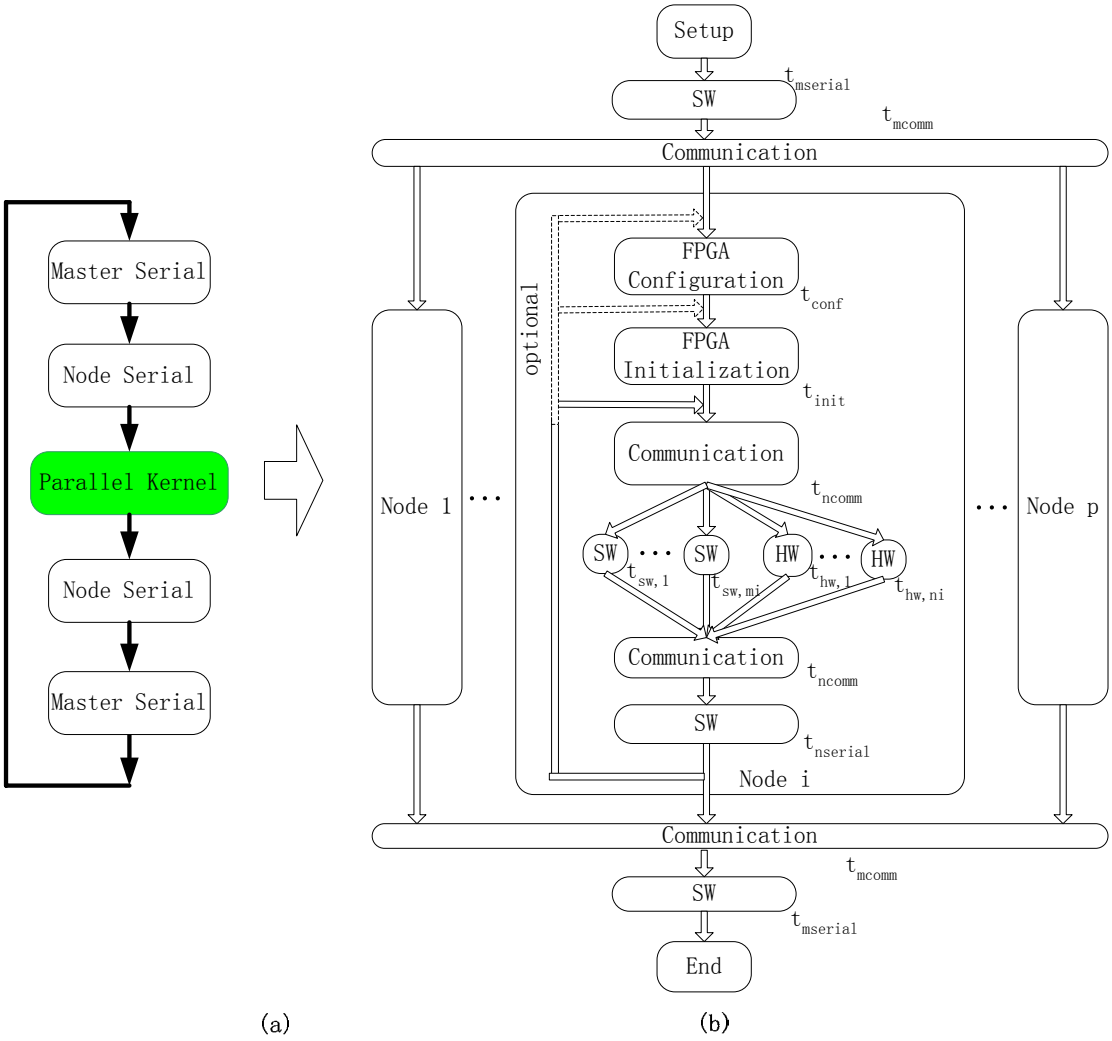

Figure 6-7: Synchronous Iterative Algorithm on Multiple RC Nodes

$$R_P = \sum_{i=1}^{I}(t_{mserial,i} + \max_{1\le j\le p}(R_{RC,j}) + t_{mcomm,i} + t_{moverhead,i}) \quad (6\text{-}15)$$

Where $t_{mserial,i}$, $t_{mcomm,i}$, and $t_{moverhead,i}$ are serial execution time, communication time, and overhead to manage all the parallel nodes. As with the single node analysis, all iterations of the SIA are similar. Therefore the serial software time, communication time, and overhead time in equation (6-15) are the same for all iterations. Now the parallel execution time becomes:

$$R_P = I(t_{mserial} + E(\max_{1\le j\le p}(R_{RC,j})) + t_{mcomm} + t_{moverhead}) \quad (6\text{-}16)$$

If we plug in the execution time model for single nodes and assume each node has m and n identical processors and FPGAs, equation (6-16) becomes

$$\begin{aligned} R_P = I(t_{mserial} + E(\max_{1\le j\le p}(t_{nserial} + E(\max(\max_{m}(t_{sw}), \max_{n}(t_{hw}))) \\ + t_{ncomm} + t_{noverhead})) + t_{mcomm} + t_{moverhead}) \end{aligned} \quad (6\text{-}17)$$

Where $t_{nserial,i}$, $t_{ncomm,i}$, and $t_{noverhead,i}$ are internal serial execution time, communication time, and overhead inside nodes. Equation (6-17) is,

$$\begin{aligned} R_P = I(t_{mserial} + t_{nserial} + E(\max_{p}(E(\max(\max_{m}(t_{sw}), \max_{n}(t_{hw}))))) \\ + t_{ncomm} + t_{noverhead} + t_{mcomm} + t_{moverhead}) \end{aligned} \quad (6\text{-}18)$$

If a program is executed on a single processor, the execution time is equal to serial execution time on microprocessors $t_{mserial}$ plus the software and hardware execution time on all nodes. The time of each node also consists of serial time, parallel time on multiple processors, and parallel time on FPGAs. Considering the hardware speedup factor σ, the execution time a single processor is:

$$R_1 = I(t_{mserial} + p(t_{nserial} + m \cdot t_{sw} + n \cdot \sigma \cdot t_{hw})) \quad (6\text{-}19)$$

The speedup is defined as the execution time on a single processor over that on a parallel system.

$$Speedup_p = \frac{R_1}{R_p} = \frac{I(t_{mserial} + p(t_{nserial} + m \cdot t_{sw} + n \cdot \sigma \cdot t_{hw})}{R_p} \quad (6\text{-}20)$$

The efficiency is defined as the speedup over the number of nodes to evaluate the contribution to performance improvement from each node.

$$Efficiency_p = \frac{R_1}{p \cdot R_p} = \frac{I(t_{mserial} + p(t_{nserial} + m \cdot t_{sw} + n \cdot \sigma \cdot t_{hw}))}{p \cdot R_p} \quad (6\text{-}21)$$

6.5 Load Imbalance Analysis

Having developed performance models for RC systems, we now look at more detailed factors affecting the model's accuracy. In previous analysis we assume dedicated systems, identical processors, and equal load distributions. To extend our models to more general cases, we try to remove these assumptions.

In a shared resource environment, processor cycles are shared by multiple programs. The computational loads caused by distributed applications are called application load, while those caused by other users or system programs are called background load [76] [52]. Imbalance of both application and background loads will cause performance degradation. For a processor j in parallel systems, Peterson and Smith used a factor $\eta_j = \gamma_j \cdot \beta_j$ to describe load imbalance. Here γ and β are both integers, and respectively represent background and application imbalance. Peterson discusses generalized models γ and β where are non intergers [88]. The parameter γ represents extra time spent by a shared resource processor over that of a dedicated processor. If time units spent by a processor j on background and application loads are l_j and 1, then the background imbalance factor γ_j is $l_j + 1$. The application imbalance parameter β_j represents the load units on

processor j. Assuming the average loads on processors are B, then β_j / B is the application imbalance scale factor for processor j. The imbalance factor can be described by random variables with a distribution function:

$$P(\eta_j = \gamma_j \beta_j = k) = \sum_{\alpha=1}^{k} P(\beta_j = \alpha) P(\gamma_j = \frac{k}{\alpha}) \tag{6-22}$$

where $P(\gamma_j = \frac{k}{\alpha}) = 0$, if $\frac{k}{\alpha}$ is not an integer [76] [52].

In heterogeneous environments, processors have different computation capabilities. If processor j requires time δ_j per unit time, and a baseline processor requires time ω for the same job, then

$$\eta_j = \gamma_j \cdot \beta_j \cdot \frac{\delta_j}{\omega} \tag{6-23}$$

Because FPGA application processors are usually dedicated systems, we do not consider background imbalance for FPGAs. For simplicity, we also assume the computational capabilities for all FPGAs are identical in this dissertation. If applications are deterministic, the execution time on FPGAs is also deterministic. In a homogeneous environment, the execution time on a parallel RC system in equation (6-24) becomes:

$$\begin{aligned} R_P = I(t_{mserial} + t_{nserial} + E(\max_{p}(E(\max(\max_{m}(\eta \cdot t_{sw}), \max_{n}(t_{hw}))))) \\ + t_{ncomm} + t_{noverhead} + t_{mcomm} + t_{moverhead}) \end{aligned} \tag{6-24}$$

6.6 Summary

In this chapter, we develop performance models for FPGA application processors, single RC nodes, and parallel RC systems running SIA algorithms. The performance models we propose effectively help to predict and optimize the performance of algorithms on new

platforms. The calculation of mean maximum such as in equation (6-24) is a difficult problem in analytic model computations. This dissertation focuses on improving the accuracy of performance models by proposing an efficient mean maximum calculation method, so other factors such as load imbalance models are just briefly introduced. We introduce the mean maximum calculation problem in the next chapter.

7 Effective Mean Maximum Approximation Method

As introduced in chapter 2, the mean maximum calculation remains as an unsolved statistics problem for years and affects the accuracy and efficiency for parallel computing models. This chapter presents an analytical method with extreme values to approximate the expectation of the maximum of random variables for both homogeneous and heterogeneous initial distributions. Compared to previous methods, it is more accurate, computationally effective, and generalizable to probability distributions. Our method provides a powerful mathematical tool to improve the accuracy and efficiency of parallel computation modeling and task graph analysis.

7.1 PERFORMANCE MODEL

Synchronous iterative algorithms are widely used in optimization, discrete-event simulation, solution to partial differential equations, Gaussian elimination and matrix inversion, finite element methods, Fast Fourier Transforms, and many others [52]. Synchronous iterative algorithms repeatedly execute a computation, with an explicit synchronization of the tasks and exchange of data performed at the end of each computation (iteration). At the end of each of the iterations, processors reach a barrier synchronization and await the arrival of the other processors before continuing. Figure 7-1 illustrates a typical synchronous iterative application.

The runtime of synchronous iterative algorithms can be described by a simplified performance model. When the algorithm has I iterations and there are P processors, the execution time R_P can be modeled as [52]:

$$R_P = \sum_{i=1}^{I} \left[t_{serial,i} + \max_{1 \le j \le P} t_{parallel,i,j} + t_{par\_overhead,i} \right] \tag{7-1}$$

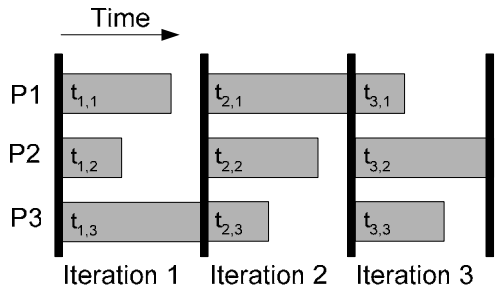

Figure 7-1: Timing of a synchronous iterative algorithm

Here $t_{serial,i}$ represents the amount of time to complete serial calculations (operations that are not or cannot be parallelized) in the i^{th} iteration. Similarly, each processor *j* completes some portion of the parallel computations for iteration *i*, requiring time $t_{parallel,i,j}$. Processors completing early sit idle waiting for the barrier synchronization operation, so $\max_{1\le j\le p}(t_{parallel,i,j})$ gives the time required for the last processor to complete iteration i. Parallel processing typically results in some additional overhead $t_{par\_overhead,i}$. Operations such as the barrier synchronization are included in this term.

We here assume all iterations require roughly the same amount of computation (the statistics for all iterations are the same). Therefore, we only need to consider the computations required for a "*typical*" iteration. The overall execution time can then be modeled as:

$$\begin{aligned} R_P &= \sum_{i=1}^{I}\left[E\left[t_{serial,i}\right] + E\left[\max_{1\le j\le P} t_{parallel,i,j}\right] + E\left[t_{par\_overhead,i}\right]\right] \\ &= I\left(t_{serial} + E\left[\max_{1\le j\le P} t_{parallel,j}\right] + t_{par\_overhead}\right) \end{aligned} \tag{7-2}$$

Where t_{serial} and $t_{par\_overhead}$ are the average time needed to complete serial and parallel overhead tasks. The mean of the maximum $E(\max_{1\le j\le p}(t_{parallel,j}))$ describes the mean

time required for the last processor to complete its parallel computations. The terms t_{serial}, $t_{par\_overhead}$, and $t_{parallel,j}$ can be found by measurement or simple calculation.

To compute the execution time in equation (7-2), we calculate the expectation for the maximum parallel execution time per iteration. Although this problem can be computed by numerical or analytical methods, an analytical solution is very helpful for performance analysis and optimization. Let s be the random variable $\max_{1\le j\le p}(t_{parallel,j})$. If the individual runtimes are identically independently distributed (i.i.d.), the distribution function of the extreme distribution s is:

$$F_S(s) = (F_{t_{parallel,j}}(s))^P \tag{7-3}$$

The density function of s is:

$$\begin{aligned} f_S(s) &= \frac{d(F_S(s))}{ds} \\ &= P(F_{t_{parallel,j}}(s))^{P-1}\ \frac{d(F_{t_{parallel,j}}(s))}{ds} \\ &= P(F_{t_{parallel,j}}(s))^{P-1}\ f_{t_{parallel,j}}(s) \end{aligned} \tag{7-4}$$

The expected maximum is then:

$$\begin{aligned} E(s) &= \int_a^b s f_S(s)ds \\ &= \int_a^b sP(F_{t_{parallel,j}}(s))^{P-1}\ f_{t_{parallel,j}}(s)ds \end{aligned} \tag{7-5}$$

Where a and b are the lower and upper bounds of random variable s. The mean maximum in equation (7-5) could be analytically solved for some simple distributions. However numerical methods often have to be used. The resulting computational load to find the mean maximum is unacceptable for many applications, such as dynamic load balancing and scheduling.

Extreme theory [66] could approximate the mean maximum when the initial random variables are i.i.d. and follow certain distributions. For normally distributed random variables with mean μ and variance σ^2:

$$E[\max_{1\le j\le P} t_{parallel,j}] \approx \mu + \sigma\left[(2\ln m)^{1/2} - \frac{\ln\ln m + \ln 4\pi}{2(2\ln m)^{1/2}} + \frac{\gamma}{(2\ln m)^{1/2}}\right] \quad (7\text{-}6)$$

Where γ is Euler's constant (0.5772).

Extreme theory gives asymptotic approximations as the number of random variables grows, but it can only work for certain distributions. To find a general and effective extreme mean maximum approximation for parallel performance evaluation, we introduce our expectation of mean maximum approximation (EMMA) method in the next section.

7.2 EMMA METHOD

7.2.1 EMMA Method for i.i.d. Random Variables

To quickly and accurately compute the mean maximum of random variables as presented before, we introduce the EMMA method for i.i.d. tasks. For simplicity, we give the conclusions without explanation first. The mathematical proofs and extensions of the method are described in the next part.

Method I: Let X_i ($1 \le i \le n$) be i.i.d. random variables, and $Y_n = \max_{i=1}^{n} X_i$. Then $P(X_i \le E(Y_n))^n \approx \varphi$, where $E(Y_n)$ is the mean of Y_n and φ is a constant taken as 0.57. If X_i has distribution function F_i with inverse function F_i^{-1}, then $E(Y_n)$ can be approximated by $F_i^{-1}(\varphi^{1/n})$.

According to this theorem, equation (7-5) can simply be calculated by

$$F_S(E(S)) = (F_{t_{parallel,j}}(E(S)))^P = 0.57 \tag{7-7}$$

$$E(S) = F^{-1}_{t_{parallel,j}}(0.57^{1/P}) \tag{7-8}$$

Compared to previous work, this theorem gives a much more effective approach for the EMV problem. By using $\varphi = 0.57$, the EMMA method replaces the complicated extreme distribution forms in order statistics. Mathematical explanation and proof will be given in the next part.

Example 1: Gaussian distribution.

Assume X_i $(1 \le i \le n)$ are i.i.d. Gaussian random variables with mean μ, variance σ^2, and $Y_n = \max_{i=1}^{n} X_i$. Here, we take $\mu = 30$ and $\sigma^2 = 9$. For each value of *n*, we use a random number generator in MATLAB to produce the n Gaussian random variables and find the maximum value. We repeat this operation 500 times and compute the expectation by taking the average of these 500 maximum values. MATLAB provides reverse distribution functions for many distributions. For the Gaussian distribution, the approximated mean maximum $E(Y_n)$ for each *n* can be simply computed as: 30+3*sqrt(2)*erfinv(2*((0.57)^(1/n))-1), where erfinv is the inverse error function for the Gaussian distribution.

In Figure 7-2, we compare the EMMA results to extreme theory [58] and MC simulations. Figure 7-2 shows that EMV from the EMMA theorem accurately agrees with the MC simulation results. We repeated the above experiment many times and change the values of σ^2, and *n*. We observed that EMMA approximates the simulation results consistently and accurately.

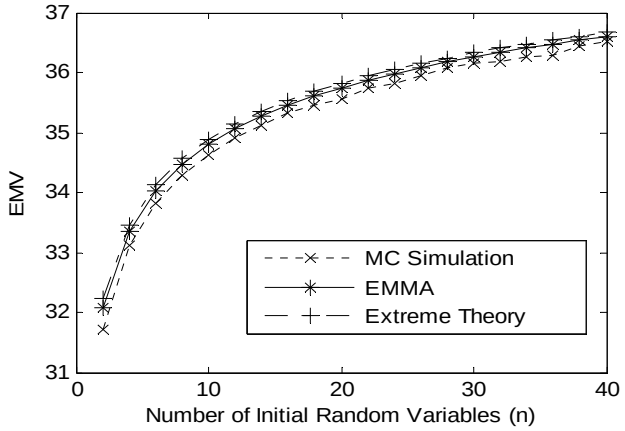

Figure 7-2: EMV by different methods for Gaussian distributions

Example 2: Binomial distribution.

Binomial is another common distribution used in computer performance modeling. For example, in parallel logic simulation, the number of gates to be simulated at a time step on each processor may follow a binomial distribution [53] [54]. Assume X_i $(1 \le i \le n)$ is binomially distributed with parameters M=5000 and p=0.02 (activity level in logic gate simulation). By using methods similar to example 1, we illustrate implementing EMMA for i.i.d. binomial distributions. The result is compared to MC simulation in Figure 7-3. The approximation accuracy increases with the number of random numbers. There are some exceptions, which is because the inverse function of the binomial distribution is discrete while the MC simulation gives continuous real numbers.

For both Gaussian and binomial distributions, the EMMA method gives results similar to MC simulation. Note that the approximation becomes more accurate when the number of random numbers grows. We compared EMMA and MC simulation for many commonly used distributions with arbitrary parameters, and observed promising results in all tests.

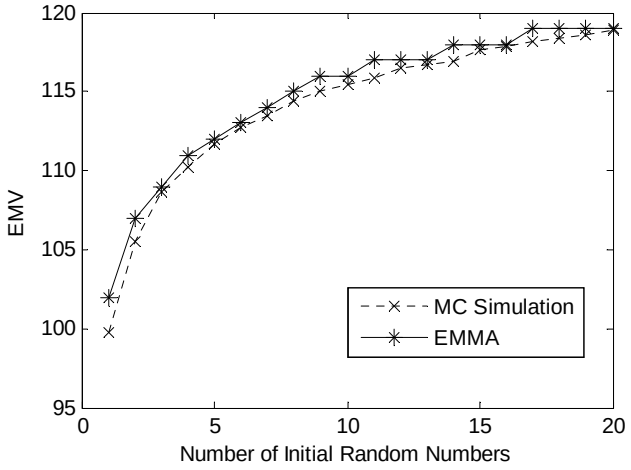

Figure 7-3: EMV by MC simulation and EMMA (Binomial distribution)

7.2.2 Mathematical Proof and Extensions

It is well known in order statistics that there are three types of distributions for extreme values: type I, type II, and type III [58]. These three types of distributions cover the asymptotic extreme distributions for most initial distributions. Most common initial distributions, such as normal, exponential, and Rayleigh distributions belong to type I. Here we explain the EMMA method by using the properties of extreme distributions.

Theorem 1: For a Type I distributions with mean μ_n and cumulative distribution function $F_{Y_n}(\cdot)$, the following property exists: $F_{Y_n}(\mu_n) \approx 0.57$.

Proof: For a Type I distribution, the probability density function (PDF) and cumulative distribution function (CDF) for the maximum values are [58]:

$$f_{Y_n}(y) = \frac{1}{k}\exp\left[-\frac{y-\alpha}{k} - \exp\left(-\frac{y-\alpha}{k}\right)\right] \tag{7-9}$$

$$F_{Y_n}(y) = \exp\left[-\exp\left(-\frac{y-\alpha}{k}\right)\right] \tag{7-10}$$

The mean of this distribution above is:

$$\mu_n = \alpha + \gamma k \tag{7-11}$$

Where γ is the Euler-Mascheroni constant. Substituting μ_n into the CDF function, we get

$$\begin{aligned} F_{Y_n}(\mu_n) &= \exp\left[-\exp\left(-\frac{(\alpha+\gamma k)-\alpha}{k}\right)\right] \\ &= \exp[-\exp(-\gamma)] \\ &\approx 0.57 \end{aligned} \tag{7-12}$$

Theorem 2: For a Type II distribution with parameters v_n and k, let μ_n and $F_{Y_n}(\cdot)$ be the mean and cumulative distribution function. The following property exists: $\lim_{k\to\infty} F_{Y_n}(\mu_n) \to 0.57$

Proof: For a Type II distribution, the probability density function (PDF) and cumulative distribution function (CDF) for the maximum random variable are [58]:

$$f_{Y_n}(y) = \frac{k}{v_n}\left(\frac{v_n}{y}\right)^{k+1} \exp\left[-\left(\frac{v_n}{y}\right)^k\right] \tag{7-13}$$

$$F_{Y_n}(y) = \exp\left[-\left(\frac{v_n}{y}\right)^k\right] \tag{7-14}$$

Where v_n is the characteristic largest value of the initial random variables and k is the shape parameter ($1/k$ is a measure of dispersion). The mean for this distribution is:

$$\mu_n = v_n \Gamma(1 - 1/k) \tag{7-15}$$

Where $\Gamma(\cdot)$ is the gamma function. Substituting μ_n into the CDF function, we get

$$\begin{aligned} \lim_{k\to\infty} F_{Y_n}(\mu_n) &= \lim_{k\to\infty} \exp\left[-\left(\frac{v_n}{v_n \Gamma(1-1/k)} \right)^k \right] \\ &= \lim_{k\to\infty} \exp\left[-\left(\frac{1}{\Gamma(1-1/k)} \right)^k \right] \\ &\to 0.57 \end{aligned} \tag{7-16}$$

In order statistics, Type I and Type II are the extreme distributions for the initial distributions unlimited in the directions of the relevant extremes. In contrast, Type III represents the limiting distribution for initial distributions with a finite upper bound or lower bound value. For execution time modeling, we are only interested in upper bounds.

Theorem 3: For Type III distribution with parameters w_n and k, let μ_n and $F_{Y_n}(\cdot)$ be the mean and cumulative distribution function. The following property exists: $\lim_{k\to\infty} F_{Y_n}(\mu_n) \to 0.57$

Proof: For Type III distribution, the PDF and CDF for the maximum random variables are [58]:

$$f_{Y_n}(y) = \frac{k}{\omega - w_n} \left(\frac{\omega - y}{\omega - w_n} \right)^{k-1} \exp\left[-\left(\frac{\omega - y}{\omega - w_n} \right)^k \right] \tag{7-17}$$

$$F_{Y_n}(y) = \exp\left[-\left(\frac{\omega - y}{\omega - w_n} \right)^k \right] \tag{7-18}$$

Where w_n is the characteristic largest value of the initial random variables, k is the shape parameter ($1/k$ is a measure of dispersion of X_n), and ω is the upper bound value of the initial distributions. The mean for this distribution is:

$$\mu_n = \omega - (\omega - w_n)\Gamma(1 + 1/k) \tag{7-19}$$

Where $\Gamma(\cdot)$ is the gamma function. By substituting μ_n into the CDF function, we get

$$\begin{aligned} \underset{k\to\infty}{F_{Y_n}}(\mu_n) &= \underset{k\to\infty}{\exp}\left[-\left(\frac{\omega - (\omega - (\omega - w_n)\Gamma(1+1/k))}{\omega - w_n}\right)^k\right] \\ &= \underset{k\to\infty}{\exp}\left[-(\Gamma(1+1/k))^k\right] \\ &\to 0.57 \end{aligned} \tag{7-20}$$

Figure 7-4 plots the CDF value at the mean point μ_n for three types of extreme distributions. It is always roughly 0.57 for Type I. With the growth of parameter *k*, the CDF values for Type II and Type III converge very quickly to 0.57 from above and below, respectively.

Based on the theorems above, we derive the following result.

Theorem 4: The CDF at the mean point for Type I is always 0.57. For Type II and III, it converges to 0.57 quickly with the shape parameter *k*.

By theorem 1 to 3, we derive the EMMA method from this theorem for i.i.d. initial distributions, whose extreme distributions meet the following sufficient conditions:

1. Type I, or

2. Type II/III with shape parameter k not too small,

For both Type II and III in extreme theory, the parameter *k* is the shape parameter, which is normally an increasing function of *n* and converges to a constant when *n* approaches

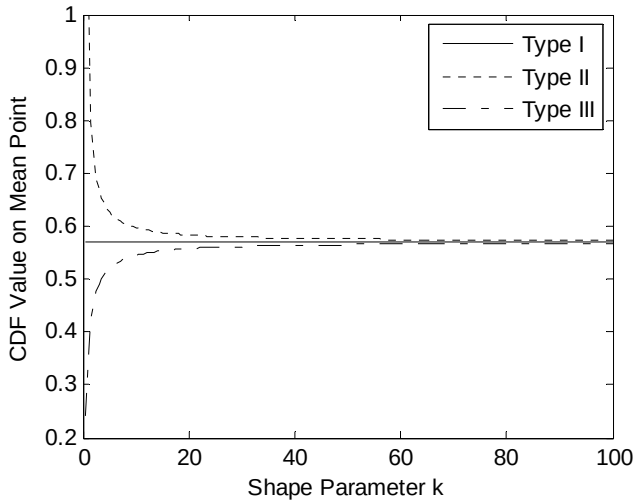

Figure 7-4: CDF values at mean maximum point for extreme distributions

infinity. Figure 7-5 gives the approximation error on some commonly used distributions. Because of a lack of analytical methods for EMV computation for most of these distributions here, we compare EMMA with MC simulation. The results are shown when the processor number is 5, 50, and 500. Note that the approximation becomes more accurate as the number of processors *n* increases.

For completeness, we now consider distributions that do not meet these two conditions. As with Figure 7-4, if for some certain initial distribution, the parameter *k* converges to a small value for a certain distribution, then a constant different than 0.57 should be used for φ to achieve more accurate approximation. However, if a certain approximation error can be tolerated, the constant 0.57 can still be used for simplicity. That is, the EMMA method is robust to parameter *k*. We describe this property by constructing a distribution converging to type III with shape parameter $k = 2$.

Example 3: Assume an initial distribution function has CDF and PDF as:

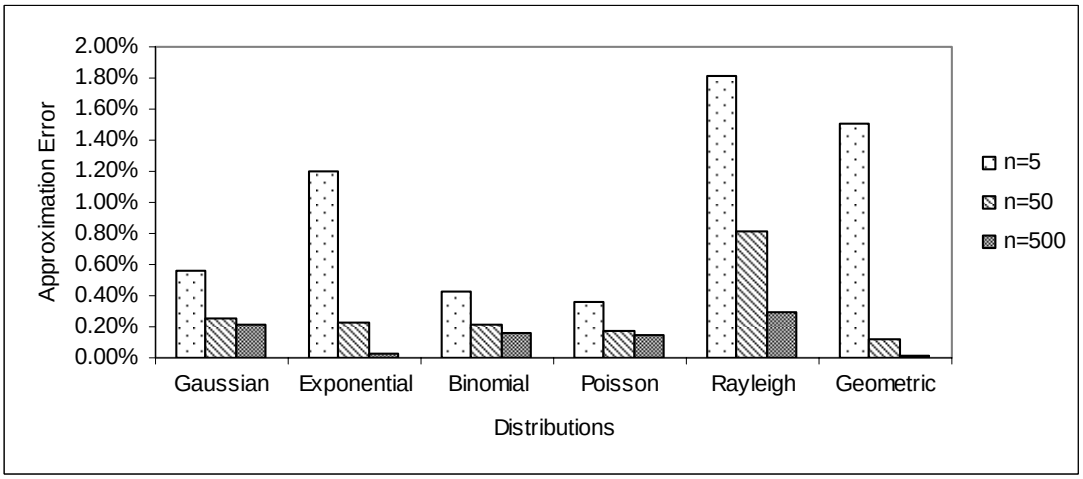

Figure 7-5: Approximation Error for Different Distributions

$$F_X(x) = 1 - ((10 - x)/10)^2 \;\; ; \;\; 0 \le x \le a \tag{7-21}$$

$$f_X(x) = (10 - x)/50 \tag{7-22}$$

This distribution is type III, the asymptotic form for the maximum value is:

$$F_{Y_n}(y) = \exp[-n((10 - y)/10)^2] \tag{7-23}$$

With the parameters $k = 2$ and $\omega = 10$. The mean is

$$\mu_{Y_n} = 10(1 - 0.5\sqrt{\pi / n}) \tag{7-24}$$

The shape parameter k is very small and the related CDF value at the mean maximum point $F_{X_n}(\mu_n)$ is around 0.46 in Figure 7-4. By extreme theory and the deduction of Theorem III, we know the EMMA method can accurately approximate the mean maximum for this distribution by taking the constant φ as 0.46. In this case, we are interested in the

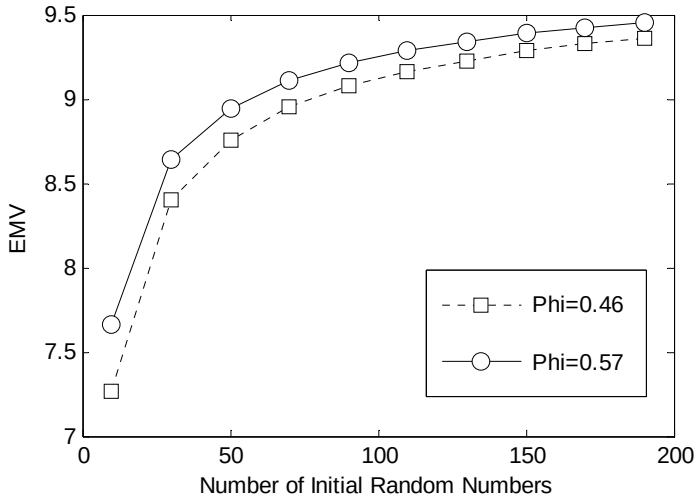

Figure 7-6: EMMA with different constants

approximation error when φ is given as 0.57. Figure 7-6 plots the approximation from EMMA method when φ is 0.46 and 0.57.

Figure 7-6 shows that for a type III distribution with small shape parameter *k*, which does not meet the sufficient conditions, the EMMA method with constant 0.57 also follows the trend very well, but with a little bigger approximation error when the number of parallel processors *n* is low.

7.2.3 EMMA Method for Heterogeneous Distribution

Method 2: Let D be a set of independent random variables that can be divided into m mutually exclusive subsets D_i $(1 \le i \le m)$. For each D_i, there are n_i i.i.d. random variables $X_{i,j}$ $(1 \le j \le n_i)$. Let $Y_n = \max(X_{i,j})$ $(1 \le i \le m$ and $1 \le j \le n_i)$ for all the probability events. Then $\prod_{i=1}^{m} P(X_{i,j} \le E(Y_n))^{n_i} \approx 0.57$, where $E(Y_n)$ is the mean of Y_n. If $X_{i,j}$ $(1 \le j \le n_i)$ has distribution function F_i, then $E(Y_n)$ can be approximated by solving the function: $\prod_{i=1}^{m} F_i(E(Y_n))^{n_i} \approx \varphi$, where φ is a constant usually taken as 0.57.

Table 7-1: Subset Parameters

Parameter	Subset 1	Subset 2	Subset 3
Mean	40	45	50
Standard Deviation	12	9	6

Table 7-2: Subset Parameters

	Subset 1	Subset 2	Subset 3
Distributions	Gaussian	Gaussian	Exponential
Parameters	$\mu = 40, \sigma = 12$	$\mu = 45, \sigma = 9$	$\mu = 30$

The above is an extension of method 1 to non-identical independent random variables. Note that different subsets do not need to have the same kind of distribution in this method. This extends EMMA for heterogeneous computing environments. Using method 2 to find EMV requires solving an implicit function where numerical methods can be used.

We illustrate method 2 using a collection of Gaussian distributions. Assume there are three subsets, each with identically distributed random variables. The parameters are shown in Table 7-1.

In Figure 7-7, we assume each subset has the same number of tasks. The X-axis is the number of tasks for each subset. We can see the EMV by Method 2 agrees with MC simulation. The EMV values for each subset are also given by MC simulation. They are all below under the overall mean maximum as expected. Because of a lack of analytical methods to calculate EMV for non-identical random variables, the largest execution time for individual subsets historically has been used as the overall execution time [57]. Figure 7-7 shows that this method can result in around ten percent error even when there are just three subsets of tasks.

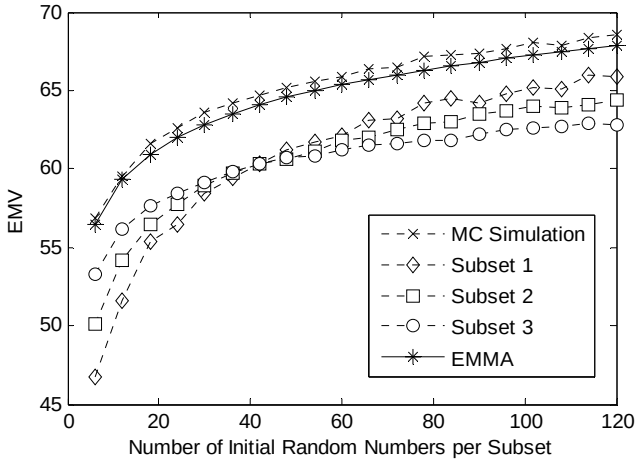

Figure 7-7: EMV from MC simulation and EMMA for heterogeneous environment (Gaussian distribution with different parameters)

We validate method 2 with various combinations of commonly used distributions and find accurate approximation results for all of them. Figure 7-8 approximates the execution time when the tasks have different distributions as shown in Table 7-2.

Note that the parameter for Subset 3 stands for mean, instead of the parameter (one over mean) normally used in the density function of an exponential distribution.

The X-axis in Figure 7-8 represents the number of processors per subset. We assume each subset has the same number of processors for simplicity. In this example, subset 3 is dominant and determines the mean maximum, which is also very accurately approximated by EMMA.

7.3 Utilization of EMMA Method

The EMMA method provides an accurate and general mathematic tool for execution time approximation in parallel computing. It is also convenient for analysis of other characteristics of the system, such as speedup and optimal processor configuration. This

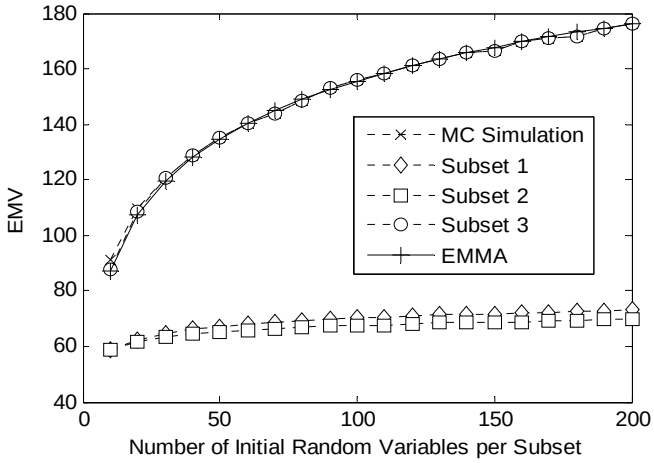

Figure 7-8: EMV from MC simulation and EMMA for heterogeneous environment (mixed distributions)

section describes using the EMMA method to analyze the system performance for some test cases.

7.3.1 Logic Simulation Applications

Logic simulation is widely used to verify modern VLSI system design before fabrication. As the number of gates per VLSI chip increases, the simulation time becomes an important issue. We now apply the model (1) and Method 1 into an example of logic simulation.

An efficient logic simulation of circuits is possible by the event-driven method, where node voltages are represented by discrete values and their changes are restricted to discrete points in time [55] [61]. The gates are modeled as functions to manipulate signals applied to their inputs and produce output signals. There is a finite delay for the gate operation depending on different gate types. On each clock cycle, plenty of the gates are inactive because their input signals remain unchanged. In the event-driven method, only the active gates are simulated. For each of the iterations, the activities of all the gates are independent and take roughly the same computational effort. Table 7-3 shows the active gates for some experimental circuits.

Table 7-3: Experimental Circuit Collections [62] [63]

Circuits	Gate count	Average activity
CKT2	1754	0.03
8080	3439	0.001-0.005
Multiplier	5000	0.01-0.02

For the event-driven method on parallel processors, tasks (gates) can be statically assigned to processors with an approximately equal amount per processor. Due to the static allocation of gates to the processors, the number of potential active gates for each processor represents a set of random variables. If we assume that the probability of each gate being active at a given time is the same and that the gates are independent, then the random variables representing the number of active gates for each processor is independent and identically distributed given each processor has the same number of gates to simulate. At the end of each of the iterations, the processors synchronize, share signal updates, and proceed to the next iteration.

We first discuss the speedup characteristics of problems with stochastic execution time. The time used for synchronization and communication are neglected for simplicity. Now, equation (7-2) is simplified as:

$$R_P = I \cdot E\left[\max_{1 \le j \le P} t_{parallel,j}\right] \tag{7-25}$$

Assume the multiplier circuit is simulated on 5 parallel processors with 1000 gates per processor. If 0.02 is picked as the average activity, the number of active gates per processor n_i ($1 \le i \le 5$) is binomially distributed with parameters 1000 and 0.02. That is, $n_i \sim B(1000,0.02)$.

Assume the computational effort for simulating each gate is one time unit and 300 iterations are needed. The expected execution time can be derived by equation (7-8), where the function F^{-1} is now the inverse function for binomial distribution:

$$R_P = 300 \cdot F^{-1}_{t_{parallel,j}}(0.57^{1/5}) \tag{7-26}$$

By using inverse Binomial distribution function, we get the expected execution time is $R_P = 7800$.

If the simulation runs on a single processor, the execution time is

$$R_1 = 300 \cdot F^{-1}_{t_{parallel,j}}(0.57^{1/1}) = 30600 \tag{7-27}$$

Note that $F^{-1}_{t_{parallel,j}}$ is now the inverse function for binomial distribution $B(5000,0.02)$. The speedup is:

$$Speedup = \frac{R_P}{R_1} = \frac{30600}{7800} = 3.92 \tag{7-28}$$

where we can see the parallel speedup cannot achieve the ideal even when the time on synchronization, communication, and overhead are not counted. The reason is for parallel computation on multiple processors:

$$E\left[\max_{1 \le j \le P} t_{parallel,j}\right] > E\left[t_{parallel,j}\right] \tag{7-29}$$

Assume this simulation task is assigned to various numbers of processors. Figure 7-9 plots the speedup with the number of processors. This example demonstrates that for problems with stochastic execution time on each processor, the speedup can never achieve the ideal due to application load imbalance.

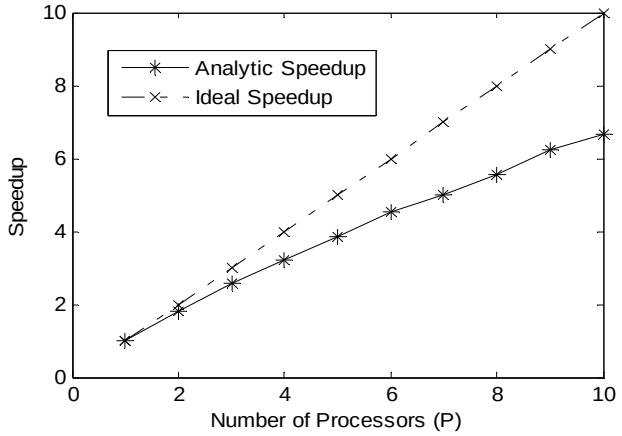

Figure 7-9: Ideal vs. analytic speedup without counting time for synchronization, communication and overheads

In practice, the synchronization and communication time cannot be neglected in many cases and can be modeled as a function of the processor number [73]. The following will introduce a method for finding the optimal processor number to achieve the minimum execution time by using the EMMA method.

For simplicity, we assume that the time for synchronization and communication is linear in the number of processor, that is, for equation (7-1):

$$t_{par\_overhead,i} = k(P-1) \tag{7-30}$$

Where k is a constant and taken as 2 in this example. Equation (7-2) becomes:

$$R_P = I\left(t_{serial} + E\left[\max_{1\le j\le P} t_{parallel,j}\right] + 2(P-1)\right) \tag{7-31}$$

After taking away the constants I and t_{serial}, which will not affect our optimization results, the cost function to minimize the execution time can be simplified as:

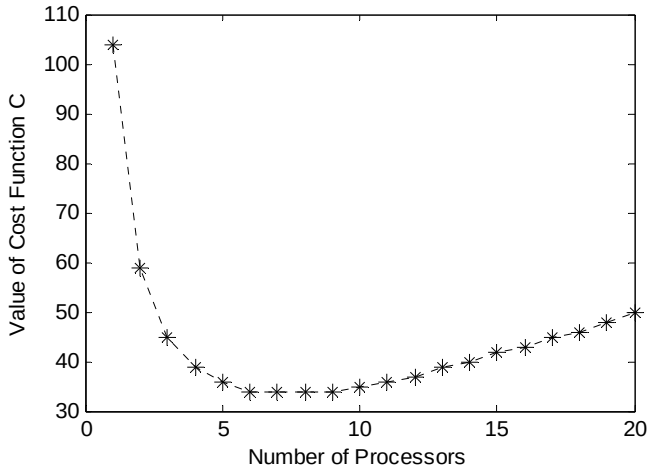

Figure 7-10: Simplified cost function for finding optimal processor number

$$C = E\left[\max_{1\le j\le P} t_{parallel,j}\right] + 2(P-1) \tag{7-32}$$

By using the EMMA method, we can easily plot this equation as shown in Figure 7-10. The optimal point is where the value of the cost function *C* has the smallest value. In this particular case, the cost function has similar value when the processor number is 6, 7, or 8. If other factors like economics are considered, 6 would be expected to be the best selection.

7.4 Execution Time for Task Graphs

A task graph is often used to describe program execution. Plenty of research addressed how to analyze the execution time of task graphs [67]. In this section, we discuss the analysis of complicated task graphs by using the EMMA method. For simplicity, some results from probability are cited without proof.

Precondition 1: Let $X_1 \cdots X_n$ be random variables and $X = \sum_{i=1}^{n} X_i$, then $E(X) = \sum_{i=1}^{n} E(X_i)$.

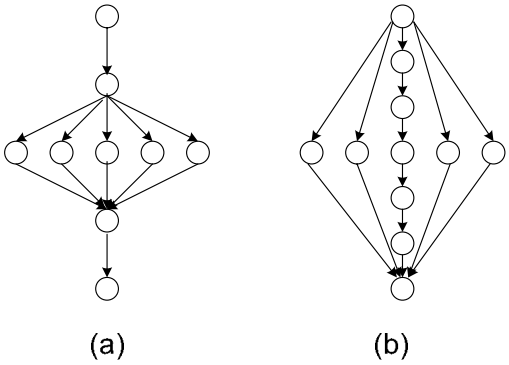

Figure 7-11: Serial and parallel task graphs

This precondition is well known in probability theory, which says that the mean of the sum is equal to the sum of the mean. For a task graph in Figure 7-11 (a), the overall structure of the task graph is serial, where each phase could be parallel tasks. In such a paragraph, the overall execution time is equal to the sum of the execution time for all phases. For phases having parallel tasks, the mean execution time of that phase can be computed by the EMMA method.

For the task graph shown in Figure 7-11 (b), the middle path consists of a series of tasks. To apply the EMMA method, we consider the overall task graph is parallel, so the distribution functions for all paths are required. We discuss finding the distribution for the sum of serial tasks in the following.

Precondition 2: Let $X_i \ (i = 1, \cdots, n)$ be a normal random variable with mean μ_i and variance σ_i^2, $X = \sum_{i=1}^{n} X_i$. Then X is still a normal random variable with mean $\mu = \sum_{i=1}^{n} \mu_i$ and variance $\sigma^2 = \sum_{i=1}^{n} \sigma_i^2$.

Since the Gaussian distribution is often used to model the running time, it is important that the distribution can be accurately calculated for the sum of Gaussian distributions. Unfortunately, there are no such nice properties for other distributions. However, the

distribution can also be approximated according to the central limit law for those non-Gaussian distributions.

Precondition 3: Let $X_i\ (i=1,\cdots,n)$ be independent and $E(X_i)=\mu_i$, $Var(X_i)=\sigma_i^2$. Assume $\sup_j(E\mid X_j\mid^{2+\varepsilon})<\infty$ for some $\varepsilon>0$. Let $X=\sum_{i=1}^{n}X_i$, then X converges to a Gaussian random variable with mean $\mu=\sum_{i=1}^{n}\mu_i$ and variance $\sigma^2=\sum_{i=1}^{n}\sigma_i^2$.

The proof of precondition 3 can be found in [58]. Once the distribution functions of all the parallel paths are available, the overall execution time in Figure 7-11 (b) can be computed by using the EMMA method for heterogeneous cases. It might not be accurate to apply the central limit law when the number of serial processes is small. A more accurate method is to compute the distribution formula for the sum of random variables. However, it is usually very complicated.

7.5 Extension to Dependent Tasks

For parallel computation performance evaluations, independence is usually assumed for simplicity. However, dependencies usually exist because of many reasons. First of all, the tasks can be dependent themselves. For example, in logic gate simulation, the active gates might be related. Secondly, for some parallel computer architectures, the parallel programs have to share some common hardware and create dependencies. Thirdly, some tasks might be dependent by sharing a common path. The communication and synchronization will also bring dependencies. It is very difficult to quantify the dependencies, so normally the dependencies are just neglected for simplicity. For example, for task graphs with common tasks, Madala approximates the execution time by assuming task path independence [67].

It is necessary to analyze the inaccuracy caused by assuming independence. In this part, we will discuss dependences among parallel tasks. We use the timing model in Figure 7-1.

7.5.1 Associated Tasks

For parallel programs with dependences, the EMMA method can be applied by neglecting the dependencies. The following part will discuss the result in this case for associated parallel tasks. Associated tasks imply that increasing the load of one task will impact that of the others. An example is that the number of active logic gates increases simultaneously on different parts of a circuit. A precise definition for association is the following [70]:

Definition 1: Random variables $X_1, \cdots, X_n$ are associated if

$$\mathrm{cov}[\Gamma(X), \Delta(X)] \geq 0 \tag{7-33}$$

For all pairs of increasing binary functions Γ and Δ.

It is well known in the theory of reliability that if X_i $(1 \leq i \leq n)$ are associated random variables, then [70] [71]

$$P[X_1 \leq y, \cdots, X_n \leq y] \geq \prod_{i=1}^{n} P[X_i \leq y] \tag{7-34}$$

Let $Y_n = \max_{i=1}^{n} X_i$, then

$$F_{Y_n}(y) \geq \prod_{i=1}^{n} F_{X_i}[y] \tag{7-35}$$

Corollary 1: For dependent associated parallel tasks, the result from the EMMA theory by ignoring dependence is an upper bound of the real mean of the maximum.

Proof: according to theorem 4, the mean of the maximum can be computed by:

$$F_{Y_n}(E(Y_n)) = 0.57 \tag{7-36}$$

From equation (7-35), we have

$$\prod_{i=1}^{n} F_{X_i}\left[E(Y_n)\right] \leq 0.57 \tag{7-37}$$

If we compute the mean of the maximum by ignoring the dependence, we consider both sides in equation (7-37) as equal. That is, we compute by using

$$\prod_{i=1}^{n} F_{X_i}\left[(Y_n)\right] = 0.57 \tag{7-38}$$

Since the function sum and cumulative distribution function are both non-decreasing, the computed results are bigger than or equal to the actual values. The equality is achieved when the random variables are mutually independent.

7.5.2 Sharing Common Paths

The dependence addressed in corollary 1 could also be caused by sharing a common path. Note that although the dependence caused by sharing a common path meets the definition of association, corollary 1 cannot be applied because of synchronization effects.

For example, assume the sub tasks in Figure 7-12 represented by cycles are identical. The running time of each subtask is Gaussian distributed with mean 30 and variance 9. By applying the EMMA method, the mean execution time for phase 1, 3, and 5 in task graph (a) is computed to be 34.033. Therefore the overall average runtime can be calculated as 34.033+30+34.033+30+34.033=162.099. For task graph (b), the running time distribution for each path is Gaussian distributed with mean 150 and variance 45, so the overall average execution time is 159.227. MC simulation results also agree to the results from EMMA. This example shows that if we compute the execution time of task graph (a) as 5 independent paths as in task graph (b), the result is less than the actual value. This does not conflict with corollary 1. The reason is that task graph (a) cannot be simply considered as a dependent counterpart of the task in graph (b), because phases 1, 3, and 5 need to be synchronized before proceeding to the next phase and this synchronization costs extra time. Some previous work argued the execution time in task graph (b) is the upper bound of its

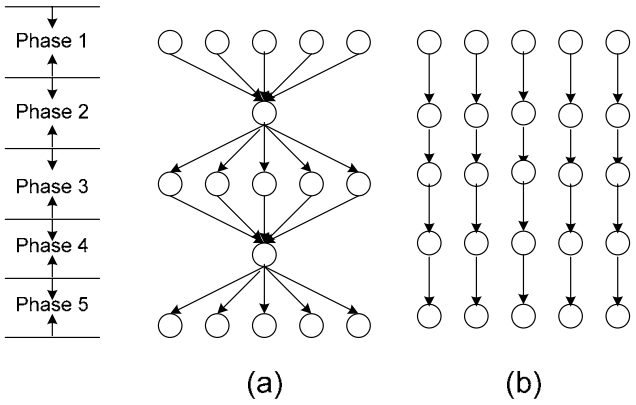

Figure 7-12: Task graph with common paths (a) and its independent counterpart (b)

counterpart with common paths (a) [71]. Here is an opposite example that disproves this claim. Hence, by applying the EMMA technique, we can account for these effects.

7.6 CONCLUSIONS

Accurate performance modeling of parallel applications faces difficulties due to the challenge of finding EMV. Despite significant previous work, the problem is still unsolved for decades, especially for heterogeneous computing. Our work can be considered as an extension of Extreme Theory, especially to heterogeneous distributions. By exploiting extreme value properties, we propose the EMMA method that is capable of finding fast, accurate approximations for parallel execution time in both homogeneous and heterogeneous environments. We present a mathematical proof and comparisons to MC simulation which demonstrate the accuracy and generality of our method. EMMA can significantly improve the accuracy and efficiency of parallel computation modeling.

Methodologies are also proposed to simplify the task graph analysis. We extend EMMA to interdependent tasks and evaluate the effect of dependencies. Further work could focus on applying the EMMA method onto different performance model and task graphs.

8 Conclusions and Future Work

While traditional CPUs struggle to keep up with Moore's law, new heterogeneous computing systems show potential for high performance scientific computing. This dissertation explores high performance reconfigurable computer architectures for linear algebra applications. First of all, we develop application-specific processors for linear algebra, which can be implemented on supercomputers as accelerators. Various linear algebra algorithms and architectures are discussed for high performance matrix computation on FPGAs. Secondly, execution time models are developed for both FPGA accelerators and reconfigurable computing systems to better understand the performance our systems. Finally this dissertation proposes an important statistics theory, which greatly increases the accuracy and convenience of parallel computing system performance modeling.

8.1 Conclusions

We have developed application-specific processors for high performance linear algebra on FPGAs. The linear algebra subroutines we explore in this dissertation include sparse matrix multiplication and a dense matrix direct solver. To achieve high performance matrix computations, various matrix algorithms and hardware architectures circuits are proposed.

Our sparse matrix multiplication solver can achieve 20 times speedup over contemporary CPUs and the performance depends less on matrix structure. On traditional CPUs, sparse matrix operations are normally inefficient because of frequent data movements. Our architecture achieves high performance by taking advantage of several factors. First, we propose an innovative sparse matrix storage format to reduce data movement overhead. Second, high performance data and control paths are utilized such as multiplication accumulator, adder tree, and summation circuits. Third, our streaming architecture greatly reduces idle cycles in data path pipelines.

Our direct solver on FPGAs achieves significant speedup over CPUs by using a hybrid architecture. Since LU decomposition is the dominant part of direct solvers, it is mapped onto FPGAs for fast computation. On the other hand, triangular solvers are implemented on CPUs to save resources and development time because these computations are a mush smaller fraction of serial execution time. The overall performance of our hybrid direct solvers is improved by an innovative LU decomposition circuitry on FPGAs, which computes LU decomposition on several parallel PEs. Our LU decomposition design is also the first work to include the pivoting algorithm for a high performance design implemented with a hardware description language.

Due to the high cost of double-precision floating point units, we propose to use a mixed-precision algorithm and architecture for high performance linear algebra. In our architecture, lower-precision floating point is used as much as possible for higher performance, while higher-precision floating point is utilized only when necessary. For linear direct solvers, a mixed-precision refinement algorithm is employed to achieve high accuracy for final results. Theoretical analysis and experimental results show that our mixed-precision direct solver successfully takes advantage of the higher performance of lower-precision floating point units without loss of accuracy.

We target our linear solvers on Cray-XD1 supercomputers for performance analysis. Cray-XD1 supercomputers consist of many computation nodes connected by a high speed interconnect. FPGAs can be connected to computation nodes by Hyper Transport as application-specific processors. Our implementation on the Cray-XD1 supercomputer includes the development for both FPGAs and the host programs. Our test results show that the performance of linear algebra can be greatly improved on Cray-XD1 supercomputers by using FPGAs.

Execution time models are important for understanding the system performance, mapping tasks, and optimizing architecture. First of all, we build performance models for application-specific processors on FPGAs by dividing computation time into several categories: parallel time, serial time, communication time, and overheads. Because the

circuit activities can be clock cycle accurately predicted, our FPGA performance model has very high accuracy. By analyzing performance models with different parameters, we are able to accurately estimate the performance of our design on different architectures.

We further build performance models for single FPGA-enhanced computation nodes and parallel computing systems, where the overall execution time is determined by both FPGAs and the host processors. The execution time on host processors is affected by application load imbalance. At the same time, background loads also increase execution time in shared resource computational environments. We evaluate these factors by using different parameters.

One difficulty in parallel performance modeling is to compute the expectation of the maximum of a set of random variables. Previous methods including extreme theory and other estimation methods are either not general or accurate enough. We propose an efficient mean maximum approximation (EMMA) method which accurately approximates the mean maximum by using very simple formulas. The EMMA method also provides an important tool for complicated task graph analysis.

8.2 Future Work

Our work shows the potential of using FPGAs for high performance linear algebra and provides performance analysis tools. Future work includes exploration of more hardware architectures, enhancing performance models, and finding applications of the EMMA theory.

First of all, we show the potential of high performance linear algebra on FPGAs by developing sparse matrix vector multiplication and dense direct solvers. Similar linear algebra computation kernels can be developed for a host of other applications. For example, by taking advantage of our sparse matrix vector multiplication circuits, it should be easy to develop iterative solvers on FPGAs. Because of the limited size of our FPGA chips, the triangular solvers are implemented on processors in our work. When larger FPGAs are

used in the future, triangular solvers can be merged onto FPGAs to increase the performance of our direct solvers.

Mixed-precision algorithms and architectures are very interesting. We point out the potential of using mixed-precision architectures on FPGAs for high performance. Future work includes developing mixed-precision algorithms and architectures for other applications and platforms. This dissertation mainly explores the execution time of mixed-precision architectures. Since mixed-precision designs require less resources and are faster, we also expect better power efficiency. Further study of such power-related issues remains to be investigated.

Our performance models focus on computation. Other factors such as I/O performance can be included in our model if applications require. Heterogeneous systems show great potential for high performance computing. Based on our performance models for a reconfigurable computer, heterogeneous computing system performance models can also be easily extended.

Finally, the EMMA theory provides an important tool for mean maximum calculations. This dissertation successfully derives mathematical forms and proofs for the EMMA theory. Future work should focus on applying the EMMA theory to various applications such as task graph analysis and schedule optimization.

List of References

List of References

[1] J. Dongarra, "Basic Linear Algebra Subprograms Technical Forum Standard", *International Journal of High Performance Applications and Supercomputing, 16(1) (2002), pp. 1--111, and International Journal of High Performance Applications and Supercomputing, 16(2) (2002)*, pp. 115--199.

[2] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[3] O. O. Storaasli. "Performance of NASA Equation Solvers on Computational Mechanics Applications," *37-th SDM Conf.*, AIAA-96-1505, Vol. 3, pp. 1213–1223.

[4] A. Pinar and M. T. Heath. "Improving Performance of Sparse Matrix-Vector Multiplication," *Supercomputing*, November 1999.

[5] E.-J. Im, K. A. Yelick. "Optimizing Sparse Matrix Computations for Register Reuse in Sparsity". *International Conference on Computational Science*, 2001.

[6] K. D. Underwood. "FPGAs vs. CPUs: Trends in peak floating-point performance," *ACM International Symposium on Field Programmable Gate Arrays*, February 2004.

[7] Y. Bi, G. D. Peterson, L. Warren, and R. Harrison, "Hardware Acceleration of Parallel Lagged-Fibonacci Pseudo Random Number Generation," *ERSA*, June 2006.

[8] R. Scrofano and V. K. Prasanna. "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware," *ERSA*, June 2004.

[9] J. L. Tripp, A.A. Hanson, M. Gokhale, and H.S. Mortveit. "Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study," *Supercomputing*, November 2005.

[10] K. Underwood, S. Hemmert, and C. Ulmer. "Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications," *Supercomputing*, November 2006.

[11] L. Zhuo and V. K. Prasanna. "Sparse matrix-vector multiplication on FPGAs," *FPGA*, February, 2005.

[12] M. deLorimier and A. DeHon. "Floating-Point Sparse Matrix-Vector Multiply for FPGAs," *FPGA*, February, 2005.

[13] Y. El-kurdi, W. J. Gross, and D. Giannacopoulos. "Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs," *FCCM*, April 2006.

[14] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems," *International Journal of High Performance Computer Applications*, Volume 21 Number 4, Winter 2007, pp 457-466, ISSN 1094-3420.

[15] R. Barrett, *Templates for the solution of Linear Systems: Building Blocks for Iterative methods,* 2nd Edition. SLAM, Philadelphia, PA, 1994.

[16] Cray Inc. *http://www.cray.com*

[17] Digilent Inc. *http://www.digilentinc.com*

[18] Xilinx Inc. *http://www.xilinx.com*

[19] H. A. ElGindy and Y. L. Shue. "On Sparse Matrix-Vector Multiplication with FPGA-based System," *10th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[20] R. Vuduc, J. Demmel, K. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels," *SciDAC 2005, Journal of Physics: Conference Series*, June 2005.

[21] T. Davis, University of Florida Sparse Matrix Collection, *http://www.cise.ufl.edu/research/sparse/matrices*, NA Digest, 92(42), October 16, 1994, NA Digest, 96(28), July 23, 1996, and NA Digest, 97(23), June 7, 1997.

[22] O. O. Storaasli, "Compute Faster without CPUs: Engineering Applications on NASA's FPGA-based Hypercomputers," *Technical Symposium on Reconfigurable Computing with FPGAs*, Manchester UK, February 2005.

[23] J. Sobieski and O.O. Storaasli, "Computing at the Speed of Thought," *Aerospace America*, Oct. 2004, pp. 35-38.

[24] O. O. Storaasli, "Engineering Applications on NASA's FPGA-based Hypercomputer," *MAPLD*, September, 2004.

[25] O. O. Storaasli, "Computing Faster without CPUs: Scientific Applications on a Reconfigurable, FPGA-based Hypercomputer," *MAPLD* Conference, September, 2003.

[26] O. O. Storaasli, R. C. Singleterry, and S. Brown, "Scientific Computations on a NASA Reconfigurable Hypercomputer," *MAPLD*. September, 2002.

[27] J. Sun, G. Peterson, O.O. Storaasli, "Sparse Matrix-vector Multiplication Design on FPGAs", *the 15th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, CA, April, 2007.

[28] Nallatech Inc. *http://www.nallatech.com*

[29] Sgi Inc. *www.sgi.com*

[30] Starbridge Inc. *www.starbridgesystems.com*

[31] SRC Computers Inc. *www.srccomp.com*

[32] G. Govindu, R. Scrofano and V. K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," *The 2005 International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2005.

[33] X. Wang, M. Leeser, and H. Yu, "A parameterized floating-point library applied to Multispectral image clustering," *MAPLD*, September 2004.

[34] Y. Dou, S. Vassiliadis, G. Kuzmanov, G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," *FPGA*, 2005.

[35] Z.H. Kamal, A. Gupta, L. Lilien, and A. Khokhar, "Classification using Efficient LU Decomposition in Sensornets," *wireless sensor network*, 2006.

[36] V. Daga, G. Govindu, S. Gangadharpalli, V. Sridhar, and V. K. Prasanna, "Efficient Floating-point Based Block LU Decomposition on FPGAs," *ERSA*, June 2004.

[37] Gokul Govindu, Seonil Choi, Viktor K. Prasanna, Vikash Daga, Sridhar Gangadharpalli, and V. Sridhar, "A High-Performance and Energy-efficient Architecture for Floating-point based LU Decomposition on FPGAs," *RAW*, April 2004.

[38] Ling Zhuo, Viktor K. Prasanna, "High-Performance and Parameterized Matrix Factorization on FPGAs," *FPL*, Madrid, Spain, August 2006.

[39] X. Wang and S.G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," *Concurrency Computation: Prac. Expei.*, Vol. 16, No. 4, April 2004.

[40] DaeGon Kim, Sanjay V. Rajopadhye, "An Improved Systolic Architecture for LU Decomposition," IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2006.

[41] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy," *Accepted in ACM TOMS*, to appear in December 2008 issue.

[42] J. W. Demmel, "Applied Numerical Linear Algebra," *SIAM Press*, 1997.

[43] R. Strzodka and D. Göddeke, "Mixed precision methods for convergent iterative schemes," *EDGE*, North Carolina, May 2006.

[44] R. Strzodka and D. Göddeke, "Pipelined mixed precision algorithm on FPGAs for fast and accurate PDE solvers from low precision componets," *IEEE Proceedings on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, may 2006.

[45] D. Göddeke, R. Strzodka, and S. Turek, "Accelerating Double Precision FEM simulations with GPUs," *ASIM*, Sep 2005.

[46] J. H. Wilkinson, "The Algebraic Eigenvalue Problem," Oxford, U.K.: Clarendon, 1965.

[47] C. B. Moler, "Iterative Refinement in Floating Point," *J. ACM (2)* (1967) 316–321.

[48] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy, "Error Bounds from Extra Precise Iterative Refinement," *Technical Report No. UCB/CSD-04-1344, LAPACK Working Note 165*, August 2004.

[49] G. W. Stewart, *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[50] Cray, Inc., "Cray XD1 FPGA Development," 2005.

[51] Cray, Inc., "Design of Cray XD1™ RapidArray Transport Core", 2005.

[52] G. D. Peterson and R. D. Chamberlain, "Parallel Application Performance in a Shared Resource Environment," *IEE Distributed Systems Engineering Journal*, August 1996.

[53] G. D. Peterson and R. D. Chamberlain, "Beyond Execution Time: Expanding the Use of Performance Models," *IEEE Parallel & Distributed Technology,* 2(2): 37-49, 11994.

[54] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Volume I, Prentice Hall, 1988.

[55] M. L. Bailey, Jr. J. V. Briner, and R. D. Chamberlain. “Parallel Logic Simulation of VLSI Systems,” *ACM Computing Surveys*, 26(3): 255--294, September 1994.

[56] R. D. Chamberlain, “Parallel logic simulation of VLSI systems,” Proceedings of the 32nd *ACM/IEEE conference on Design automation conference*, p.139-143, June 12-16, 1995, San Francisco, California, United States.

[57] V. D. Agrawal and S. T. Chakraadhar, “Performance analysis of synchronized iterative algorithm on multiprocessor systems,” *IEEE Trans. Parallel and Distributed Systems*, VOL. 3, NO. 6, 739-745, Nov. 1992.

[58] J. Jacod and P. Protter, *Probability Essentials*. *Springer*, 2000.

[59] R. -D. Reiss, *Approximate Distributions of Order Statistics with Applications to Nonparametric Statistics*. *Springer*, 1989.

[60] S. S. Gupta, "Selection and ranking procedures and order statistics for the binomial distribution," *in Classical and Contagious Discrete Distributions, G. P. Patil, Ed. Calcutta: Statistical Publishing Society*, 1965, pp. 219-230.

[61] K.T. Cheng and V. D. Agrawal, Unified Methods for VLSI Simulation and Test Generation. *Boston: Kluwer Academic*, 1989.

[62] P. Agrawal, “Concurrency and communication in hardware simulators,” *IEEE Trans. on Computer Aided Design*, vol. CAD-5, pp. 617-623, Oct. 1986.

[63] L. Soule and T. Blank, “Statistics for parallelism and abstraction level in digital simulation,” in Proc. *24th ACM/IEEE Design Automat*. Conf., 1987, pp. 588-591.

[64] G. D. Peterson and R. D. Chamberlain, “Sharing Networked Workstations: A Performance Model,” *6th IEEE Symposium on Parallel and Distributed Processing*, pp. 308-315, Dallas, TX, October 1994.

[65] B. W. Weide, “Analytic Models to Explain the Anomalous Behavior of Parallel Programs,” *In International Conference on Parallel Processing*, pp. 183-187, 1981.

[66] A.H. –S. Ang and W. H. Tang, *Probability Concepts in Engineering Planning and Design Vol. II. Rainbow Bridge*, 1984.

[67] S. Madala and J. B. Sinclair, “Performance of Synchronous Parallel Algorithms with Regular Structures,” *IEEE Transactions on Parallel and Distributed Systems*, 2(1): 105-116, January 1991.

[68] J. T. Robinson, “Some analysis techniques for asynchronous multiprocessor algorithms,” *IEEE Trans. Software Eng*., vol. SE-5, pp. 24-31, Jan. 1979.

[69] H. A. David, *Order Statistics*. *New York: Wiley*, 1981.

[70] R. E. Barlow and F. Proschan, "Statistical Theory of Reliability and Life Testing," New York: hold 1975.

[71] Nihal Yazici-Pekergin and Jean-Marc Vincent, "Stochastic Bounds on Execution Times of Parallel Programs," *IEEE Trans. on Software Eng., vol. 17, No. 10*, Oct. 1991.

[72] J. Sun and G. D. Peterson, "Effective Execution Time Estimation of Heterogeneous Parallel Computing," *PDCS*, Sep, 2006.

[73] M. A. Driscoll and W. R. Daasch, "Accurate Predictions of Parallel Program Execution Time," *Journal of Parallel and Distributed Computing. Vol. 25, No. 1*, Feb, 1995.

[74] Hu, L. and Gorton, I., "Performance Evaluation for Parallel Systems: A Survey," *UNSWCSE-TR-9707, pp. -56*, Sydney, Australia, 1997.

[75] Kant, K., "Introduction to Computer System Performance Evaluation New York," *McGraw-Hill, Inc.*, 1992.

[76] Melissa C. Smith and Gregory D. Peterson, "Parallel Application Performance on Shared High Performance Reconfigurable Computing Resources," *Performance Evaluation*, 60(1-4): 107-125, 2005.

[77] J. Sun, G. Peterson, O.O. Storaasli, "Mapping Sparse Matrix-Vector Multiplication on FPGAs," *RSSI*, 2007.

[78] Mitrion inc. http://www.mitrion.com/

[79] Xtremedata inc. http://www.xtremedatainc.com/

[80] DRC Computer inc. http://www.drccomputer.com/

[81] G. H. Golub, C. F. Loan, *Matrix Computations*. 3rd edition, Johns Hopkins, 1996.

[82] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. 2nd Edition, SIAM Press, 2002.

[83] H. Bowdler, R. Martin, G. Peters, and J. Wilkinson. "Handbook series linear algebra: Solution of real and complex systems of linear equations," *Numerische Mathematic*, 8: 217-234, 1966.

[84] J. Demmel, M. Heath, and H. van der Vorst, "Parallel numerical linear algebra," in *Acta Numerica*, p111-198, Cambridge University Press, Cambridge, UK, 1993.

[85] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *In AFIPS Conference Proceedings*, pp. 483-485, 1967, Reston, VA.

[86] J. Sun, "Obtaining High Performance via Lower-Precision FPGA Floating Point Units," *Supercomputing*, Reno NV, Nov. 2007.

[87] K. Turkington, K. Masselos, G. A. Constantinides, P. Leong, "FPGA Based Acceleration of the Linpack Benchmark: A High Level Code Transformation Approach," *FPL*, Aug. 2006.

[88] G. D. Peterson, "Parallel Application Performance on Shared, Heterogeneous Workstations," Ph.D. dissertation, Washington University, Missouri, 1994.

Vita

Junqing Sun was born in Jiangsu, China. He received his B.S. and M.S. degrees from Tongji University, China in 2001 and 2004. He was ranked the top out of 70 students in his class for his undergraduate and granted “Best 100 Students of Tongji University”. He was certified as an “Advanced Programmer” by the Chinese Ministry of Information Industry.

Junqing Sun completed his Ph.D. with a cumulative GPA of 4.0 from The University of Tennessee, Knoxville in 2007. He won the first place award for the ACM Student Research Competition at Supercomputing 2007. He was also selected as a member of American Academic Honor Society Phi Kappa Phi. He filed one patent application during his graduate school.