Performance Analysis of Hardware/Software Co-Design of Matrix Solvers

A Thesis Presented to the
College of Graduate Studies and Research
In Fulfillment of the Requirement
For the Degree of Master of Science
In the Department of
Electrical and Computer Engineering
University of Saskatchewan
Saskatoon, Saskatchewan
Canada

By

Peng Huang

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Electrical and Computer Engineering
57 Campus Drive
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude and appreciation to my supervisor, Dr. Daniel Teng, for his tremendous support, invaluable guidance and constant encouragement during the course of my studies. The completion of this thesis would not have been possible without Dr. Teng's exceptional supervision and ever lasting support. I am also grateful to him for providing me with various opportunities to pursue a dynamic and fascinating area of digital systems as well as explore opportunities out of the lab.

I also wish to thank all the members of VLSI lab; working with them made my time during graduate study a wonderful experience.

A countless and sincere thanks goes to my family, especially my wife, Zhang Bei, and my parents, Huang Peikuan and Qiu Sufang, for their continuous support and encouragement throughout my studies.

ABSTRACT

Solving a system of linear and nonlinear equations lies at the heart of many scientific and engineering applications such as circuit simulation, applications in electric power networks, and structural analysis. The exponentially increasing complexity of these computing applications and the high cost of supercomputing force us to explore affordable high performance computing platforms. The ultimate goal of this research is to develop hardware friendly parallel processing algorithms and build cost effective high performance parallel systems using hardware in order to enable the solution of large linear systems.

In this thesis, FPGA-based general hardware architectures of selected iterative methods and direct methods are discussed. Xilinx Embedded Development Kit (EDK) hardware/software (HW/SW) codesigns of these methods are also presented. For iterative methods, FPGA based hardware architectures of Jacobi, combined Jacobi and Gauss-Seidel, and conjugate gradient (CG) are proposed. The convergence analysis of the LNS-based Jacobi processor demonstrates to what extent the hardware resource constraints and additional conversion error affect the convergence of Jacobi iterative method. Matlab simulations were performed to compare the performance of three iterative methods in three ways, i.e., number of iterations for any given tolerance, number of iterations for different matrix sizes, and computation time for different matrix sizes. The simulation results indicate that the key to a fast implementation of the three methods is a fast implementation of matrix multiplication. The simulation results also show that CG method takes less number of iterations for any given tolerance, but more computation time as matrix size increases compared to other two methods, since matrix-vector multiplication is a more dominant factor in CG method than in the other two methods. By implementing matrix multiplications of the three methods in hardware with Xilinx EDK HW/SW codesign, the performance is significantly improved over pure software Power PC (PPC) based implementation. The EDK implementation results show that CG takes less computation time for any size of matrices compared to other two methods in HW/SW codesign, due to that fact that matrix

multiplications dominate the computation time of all three methods while CG requires less number of iterations to converge compared to other two methods.

For direct methods, FPGA-based general hardware architecture and Xilinx EDK HW/SW codesign of WZ factorization are presented. Single unit and scalable hardware architectures of WZ factorization are proposed and analyzed under different constraints. The results of Matlab simulations show that WZ runs faster than the LU on parallel processors but slower on a single processor. The simulation results also indicate that the most time consuming part of WZ factorization is matrix update. By implementing the matrix update of WZ factorization in hardware with Xilinx EDK HW/SW codesign, the performance is also apparently improved over PPC based pure software implementation.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BDB	Bordered-Diagonal-Block
BRAM	Block Random Access Memory
CG	Conjugate Gradient
CGNR	Conjugate Gradient on the Normal Equations
DED	Diagonal Element Detector
EDK	Embedded Development Kit
FIFO	First in First out
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GE	Gaussian Elimination
HDL	Hardware Description Language
HERA	Heterogeneous Reconfigurable Machine
HW/SW	Software/Hardware
ISE	Integrated Software Environment
JPU	Jacobi Processor Unit
JTAG	Joint Test Action Group
LNS	Logarithmic Number System
LU	Lower-Upper
MAC	Multiplication and Accumulation
MGT	Multi-Gigabit Transceiver
MIMD	Multiple-Instruction, Multiple-Data
MMB	Matrix Multiplication Block
MPI	Message Passing Interface
M-SIMD	Multiple-SIMD
NEWS	North, East, West and South
OCM	On-chip Memory
OPB	On-the-Chip-Peripheral Bus
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect

PE	Processing Element
PIE	Parallel Implicit Elimination
PLB	Processor Local Bus
PPC	Power PC
QIF	Quadrant Interlocking Factorization
SDK	Software Development Kit
SIMD	Single-Instruction, Multiple-Data
SOC	System-on-a-Chip
SPICE	Simulation Program with Integrated Circuit Emphasis
TMD	Toronto Molecular Dynamic
UART	Universal Asynchronous Receiver/Transmitter
WZ	Matrices in W and Z shapes
XPS	Xilinx Platform Studio
ZW	Matrices in Z and W shapes

Chapter 1

Introduction

Many scientific and engineering problems, such as circuit simulation, applications in electric power networks, and structural analysis [1, 2, 3], involve solving large systems of simultaneous linear equations. Parallel implementations of these computation intensive processes were limited primarily to multiprocessor computers. However due to the exponentially increasing complexity of these applications, the high cost of supercomputing forces us to explore new, sustainable, and affordable high performance computing platforms. Configurable computing [4], where hardware resources are configured appropriately to match specific hardware designs, has recently demonstrated its ability to significantly improve performance for computing intensive applications. With steady advances in silicon technology, as predicted by Moore's Law, Field Programmable Gate Array (FPGA) technologies [5] have enabled the implementation of System-on-a-Chip (SOC) [6] computing platforms, which, in turn, have given a significant boost to the field of configurable computing.

1.1 Motivation

Driven by the advanced fabrication technology of semiconductor and market demand, complexities of sub 200,000 logic gates on a single chip are now moving to 10 million-plus logic gates with 50 million logic gates in sight [7]. The productivity of semiconductor fabrication over the last twenty years has seen a 58% compounded annual growth; however the productivity of chip design has lagged behind, with only a 21% compounded annual rate. This clearly shows the widening productivity gap between design and fabrication. As product life time is decreasing form 3-5 years to 1-2 years, in many cases as short as a few months,

chip design cycles of months or years are no longer acceptable, since time-to-market is of critical importance in the semiconductor industry.

One of the major goals in semiconductor industry is to increase design productivity, which requires improvements in design methodology. Design methodology is a combination of design software, design flow and design techniques by which designers can follow step-by-step to produce a design that meets its functional specification. Functional verifications are very important steps during a design cycle. It is estimated that the functional verification of a design requires up to 70% of all design time [7]. Digital simulator is one of the verification tools which provide faster simulation results but do not guarantee the performance requirements because many of the essential features are not taken into account, for example: power consumption, non-linearity of load capacitors, parasitic feedbacks, and the influence of temperature. Additional steps and software tools are required for verifying these important effects of design, which increase the complexity of design methodology. Circuit simulation such as Simulation Program with Integrated Circuit Emphasis (SPICE) [8] is able to provide much more accurate results by taking those effects into account, but the existing circuit simulators are time consuming when dealing with large size circuits containing millions of components.

It is known that the most time-consuming task in computer simulation of large systems, such as electronic circuits and power systems, is solving large linear systems. Some efforts [9] have been made to exploit the power of parallel computers in speeding up matrix computations. However, case studies [6, 10, 11] showed hardware accelerator can be 1,000 to 10,000 faster than a software simulator core. The research results did not catch much attention mostly because the industry at that time did not have the design challenges as they are facing today. The motivation of this research is to build cost-effective high performance parallel systems using hardware in order to enable the solution of large linear systems, i.e. the solution of matrices.

Matrix solving methods contains two categories: iterative methods [17] and direct methods [16]. Gaussian elimination (GE) and Lower-Upper (LU) factorization [12, 13] are two of

direct methods used in SPICE simulation to solve matrices. Even though these matrix solving methods have been parallelized to run on parallel computers due to the advent of parallel computers, their performance is not satisfactory since both methods are essentially algorithms in which elimination and factorization are performed serially. Most iterative methods such as Jacobi and conjugate gradient (CG) aim at parallel processing, but accuracy is not guaranteed for limited iterations. Direct methods such as WZ factorization [14] and parallel elimination method (PIE) [15] are able to achieve exact solutions without considering rounding off error and solve matrices in parallel compared to LU and GE but mainly target on dense matrices. Since applications like circuit simulation and power systems produce large size sparse matrices, in order to extend parallel properties of these direct methods, reordering technique like minimum fill-in [16] should be used to transform the sparse matrices into bordered-diagonal-block (BDB) forms where submatrices are dense and can be factorized by multiprocessor in parallel.

One possible solution for improving the performance of matrix solving is to develop hardware friendly matrix solving algorithms which can be efficiently implemented in hardware. Hardware friendly algorithms are normally parallel in nature. The independent evaluation procedures of these algorithms are increased over methods like GE and LU, which makes them well suited for hardware design. The success of the proposed solution for matrix solving will simplify the design methodology, narrow the design productivity gap, and also reduce the design cost.

1.2 Thesis Overview and Objectives

This thesis focuses on the development of hardware friendly parallel processing algorithms and “sea-of-processor” architectures for the hardware accelerator to replace the software simulator core. The algorithms and architectures are verified by building a prototype using Matlab simulation and FPGA-based hardware device respectively. The detailed objectives are listed as follows:

I. Iterative methods

a. Propose FPGA-based general hardware architectures of Jacobi, combined Jacobi and Gauss-Seidel, and CG methods.

b. Compare the performance of three iterative methods based on the results of Matlab simulations and FPGA-based HW/SW codesigns.

c. Convergence analysis of a logarithmic number system (LNS) [19, 20, 21, 22] based Jacobi processor: Due to the complexity of hardware designs of arithmetic units such as multiplier and divider, LNS provides an alternative to floating point with the possibility to simplify arithmetic operations. For matrix solvers, fast multiplication and division operations can be achieved by using addition and subtraction operations on the logarithms of the input data. It is interested to know how the simplified error correction circuit is related to the convergence of Jacobi method.

II. Direct methods

a. Investigate other factorization algorithms, i.e., ZW factorization and X factorization, as alternatives to WZ factorization and PIE respectively.

b. Propose FPGA-based general single unit and scalable hardware architectures of WZ factorization. Analyze the architectures under different constraints.

c. Analyze the performance of WZ factorization based on the results of Matlab simulations and FPGA-based HW/SW codesigns. Extend the BDB form to WZ factorization for parallel processing.

1.3 Thesis Outline

There are four primary topics of interest discussed in this thesis, including hardware implementation of iterative methods, hardware implementation of direct methods, performance analysis and suggestions for future work.

Chapter 2 describes the background of circuit simulation, selected iterative and direct methods (i.e., Jacobi, combined Jacobi and Gauss-Seidel, CG, PIE and WZ factorization) targeting on their parallel property for solving sparse and dense matrices, and several existing

hardware architectures for general applications. Chapter 3 discusses FPGA-based general hardware architectures and LNS-based hardware architectures of Jacobi, combined Jacobi and Gauss-Seidel, and CG, followed by HW/SW codesigns of three iterative methods by using Xilinx Embedded Development Kit (EDK) [18]. In Chapter 4, an implementation of WZ factorization is presented. Firstly, ZW factorization and X factorization are introduced as alternatives to WZ factorization and PIE respectively. Single unit and scalable hardware architectures of WZ factorization are proposed and analyzed under different constraints. Xilinx EDK HW/SW codesign of WZ factorization is also presented followed by reordering techniques [16] dealing with large size sparse matrices. This will lead into Chapter 5 where performance of design simulations is analyzed. Finally, the conclusion and suggestions for future work will be given in Chapter 6.

Chapter 2

Background

The algorithms used in SPICE define the traditional approach to circuit simulation. The goal of this chapter is to provide a brief background of basic concepts of SPICE and major matrix solving methods (i.e., GE and LU factorization) used in SPICE. Since evaluation procedures of GE and LU are not processed in parallel, selected iterative and direct methods (i.e., Jacobi, combined Jacobi and Gauss-Seidel, conjugate gradient, PIE and WZ factorization) which are parallel in nature are introduced for solving sparse and dense matrices. Several existing FPGA based hardware architectures targeting on general applications are also reviewed.

2.1 SPICE Algorithm Overview

SPICE [24, 25] is a general purpose analog electronic circuit simulator and is used to provide analysis of circuits containing active components such as bipolar transistors, field effect transistors, diodes and passive components such as resistors, capacitors and inductors. The program originates from the University of California, Berkeley. SPICE is a powerful program which allows designers to evaluate designs without actually building them.

Most of SPICE algorithms [25] can be explained by the following block diagram shown in Figure 2.1. The key of all algorithms inside SPICE is nodal analysis (blocks 3 and 4) including formulating the nodal matrix and solving the nodal matrix for the circuit voltages. The inner loop (blocks 2-6) finds the solution for nonlinear circuits where nonlinear devices are replaced by equivalent linear models. The solution process starts with an initial guess (block 1), goes through the inner loop (blocks 2-6), and repeats until it reaches convergence. The time domain solution is represented by the outer loop (blocks 7-9), together with the inner

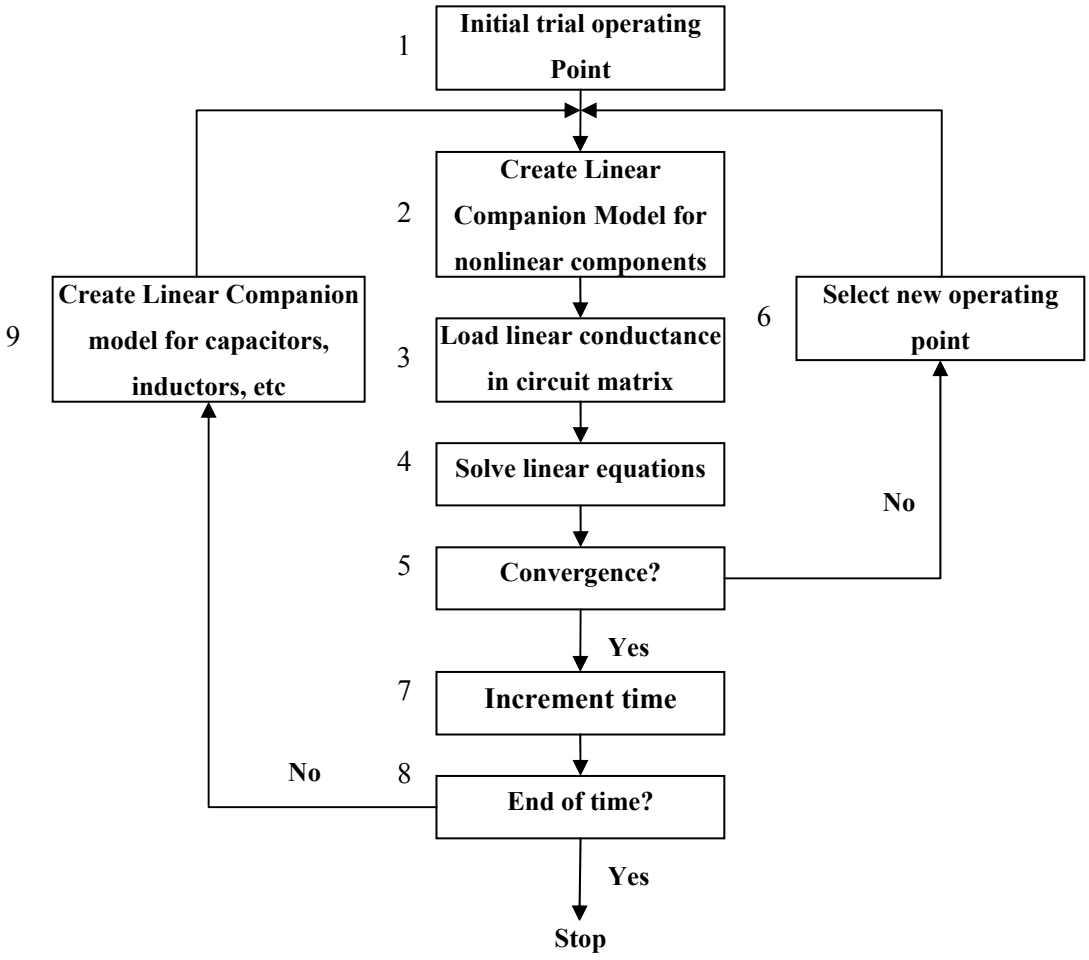

Figure 2.1: Flow chart of SPICE algorithms

loop, it performs a transient analysis creating equivalent linear models for energy-storage components such as capacitors, inductors, etc.

SPICE begins an analysis by reading elements from the input file. Using matrix construction by inspection and a set of predefined element templates [25], system equations are described in a set of linear matrices. SPICE has two solution algorithms, one for linear circuits and one for nonlinear circuits. For linear circuits, only two of the blocks are needed: load the Nodal Matrix (block 3) using Kirchhoff's current law [26] and solve the nodal matrix (block 4) using GE or LU factorization. For Non-Linear circuits, SPICE needs to create equivalent linear models for the non-linear devices such as diode. The loop (blocks 1-6)

iteratively finds the exact solution as follows: guess an operating point, create equivalent linear models and solve the nodal matrix for the circuit voltages. Then, choose a new operating point (block 6) based on the new voltages and start the loop again until the voltage and current reach convergence.

2.2 Matrix Solving Methods Used in SPICE

2.2.1 Gaussian Elimination (GE)

GE and LU factorization [12, 13] are two major methods used in SPICE to solve matrix. GE contains forward elimination and backward substitution. Forward elimination uses scaling of each equation followed by subtraction from the remaining equations in order to eliminate unknowns one by one until matrix A is reduced to an upper triangular matrix. The final solution can then be found by backward substitution which computes each element of x in reverse order. Using GE method, any linear system equation can be solved in at most cubic time. Consider the system of linear equations $Ax = b$ shown in Equation 2.1.

$$\begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & a_{2n} \\ . & . & . & & . \\ . & . & & . & . \\ a_{n1} & a_{n2} & . & . & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix} \tag{2.1}$$

Assume solution $x = \left[x_1^k, x_2^k, ..., x_n^k, \right]$, which can be obtained in two steps:

First step: column elimination.

$$\begin{aligned} e_i &= (a_{i1}, a_{i2}, ..., a_{in}), \\ e_i^{(j)} &= e_i - (a_{ij} / a_{jj}) e_j, \\ b_i^{(j)} &= b_i - (a_{ij} / a_{jj}) b_j, \end{aligned} \tag{2.2}$$

where j = 1, 2,…, n-1 representing j_{th} column elimination stages and $i = j$+1, j+2,…, n representing i_{th} row of matrix A. The first step repeats for j = 1, 2,…, n-1 until the system is transformed into a right triangular matrix shown in Equation 2.3.

$$\begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ 0 & a_{22}^{(1)} & . & . & a_{2n}^{(1)} \\ . & . & . & & . \\ . & . & & . & . \\ 0 & 0 & . & . & a_{nn}^{(n-1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2^{(1)} \\ . \\ . \\ b_n^{(n-1)} \end{bmatrix} \tag{2.3}$$

Second step: back-substitution process. x is solved starting from x_n using Equation 2.4 upwards to x_1 using Equation 2.5. The entire process is done serially.

$$x_n^k = b_n^{(n-1)} \Big/ a_{nn}^{n-1}. \tag{2.4}$$

$$x_i^k = \frac{1}{a_{ii}^{(i-1)}} \left[b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{i,j}^{(i-1)} x_j^k \right] \text{ for } i = n-1, n-2, ..., 1. \tag{2.5}$$

2.2.2 LU Factorization

The LU factorization decomposes matrix as the product of a lower and upper triangular matrices. There are several ways for LU factorization such as Doolittle's method and Crout's methods [12]. Doolittle's method has all 1's in the diagonal of lower triangular matrix as showed in Equations 2.6. The factorization process begins with the first row of upper triangular matrix U using Equation 2.7 followed by the first column of lower triangular matrix L using Equation 2.8. The evaluation process repeats with second row of matrix U followed by the second column of matrix L until the last row of matrix U and the last column of matrix L are found. The Crout's method is similar to Doolittle's except that $U_{kk} = 1$ instead of $L_{kk} = 1$ and the factorization process begins with first column of matrix L followed by the first row of matrix U.

$$\begin{bmatrix} a_{11} & a_{12} & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & a_{2n} \\ . & . & . & & . \\ . & . & & . & . \\ a_{n1} & a_{n2} & . & . & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & . & . & 0 \\ L_{21} & 1 & & & 0 \\ . & . & . & & . \\ . & . & & . & . \\ L_{n1} & L_{n2} & . & . & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & . & . & U_{1n} \\ 0 & U_{22} & . & . & U_{2n} \\ . & . & . & & . \\ . & . & & . & . \\ 0 & 0 & . & . & U_{nn} \end{bmatrix} \tag{2.6}$$

$$U_{ij} = a_{ij} - \sum_{k=1}^{i-1} L_{ik} U_{kj} \qquad \text{for } i = 1, 2, ..., n \text{ and } j = i, i+1, ..., n. \tag{2.7}$$

$$L_{ij} = (a_{ij} - \sum_{k=1}^{j-1} L_{ik} U_{kj}) \frac{1}{U_{jj}} \quad \text{for } j = 1, 2, ..., n-1 \text{ and } i = j+1, j+2, ..., n. \tag{2.8}$$

After factorization of matrix A, Equation 2.1 can be rewritten as $Ax = LUx = b$. For given A and b, the solution x can be obtained in two steps: Firstly, solving the equation $Ly = b$ for y; Secondly, solving $Ux = y$ for x. In these two steps, y and x can be solved directly using forward and backward substitution due to factorized lower and upper triangular matrices. The algorithm description shows that LU factorization is more trivial and requires twice substitution in order to solve x, but it is computationally efficient when a matrix equation is solved for multiple times for different b as compared to GE.

GE and LU factorization have been modified for parallel processing due to the advent of parallel computing [9], their performance is still not satisfactory since both methods are essentially algorithms in which elimination and factorization are processed in serial, i.e., only one row or one column is solved at a time. In next section, selected matrix solving methods which are more suitable for parallel computation and aim at a parallel machine will be introduced.

2.3 Parallel Iterative Matrix Solving Methods

2.3.1 Jacobi Iterative Method

The Jacobi method [12] is an algorithm in linear algebra for determining the solutions of linear systems with largest absolute values in each row and column dominated by the

diagonal elements. Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges. The solution to set of linear equations, expressed in matrix terms as Equation 2.1, where A is an $n \times n$ matrix, is obtained as follows: Let $A = L + D + U$, where L is the lower triangular matrix containing all elements of A below the diagonal, U is the upper triangular matrix containing all elements of A above the diagonal, and D is the diagonal matrix consisting of only the diagonal elements. Substituting $A = L + D + U$ into Equation 2.1 yields

$$x = D^{-1}\left[b - (L+U)x\right] \tag{2.9}$$

The Jacobi method can be expressed as:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left[b_i - \sum_{j \neq i} a_{ij} x_j^k\right] \quad \text{for } i = 1, 2, ..., n. \tag{2.10}$$

Initially guess $x^0 = \left[x_1^0, x_2^0, ...x_n^0\right]$, substitute x^0 into the right-hand side of Equation 2.10 to calculate new, possibly more accurate, values of x_i. This evaluation process repeats until the convergence condition is met. Jacobi will always converge if the matrix A is strictly diagonally dominant, which means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms shown in Equation 2.11.

$$\left|a_{ii}\right| > \sum_{i \neq j} \left|a_{ij}\right| \tag{2.11}$$

The Jacobi method sometimes converges even if this condition is not satisfied. It is necessary, however, that the diagonal terms in the matrix are greater (in magnitude) than the other terms. Furthermore, Jacobi is barely used as a stand-alone solver, but rather as a preconditioner to reduce the condition number, thus increase the rate of convergence for more advanced iterative methods like conjugate gradient [27].

2.3.2 Combined Jacobi and Gauss-Seidel

Unlike Jacobi method, Gauss-Seidel method [12] uses new values of x_i as soon as they become available. For example, when calculating x_2^1, the new value x_1^1 is used instead of the old value of x_1^0. The Gauss-Seidel can be expressed as:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left[b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right] \tag{2.12}$$

There are two important characteristics of the Gauss-Seidel method. Firstly, the computations are processed in serial. Since each component of the new iterate depends upon all previously computed components, the updates cannot be done simultaneously as in the Jacobi method. Secondly, the new iterate x^{k+1} depends upon the order in which the equations are examined. If this ordering is changed, the components of new iterates will also change. A more hardware friendly approach is to combine Jacobi and Gauss-Seidel methods. In this combined method, a number of variables, x, are calculated in parallel dependent upon available hardware resource. If the hardware resource allows maximum p variables to be calculated in parallel, x_1, $x_2 \ldots x_p$ are calculated first. The new values are used for calculating the next p variables, x_{p+1}, $x_{p+2} \ldots x_{2p}$, and so on.

2.3.3 Conjugate Gradient (CG)

Conjugate gradient (CG) [27, 28] is an algorithm for finding the nearest local minimum of a system of n variables which assumes that the gradient of the function can be computed. CG derives its name from the fact that it generates a sequence of conjugate (or orthogonal) vectors and uses conjugate vectors as search directions instead of the local gradient for going downhill until the final solution is reached. CG is effective for the numerical solution of particular linear systems, namely those whose matrix is symmetric and positive-definite, since storage for only a limited number of vectors is required [29, 30, 31].

CG proceeds by generating vector sequences x^{k+1} of iterates, i.e., successive approximations to the solution, residuals r^{k+1}corresponding to iterates, and search directions

%Start

%Begin with first iterate.

$k = 0$

%Set up initial values, where search direction
is equal to the residual for the first iterate.

$d^0 = r^0 = b - Ax^0$

$\delta_{new} = (r^0)^T r^0$

while $k < k_{max}$ do

$\alpha^k = \frac{\delta_{new}}{(d^k)^T A d^k}$

%Update new values of solution x.

$x^{k+1} = x^k + \alpha^k d^k$

%Update new values of residual r.

$r^{k+1} = r^k - \alpha^k A d^k$

$\delta_{old} = \delta_{new}$

$\delta_{new} = (r^{k+1})^T r^{k+1}$

$\beta^k = \frac{\delta_{new}}{\delta_{old}}$

%Update new values of search direction d.

$d^{k+1} = r^{k+1} + \beta^k d^k$

%Increase iterate.

$k = k + 1$

%End

Figure 2.2: Algorithm description of CG

d^{k+1} used in updating iterates and residuals. Considering the linear system $Ax = b$ in Equation 2.1, denote the initial guess for x by x^0. The resulting algorithm is summarized in Figure 2.2. The input vector x^0 can be an approximate initial solution or zero.

CG method can also be applied to an arbitrary system where A is not symmetric, not positive-definite, and even not square by transforming A into normal equations A^TA and right-hand side vector b into A^Tb shown in Equation 2.13, since A^TA is a symmetric positive

definite matrix for any A. The result is called conjugate gradient on the normal equations (CGNR).

$$A^T Ax = A^T b, \tag{2.13}$$

However, the downside of forming the normal equations is that the condition number $\kappa(A^TA)$ is equal to $\kappa(A)^2$ and so the rate of convergence of CGNR may be very slow.

2.4 Parallel Direct Matrix Solving Methods

2.4.1 Parallel Implicit Elimination (PIE)

PIE method for the solution of linear system was introduced by Evans and Abdullah [14, 32]. This method simultaneously eliminates two matrix elements, in stead of just one in GE. Thus, PIE is suitable for parallel implementation. Considering the linear system $Ax = b$ in Equation 2.1, the basis of PIE method is to transform matrix A into butterfly form Z as shown in Equation 2.14 by multiplying matrix A with transformation matrix W. This transformation process is called parallel elimination.

$$Z = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1,n-1} & a_{1,n} \\ & a_{22} & . & . & . & a_{2,n-1} & \\ & & . & . & . & & \\ & 0 & & a_{(n+1)/2.(n+1)/2} & & 0 & \\ & & . & . & . & & \\ & a_{n-1,2} & . & . & . & a_{n-1,n-1} & \\ a_{n,1} & a_{n,2} & . & . & . & a_{n,n-1} & a_{n,n} \end{bmatrix} \text{ for } n = \text{ odd.} \tag{2.14}$$

By combining $WA=Z$ and $Ax=b$, $WAx=Wb$ is obtained, which can be rewritten as $Zx = b'$, where $b' = Wb$. Vector b is also updated once matrix Z is obtained. The solution process of PIE is similar to the solution procedure shown in WZ factorization. In this section, only the transformation process will be introduced, which is summarized as follows: Matrix A is denoted in shorthand form in Equation 2.15.

$$A = \begin{bmatrix} a_{11} & a_{1j} & a_{1n} \\ a_{i1} & A_{ij} & a_{in} \\ a_{n1} & a_{nj} & a_{nn} \end{bmatrix} \text{ for } i, j = 2,\ 3,...,\ n-1. \tag{2.15}$$

For the first evaluation stage, the transformation matrix W is shown in Equation 2.16.

$$W_1 = \begin{bmatrix} 1 & 0 & 0 \\ -w_{i1} & I_{n-2} & -w_{in} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

where i = 2, 3,…, n-1 and I_{n-2} is the unit matrix of order n-2. Elimination is achieved by taking product of W_1 and A.

$$\begin{aligned} W_1 A &= \begin{bmatrix} 1 & 0 & 0 \\ -w_{i1} & I_{n-2} & -w_{in} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{1j} & a_{1n} \\ a_{i1} & A_{ij} & a_{in} \\ a_{n1} & a_{nj} & a_{nn} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{1j} & a_{1n} \\ -a_{11}w_{i1} + a_{i1}I_{n-2} - a_{n1}w_{in} & -w_{i1}a_{1j} + A_{ij} - w_{in}a_{nj} & -a_{1n}w_{i1} + a_{in}I_{n-2} - a_{nn}w_{in} \\ a_{n1} & a_{nj} & a_{nn} \end{bmatrix} \end{aligned} \tag{2.17}$$

Referring to Equation 2.14, in order to meet $W_1 A = Z$, $-a_{11}w_{i1} + a_{i1}I_{n-2} - a_{n1}w_{in}$ and $a_{1n}w_{i1} + a_{in}I_{n-2} - a_{nn}w_{in}$ shown in Equation 2.17 need to be equal to zero. By solving n-2 sets of 2×2 equations, matrix Z_1 can be obtained in the form of Equation 2.18.

$$Z_1 = \begin{bmatrix} a_{11} & a_{1j} & a_{1n} \\ 0 & A'_{n-2} & 0 \\ a_{n1} & a_{nj} & a_{nn} \end{bmatrix} \text{ for } j = 2,3 \ldots n\text{-}1. \tag{2.18}$$

A_{n-2} is the remaining matrix of order n-2, which is updated by Equation 2.19.

$$A'_{n-2} = -w_{i1}a_{1j} + A_{n-2} - w_{in}a_{nj} \tag{2.19}$$

This evaluation process recursively repeats for (n-1)/2 stages. The final matrix Z is obtained in the form of Equation 2.14. An example of PIE is given in Appendix A.

2.4.2 WZ Factorization

WZ factorization was introduced by Evans and Hatzopoulos in 1973 [15, 33]. A method using WZ factorization to solve matrix is called quadrant interlocking factorization (QIF) [14]. WZ method decomposes coefficient matrix A into two interlocking quadrant factors of butterfly form denoted by W and Z or as

$$A = WZ \tag{2.20}$$

where W and Z are shown in Equations 2.21 and 2.22,

$$W = \begin{bmatrix} 1 & & & 0 & & & 0 \\ w_{21} & 1 & & & & 0 & w_{2n} \\ w_{31} & w_{32} & 1 & .. & 0 & w_{3,n-1} & w_{3n} \\ .. & .. & .. & .. & .. & .. & .. \\ w_{n-2,1} & w_{n-2,2} & 0 & .. & 1 & w_{n-2,n-1} & w_{n-2,n} \\ w_{n-1,1} & 0 & & & & 1 & w_{n-1,n} \\ 0 & & & 0 & & & 1 \end{bmatrix} \tag{2.21}$$

and

$$Z = \begin{bmatrix} z_{11} & z_{12} & z_{13} & .. & z_{1,n-2} & z_{1,n-1} & z_{1n} \\ & z_{22} & z_{23} & .. & z_{2,n-2} & z_{2,n-1} & \\ & & z_{33} & .. & z_{3,n-2} & & \\ & 0 & .. & .. & .. & 0 & \\ & & z_{n-2,3} & .. & z_{n-2,n-2} & & \\ & z_{n-1,2} & z_{n-1,3} & .. & z_{n-1,n-2} & z_{n-1,n-1} & \\ z_{n1} & z_{n2} & z_{n3} & .. & z_{n,n-2} & z_{n,n-1} & z_{nn} \end{bmatrix} \tag{2.22}$$

Equation 2.20 can be rewritten as

$$\begin{bmatrix} a_{11} & a_{1i} & a_{1n} \\ a_{i1} & A_{n-2} & a_{in} \\ a_{n1} & a_{ni} & ann \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ w_{i1} & W_{n-2} & w_{in} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_{11} & z_{1i} & z_{1n} \\ 0 & Z_{n-2} & 0 \\ z_{n1} & z_{ni} & z_{nn} \end{bmatrix} \tag{2.23}$$

The elements of matrices W and A can be evaluated in (n-1)/2 distinct stages. The basic steps are summarized as follows.

At the first stage, it can be observed that the elements of first and last rows of the matrix Z are obtained by

$$\begin{aligned} &a_{11} = z_{11}, \;\; a_{1i} = z_{1i}, \;\; a_{1n} = z_{1n} \\ &a_{n1} = z_{n1}, \;\; a_{ni} = z_{ni}, \;\; a_{nn} = z_{nn} \;\; \text{for } i = 2, 3, \ldots, n\text{-}1. \end{aligned} \tag{2.24}$$

The elements of the first and last columns of the matrix W are evaluated by solving (n-2) sets of 2×2 linear systems given by

$$\begin{aligned} &z_{11}w_{i1} + z_{n1}w_{in} = a_{i1} \\ &z_{1n}w_{i1} + z_{nn}w_{in} = a_{in} \quad \text{for } i = 2, 3, \ldots, n\text{-}1. \end{aligned} \tag{2.25}$$

In general, each 2×2 linear system is solved by Cramer's rule using Equation 2.26.

$$\left.\begin{aligned} x_1 &= z_{ii}z_{n-i+1,n-i+1} - z_{n-i+1,i}z_{i,n-i+1}, \\ x_2 &= a_{ji}z_{n-i+1,n-i+1} - z_{n-i+1,i}a_{j,n-i+1}, \\ x_3 &= z_{ii}a_{j,n-i+1} - a_{ji}z_{i,n-i+1}, \end{aligned}\right\} \rightarrow \begin{aligned} W_{ji} &= \frac{x_2}{x_1} \\ W_{j,n-i+1} &= \frac{x_3}{x_1} \end{aligned} \tag{2.26}$$

The remaining matrix A_{n-2} is then updated by Equation 2.27.

$$a_{ij} = a_{ij} - w_{i1}z_{1j} - w_{in}z_{nj} \;\; \text{for } i, j = 2, 3 \ldots n\text{-}1 \tag{2.27}$$

For the following evaluation stages, this evaluation process is recursively repeated for the remaining matrices A_{n-2i} (i.e. A_{n-2}, $A_{n-4} \ldots 2$ for n is even and A_{n-2}, $A_{n-4} \ldots 1$ for n is odd.).

In order to solve the system by the QIF method, $Ax=b$ can be rewritten as $(WZ)x = b$. Only two related and simpler linear systems of forms $Wy = b$ and $Z\,x = y$ need to be solved. The solution procedure for y is carried out in pairs from top and bottom. In general, at i_{th} stage,

$$y_i = b_i \;\; \text{and} \;\; y_{n-i+1} = b_{n-i+1} \tag{2.28}$$

Then update b_i using Equation 2.29.

$$b_j^{'} = b_j - w_{ji}y_i - w_{j,n-i+1}y_{n-i+1} \;\; \text{for } j = i+1, \; i+2 \ldots . \; n-i. \tag{2.29}$$

The final solution x is computed from the middle of matrix Z. This process can be distinguished into two cases: n is odd and n is even. If n is odd, $x_{(n+1)/2}$ is obtained by Equation 2.30.

$$x_{\frac{1}{2}(n+1)} = \frac{y_{\frac{1}{2}(n+1)}}{z_{\frac{1}{2}(n+1),\frac{1}{2}(n+1)}} \tag{2.30}$$

Update y for next stage,

$$y'_j = y_{j-}x_{\frac{1}{2}(n+1)}z_{,\frac{1}{2}(n+1)} \text{ for } j = 1,2...n, j \neq \frac{1}{2}(n+1) \tag{2.31}$$

The remaining elements of x can be obtained in pairs by solving (n-1)/2 sets of 2×2 linear equations. If n is even, x can be evaluated directly by solving n/2 sets of 2×2 linear equations starting with $x_{(n-1)/2}$ and $x_{(n+1)/2.}$ In general, at i_{th} stage, solve Equation 2.32 to obtain the values of x,

$$\begin{aligned} z_{ii}x_i + z_{i,n-i+1}x_{n-i+1} &= y_i \\ z_{n-i+1}x_{i+zn-i+1}x_{n-i+1} &= y_{n-i+1} \end{aligned} \tag{2.32}$$

and update y_j in Equation 2.33.

$$y_j = y_j - x_i z_{ji} - x_{n-i+1}z_{j,n-i+1}. \tag{2.33}$$

An example of WZ factorization is given in Appendix A.

2.4.3 Comparison of Direct Methods

As LU factorization compared to GE, WZ factorization is more trivial and requires twice substitution in order to solve x, but is computationally efficient to solve a matrix equation multiple times for different b compared to PIE. According to the similarity of evaluation processes, the first type of comparison can be made between PIE and GE. During the elimination stage of PIE method, two columns of transformation matrix W are solved or two rows of matrix Z are eliminated simultaneously. The solution stage of PIE starts from the middle of vector x being completed bi-directionally in parallel resulting in increased stability. GE is only able to eliminate one column and solve one x at a time. In general, the timings on multiprocessor for PIE are better than GE. For the larger matrices the gains of speed up vary

from 6% for 1 processor to 10% for 10 processors [14]. These gains will increase for larger matrices and larger number of processors.

The second type of comparison can be made between WZ and LU. The WZ factorization solves W elements from left and right and Z elements from top and bottom bi-directionally which is similar as PIE method. The solution stage of WZ solves two values simultaneously from the top and bottom moving inwards bi-directionally for vector y and from the middle moving outwards bi-directionally for vector x. LU is only able to solve one row of U followed by one column of L at a time. The solution stage of LU is also processed in serial. The timings on the sequent multiprocessor show that the WZ factorization is faster than LU and for larger matrices, the gains appear to be 20% for all values of processor [33].

2.5 Hardware Architectures for General Applications

The flexibility, re-programmability and run-time reconfigurability of FPGAs have great potential to offer an alternative computing platform for high performance computing. Recent significant advances in FPGA technology and the inherent advantages of configurable logic have brought new research efforts in the configurable computing field: parallel processing on configurable chips [34, 35]. In this section, three FPGA based hardware architectures are reviewed for general applications.

2.5.1 Toronto Molecular Dynamic (TMD) Architecture

TMD was designed for molecular dynamics simulations [36, 37]. This architecture can also be used to solve other computing-intensive problems. The architecture is built entirely using FPGA computing nodes which are implemented by Virtex-II Pro XC2VP100 FPGAs [38]. The machine enables designers to implement large-scale computing applications using a heterogeneous combination of hardware accelerators and embedded microprocessors spread across many FPGAs, all interconnected by three levels communication networks.

The TMD architecture is divided into there hierarchical tiers shown in Figure 2.3, allowing it to scale up to reconfigurable machine containing many FPGAs. The lowest tier exists within

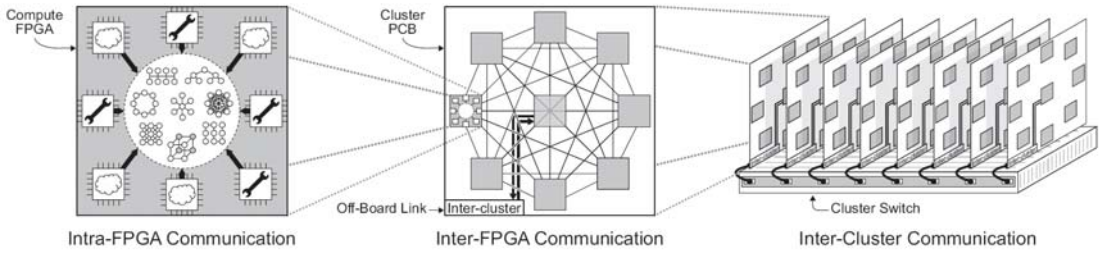

Figure 2.3: TMD architecture hierarchy [36].

FPGA, where different topologies can be specified for interconnecting computing engines and embedded processors. The middle tier is based on cluster printed circuit board (PCB) level which consists of 8 FPGAs for computing purpose and 1 FPGA for communicating with other clusters. The highest tier is the large network by interconnecting multiple clusters.

TMD communications networks can be divided into there levels. The first level is intra-FPGA communication which is implemented using point-to-point unidirectional FIFOs. The second level is inter-FPGA communication which uses multi-gigabit transceiver (MGT) [39] hardware to implement communication between FPGAs. The third level is inter-cluster communication by aggregating four MGT links, enabling the use of infiniband switches for implementing global interconnection between clusters.

2.5.2 Hardware/Software (HW/SW) Partitioned Computing System

Usually, a reconfigurable computing system has multiple nodes which can be implemented by processors, FPGAs, or both. This hybrid architecture [40] utilized both the processors and the FPGAs in the system for computing purpose, as shown in Figure 2.4. The design is based on Cray XD1 [41]. The basic unit is a computing blade, which consists of two AMD 2.2 GHz processors and one Xilinx Virtex-II Pro XC2VP50 [38]. Six computing blades fit into one chassis, interconnected by a non-blocking cross-bar switching fabric which provides two 2GB/s links to each node. The nodes communicate using Message Passing Interface (MPI) [42]. In this system, only the processors of the blades (nodes) are connected through communication network.

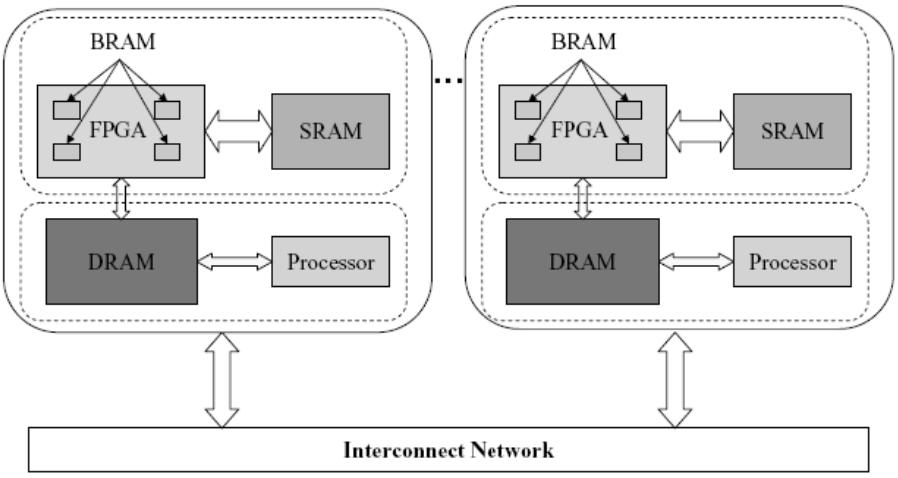

Figure 2.4: HW/SW partitioned system architecture [40].

For different applications, tasks capable of different functions need to be specified. The HW/SW partition is based on the workload of the task so that processor and FPGA are both fully utilized. In other word, the computation times of processor and FPGA need to be equal considering the data transfer time and communication costs. Other than workload partition, the coordination between processor and FPGA is also very important.

2.5.3 Mixed-Mode Heterogeneous Reconfigurable Machine (HERA) Architecture

The HERA machine [43] is based on Xilinx Virtex-II and Virtex-II pro platform FPGAs. This machine can implement the single-instruction, multiple-data (SIMD), multiple-instruction, multiple-data (MIMD) and multiple-SIMD (M-SIMD) execution modes in one machine.

Figure 2.5 shows the general architecture of HERA machine with $m \times n$ processing elements (PEs) interconnected by a 2-D mesh network. The architecture employs fast, direct North, East, West and South (NEWS) connections for communications between nearest neighbors. The global communication is achieved by the Cbus and column bus. Every column has a Cbus and all the Cbuses are connected to the column bus. Every PE is built on a single-precision IEEE 754 FPU [44, 45] with tightly-coupled local memory shown in Figure 2.6, and supports

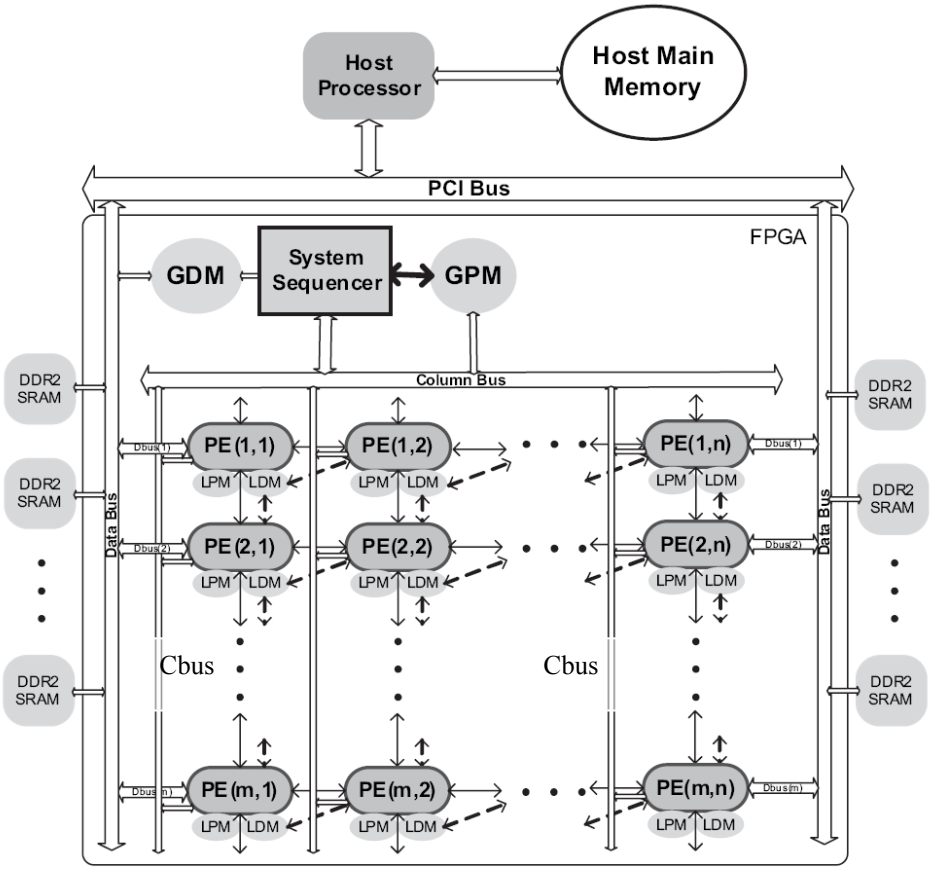

Figure 2.5: HERA system architecture [43].

dynamic switching among SIMD, MIMD and M-SIMD at runtime. Most of the instruction decoding is carried out by the local control unit within PE. The computing process is controlled by a system sequencer that communicates with the host processor via the peripheral component interface (PCI) bus. The capabilities of each PE and the number of PEs can be reconfigured on the basis of the application's requirements and available resources in target FPGA devices respectively. The operating mode of each PE is configured dynamically by the host processor through the operating mode register of PE.

In this chapter, background related to SPICE, selected parallel matrix solving methods and the existing FPGA-based hardware architectures for general applications have been reviewed. GE and LU are two major direct matrix solving methods used in SPICE, which are processed in serial. To improve the performance of matrix solving process, FPGA-based hardware

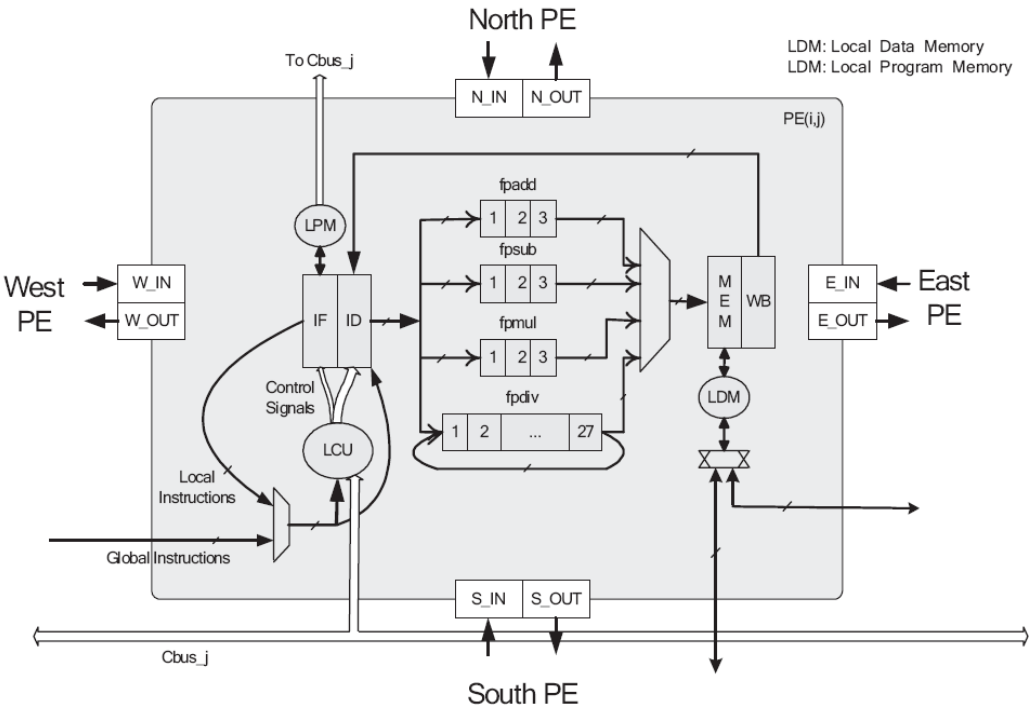

Figure 2.6: HERA PE architecture [43]

implementations of parallel matrix solving methods are desired. Most iterative methods such as Jacobi and CG aim at parallel processing for solving large sparse matrices. In Chapter 3, FPGA-based general hardware architectures, LNS-based hardware designs and FPGA-based HW/SW codesigns of Jacobi, combined Jacobi and Gauss-Seidel, and CG are presented.

Chapter 3

General Hardware Architectures and HW/SW Codesigns of Iterative Methods

In computational mathematics, an iterative method attempts to solve a problem (for example an equation or a linear system) by using successive approximations to obtain more accurate solutions at each step starting from an initial guess. The goal of this chapter is to provide detailed approaches to FPGA-based hardware design including FPGA-based general hardware architectures, LNS based hardware designs and FPGA-based HW/SW codesigns to Jacobi, combined Jacobi and Gauss-Seidel, and CG for solving sparse matrices.

3.1 FPGA-based Hardware Architectures

3.1.1 Jacobi Iterative Method

Assume no memory constraint. An ideal hardware architecture for Jacobi method would consists of n Jacobi processor units (JPU) for n-vector x, where JPU is the basic unit of Jacobi hardware architecture. Within each JPU there are n-1 multipliers, 1 divider, and binary tree adders/subtractors as showed in Figure 3.1. This architecture requires $n\times(n\text{-}1)$ multipliers in total. Multiplication of inputs a_{ij} and x_j are processed in parallel for each x_i and new n-vector x is updated in parallel as well, where $i, j = 1, 2,\ldots, n$ and $j \neq i$. Considering the hardware resource, several other possible cases have been considered, where different case has different JPU.

1. n JPUs, 1 multiplier in each JPU: Multiplications of inputs a_{ij} and x_j are processed serially in each JPU. For each iteration, output for each x will be obtained and updated

in parallel which means new value for each x is updated once all the outputs new x_i are available.

2. 1 JPU and n-1 multipliers in the JPU: Multiplication of inputs a_{ij} and x_j are processed in parallel for each x. Output x's will be obtained and updated serially. This architecture is actually equivalent to an ideal architecture for Gauss-Seidel method.

3. p ($p<n$) JPUs and 1 multiplier in each JPU: A subset of n-vector x are processed in parallel at a time which requires $\lceil n/p \rceil$ number of times to process the n-vector x, where $\lceil\ \rceil$ is a ceiling function. Inside each JPU, multiplications of inputs a_{ij} and x_j are processed serially.

4. 1 JPU and l multipliers in each JPU: A subset of multiplications of inputs a_{ij} and x_j are processed in parallel at a time which requires $\lceil (n-1)/l \rceil$ number of times to process one x_i.

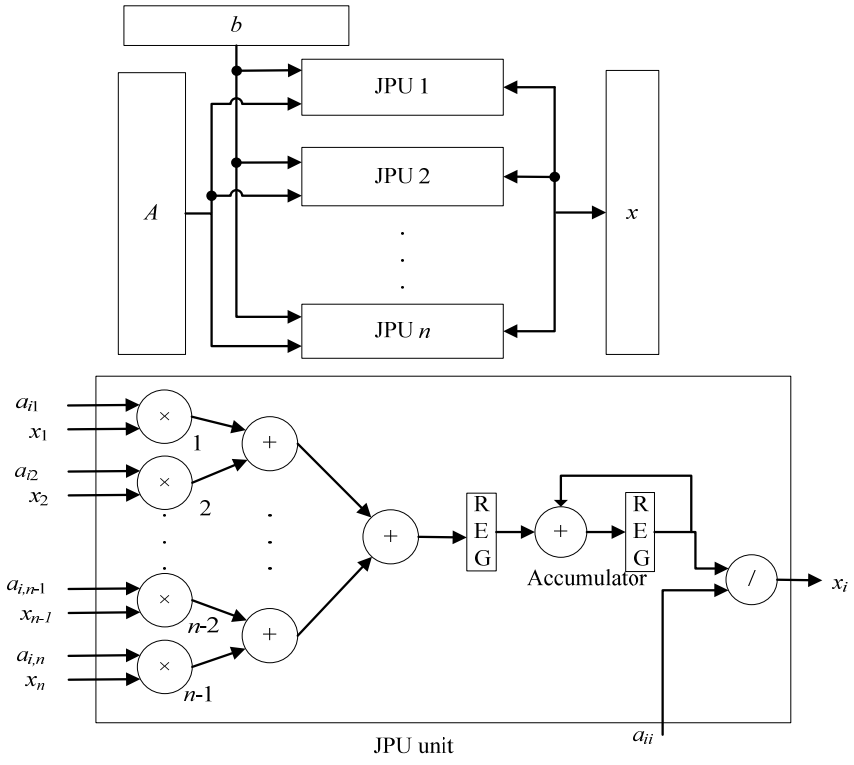

Figure 3.1: Ideal hardware architecture for Jacobi method

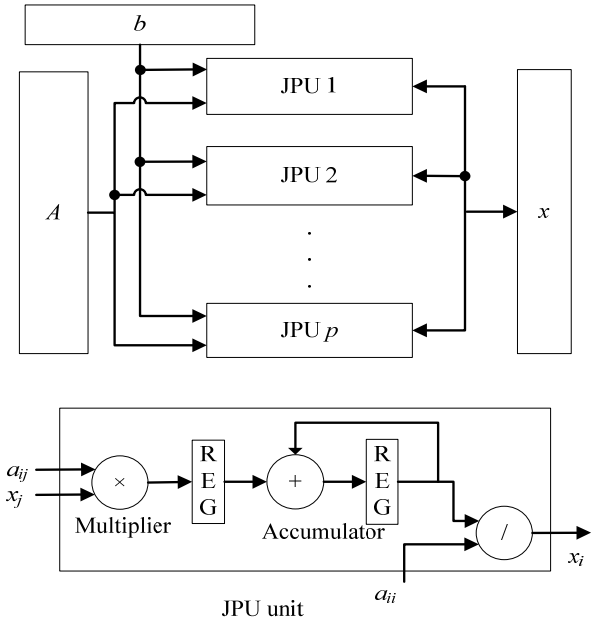

Figure 3.2: Hardware architecture for Jacobi method (case 3)

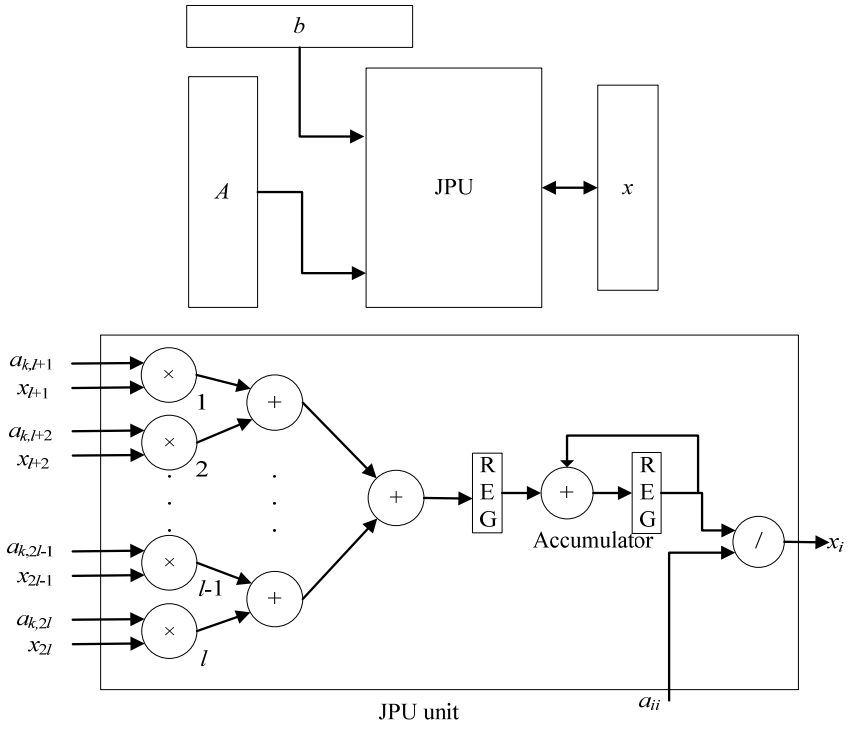

Figure 3.3: Hardware architecture for Jacobi method (case 4)

Cases 1 and 2 are applicable to small linear systems which require less hardware resource for implementation. Cases 3 and 4 have scalable architectures and can be applied to large linear systems considering the available hardware resource in reality. Figure 3.2 shows a hardware architecture which implements case 3. For each iteration, a set of coefficients, $a_{p+1,q}$, $a_{p+2,q}$,…, $a_{2p,q}$ from A and x_q from x are placed at the inputs of JPU 1 to p. After n clock cycles (i.e. for q = 1 to n), a total of p number of new x_i are generated and loaded into x. The first set of new x_i are x_1, x_2,…,x_p, the second set of new x_i are x_{p+1}, x_{p+2},…,x_{2p}, and so on. The second iteration repeats from x_1, x_2,…,x_p until convergence has been reached.

Figure 3.3 shows a hardware architecture which implements case 4. This architecture parallelizes multiplications inside each JPU instead of having multiple JPUs processed in parallel. For each iteration, a set of coefficients, $a_{k,l+1}$, $a_{k,l+2}$,…,$a_{k,2l}$ from A and x_{l+1}, x_{l+2},…,x_{2l} from x are placed at the inputs of multiplier 1 to l. After $\lceil (n-1)/l \rceil$ clock cycles, a new x_i is generated and loaded into x. All new x_i's will be generated in serial. The second iteration again repeats from x_1 until convergence has been reached.

One major difference between cases 3 and 4 is that how new values of x_i are updated. In case 3, a total of p new values of x_i can be updated at a time with inputs processed in serial, while in case 4 only one new value of x_i will be updated at time but at a faster rate of update of each new x_i compared to case 3. Also, case 3 requires one accumulator for the product of a_{ij} and x_j in each JPU, while case 4 requires additional binary tree adders/subtractors in order to take advantage of parallel multipliers [46].

3.1.2 Combined Jacobi and Gauss-Seidel Method

Assume no memory constraint. The ideal hardware architecture of Gauss-Seidel method is equivalent to case 2 of hardware architectures of Jacobi, which consists of 1 JPU for updating x_i. Within JPU, there are n-1 multipliers, 1 divider, and binary tree adders/subtractors. This architecture is able to process the multiplication of a_{ij} and x_j in parallel inside JPU, but only update one new x_i at a time. However, considering the hardware resource constraint, it is difficult to realize completely parallel multiplications inside JPU in particular for large linear

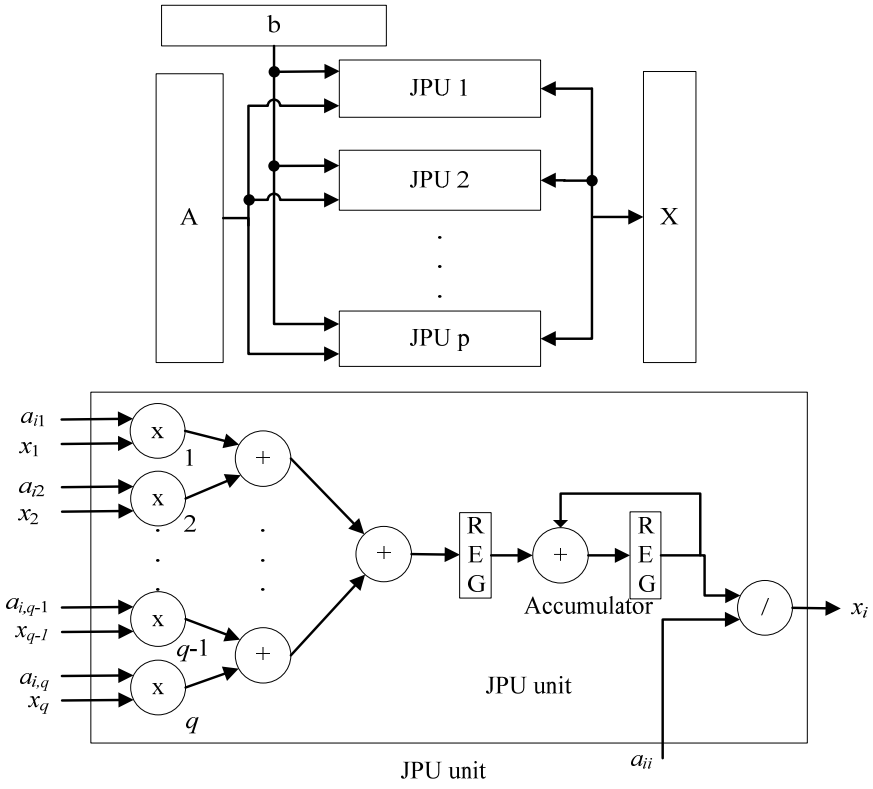

Figure 3.4: Hardware architecture for combined Jacobi and Gauss-Seidel method

systems. A more realistic approach is to combine Jacobi and Gauss-Seidel method which requires $p \times q$ multipliers in total, where p is the number of JPUs and q is the number of multipliers inside each JPU. Given limited hardware resource, there is a trade off between the number of JPUs and multipliers (i.e. p and q). In this combined method, a subset of variables, x, are calculated in parallel dependent upon available hardware resource. If the hardware resource allows maximum p variables to be calculated in parallel, $x_1, x_2, \ldots, x_p$ are calculated first. The new values are used for calculating the next p variables, $x_{p+1}, x_{p+2}, \ldots, x_{2p}$, and so on.

Figure 3.4 shows a hardware architecture which implements the combined Jacobi and Gauss-Seidel method. This architecture contains p JPUs processed in parallel and q multipliers inside each JPU. For each iteration, a set of coefficients, $a_{i,q+1}, a_{i,q+2}, \ldots, a_{i,2q}$ from A and $x_{q+1}, x_{q+2}, \ldots, x_{2q}$ from x are placed at the inputs of multiplier 1 to q inside JPU. After $\lceil (n-1)/q \rceil$ clock cycles, a total of p number of new x_i are generated and loaded into x. The

first set of new x_i are $x_1, x_2,\ldots,x_p$, the second set of new x_i are $x_{p+1}, x_{p+2},\ldots,x_{2p}$, and so on. The second iteration again repeats from $x_1, x_2,\ldots,x_p$ until convergence has been reached.

3.1.3 Conjugate Gradient Method

As described in Chapter 2, CG used as iterative method is memory-efficient and runs quickly with sparse matrices. Figure 3.5 shows one way to implement CG in hardware. This architecture consists of four matrix-multiplication blocks (MMB), two dividers, and two adder/subtractors. Matrix-vector multiplication, vector-vector multiplication and scalar-vector multiplication are implemented in the same block, i.e., MMB. Scalar-vector multiplication is treated as matrix-vector multiplication by transforming the scalar into a diagonal matrix with all diagonal values equal to the scalar value, where a diagonal matrix is a square matrix in which the entries outside the main diagonal are all zero.

In this realization, there are five global signals. The first global signal, *sel_r*^{0}, is used to determine when to select $b - Ax$ and $r - \alpha Ad$ associated with r^{o} and r^k, where r^0 is the initialized residual, r^k is the new residual and k is the number of current iteration. The second global signal, *sel_d*^{0}, is used to determine when to select d^0 and d^k, where d^0 is the initialized search direction and d^k is the current search direction. The third global signal, *sel_accu*, is used to accumulate multiplications to obtain $d^{\text{T}}Ad$. The fourth global signal, *sel_αβ*, is used to determine when to select $x + \alpha d$ and $r + \beta d$ associated with x^{k+1} and d^{k+1}, where x^{k+1} is the new approximation to solution x. The last global signal, *sel_x*^{0}, is used to determine when to select x^0 and x^{k}. Divider_1 and Divider_2 perform scalar divisions to obtain α and β respectively, where α and β are associated with calculations of x, r and d.

This architecture can be divided into three blocks including BLOCK_A, BLOCK_B and BLOCK_C shown in Figure 3.5. Only one block will be enabled at a time. For the first iteration, BLOCK_A is enabled at the beginning, where *sel_r*^{0} selects b, A, and x^0 to calculate the initial residual r^0 which is also equal to the initial search direction d^0, and δnew is obtained by multiplying r^0 with $(r^0)^{\text{T}}$. Then BLOCK_B is enabled, where *sel_d*^{0} selects d^0, *sel_accu* is used to accumulate the multiplication results, $(d^0)^{\text{T}}Ad^0$, and Divider_1 performs a scalar division between δnew and $(d^0)^{\text{T}}Ad^0$ to obtain α^0. Then BLOCK_C is enabled, where

sel_x^0 selects x^0 and *sel_αβ* selects x^0 and α^0 to calculate x^1. After x^1 is obtained, BLOCK_A is enabled again, where *sel_r*^{0} selects r^0, α^0, and Ad^0 to calculate the second residual r^1, *δold* is set to *δnew*, and *δnew* is renewed by multiplying r^1 with $(r^1)^T$. Divider_2 performs a scalar division between *δnew* and *δold* to obtain β^0. At the end of this iteration, BLOCK_ C is enabled again, where *sel_αβ* selects r^1 and β^0 to calculate d^1.

For second iteration, BLOCK_B is enabled at the beginning, where *sel_d*^{0} selects d^1, *sel_accu* is used to accumulate the multiplication results, $(d^1)^T Ad^1$ and Divider_1 performs a scalar division between *δnew* and $(d^1)^T A\ d^1$ to obtain α^1. Then BLOCK_C is enabled, where *sel_x*^{0} selects x^1 and *sel_αβ* selects x^1 and α^1 to calculate x^2. After x^2 is obtained, BLOCK_A is enabled, where *sel_r*^{0} selects r^1, α^1, and Ad^1 to calculate the third residual r^2, *δold* is set to *δnew* and *δnew* is renewed by multiplying r^2 with $(r^2)^T$. Divider_2 performs a scalar division between *δnew* and *δold* to obtain β^1. At the end of this iteration, BLOCK_ C is enabled again, where *sel_αβ* selects r^2 and β^1 to calculate d^2. For following iterations, the process repeats as second iteration until convergence has been reached.

The above analysis shows that the key to a fast implementation of CG is a fast implementation of MMB. A simplified multiplication and accumulation (MAC) unit for matrix multiplication is shown in Figure 3.6. Assume a_{ij} and x_j can be available as inputs data simultaneously, the MAC unit computes $a_{ij} \times x_j$, and then adds $a_{ij}x_j$ to the accumulate register, where accumulate register is initialized to zero. The MAC unit accumulates the products of inputs fed every cycle. After *n* (matrix size) cycles, one element of resulted vectors will be obtained and stored into memory until all of inputs are processed. CG is usually completed after *n* iterations. In practice, the recursive formula for the residual, $r^{k+1} = r^k - \alpha^k Ad^k$, results in accumulated floating point round off error which will cause the residual *r* to gradually lose accuracy. The evaluation process will be terminated early due to this floating point round off error. This problem can be corrected by recalculating the exact residual, $r^{k+1} = b - Ax^k$, where global signal *sel_r*^{0} selects *b*, *A*, and x^k to calculate r^{k+1} in BLOCK_A.

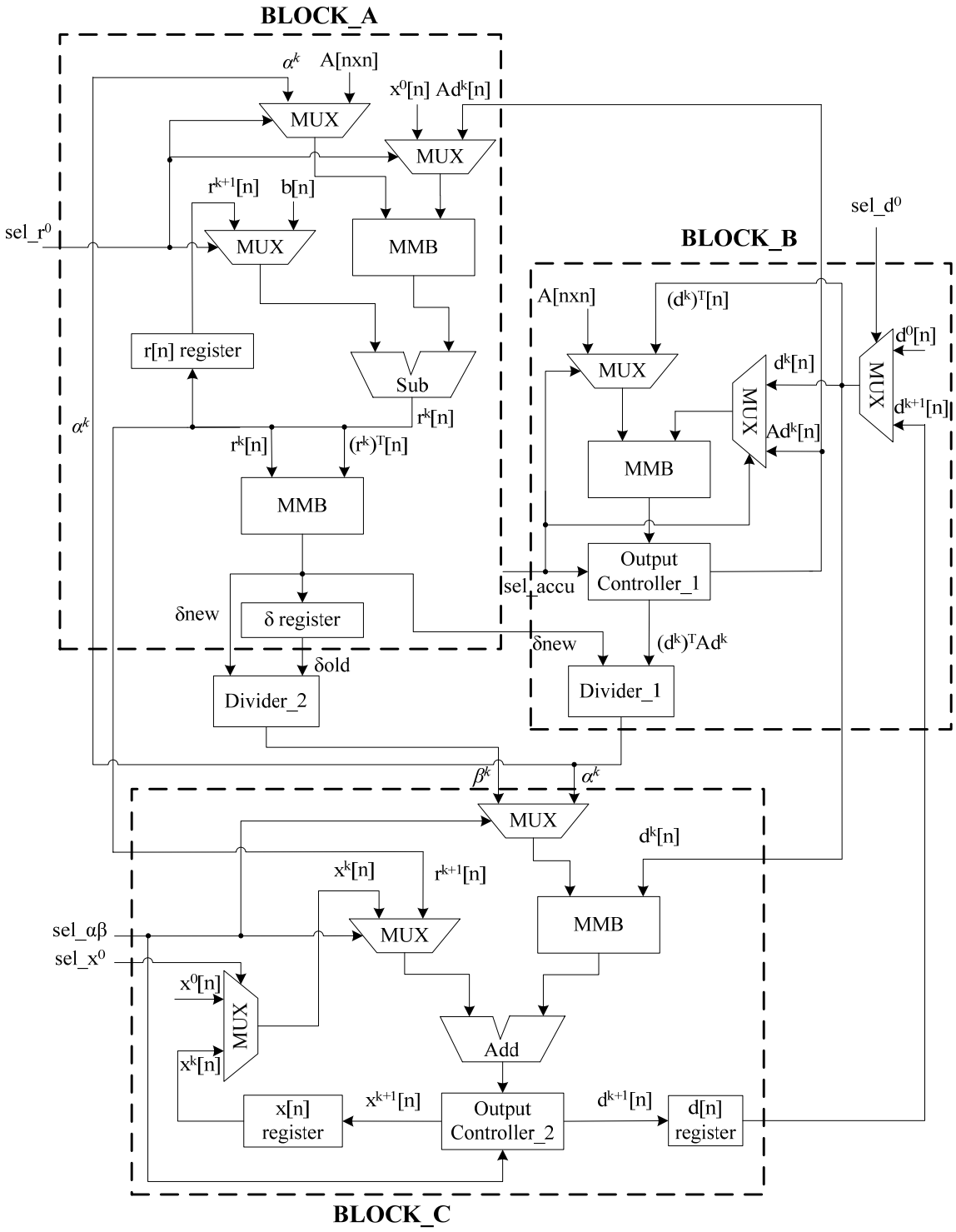

Figure 3.5: General hardware architecture for CG algorithm

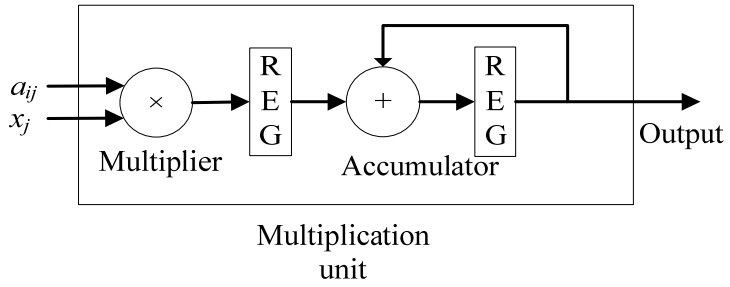

Figure 3.6: Hardware architecture for multiplication unit

3.2 LNS-based Hardware Design of Jacobi Processor

Due to the complexity of hardware designs of arithmetic units such as multiplier and divider, LNS has been studied in an effect to simplify arithmetic computations for lower computation complexity, higher computation speed, and smaller counts size [19, 20, 21, 22]. In LNS, fast multiplication and division operations can be achieved by using addition and subtraction operations on the logarithms of the input data; i.e., the hardware cost of multiplications and divisions are similar in LNS. The reduced circuit size and possibly increased speed of multiplication and division make LNS becomes a viable solution to many computational intensive applications such as hardware implementation of matrix solving.

Figure 3.7 shows one way to implement LNS-based JPU unit [51] for case 3 described in Section 3.1.1, where logarithm and antilogarithm converters can be implemented according to [19, 21]. In this realization, two logarithm converters are used for the multiplications of a_{ij} and x_j as well as the division of $b_i - \sum_{j \neq i} a_{ij} x_j^m$ and a_{ii}. The global signal, *sel_div*, is used to determine when to select $b_i - \sum_{j \neq i} a_{ij} x_j^m$ and a_{ii} to perform LNS-based division. A second global signal, *sel_*b_i, is used to determine when the accumulator is used for executing additions and subtractions associated with $\sum_{j \neq i} a_{ij} x_j^m$ and $b_i - \sum_{j \neq i} a_{ij} x_j^m$. An external signal, *a*$_{ii}$*_en*, from a

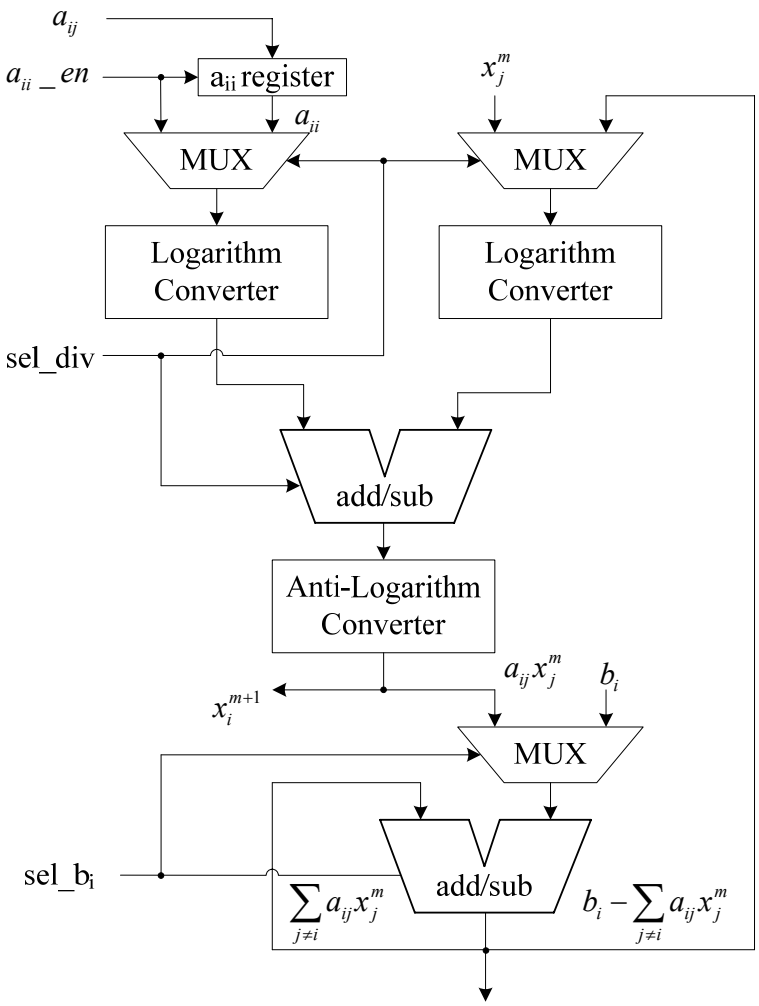

Figure 3.7: Hardware architecture for LNS-based Jacobi processor

diagonal element detector (DED) (not shown in Figure 3.7), is used to pick and store the diagonal element, a_{ii} for individual JPUs. Each JPU has a DED as showed in Figure 3.8, where *input_addr* is the base address of certain row corresponding to that JPU. An initial address denoted as *init_addr* is obtained by *input_addr* + (p-1), where p is the sequence number of JPU. For first subset of x_i, *sel_init_addr* is set to high and the diagonal element address *diag_addr* is equal to *init_addr*. For second subset of x_i, the *sel_init_addr* is set to low the *diag_addr* is obtained by adding *init_addr* with $n_p(n+1)$, where n_p is the total number of JPU, n is the size of matrix A, and $n_p(n+1)$ is a pre-calculated number. For the following subsets of x_i, *sel_init_addr* is set to low and the *diag_addr* is obtained by adding the

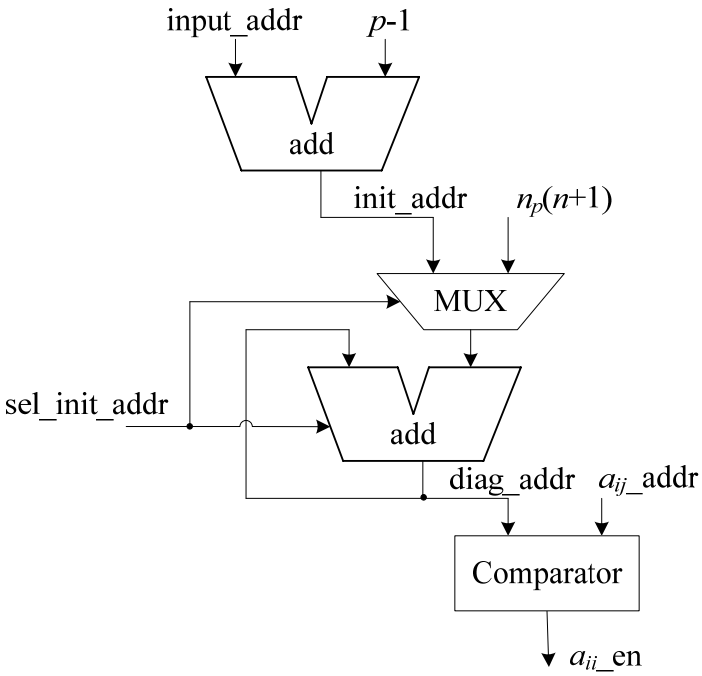

Figure 3.8: Hardware architecture for DED

diag_addr again with $n_p(n+1)$. This process repeats until the first iteration is done. The following iteration begins again by setting *sel_init_addr* to high, thus *diag_addr* is reset to 0 and *init_addr* is selected for the first subset of x_i. The obtained *diag_addr* will be compared with a_{ij}*_addr*, where a_{ij}*_addr* is the address of matrix element. If *diag_addr* is equal to a_{ij}*_addr*, matrix element a_{ij} will be chosen and stored as diagonal element by a_{ii}*_en*.

LNS-based combined Jacobi and Gauss-Seidel can be implemented in a similar way as LNS-based Jacobi Processor where multiplication and division are replaced with LNS-based design. For CG method, MMB takes up the major part of multiplication which can also be replaced with a LNS-based block.

3.3 Xilinx EDK HW/SW Codesign of Iterative Methods

Xilinx EDK [18] is a suite of tools and IP blocks that designs a complete embedded processor system for implementation in a Xilinx FPGA device. The suite includes Xilinx platform studio (XPS), software development kit (SDK), hardware IP for the Xilinx

embedded processors, drivers and libraries for embedded software development, and GNU compiler and debugger for C/C++ software development. XPS is the development environment used for designing the hardware portion of embedded processor system and SDK is an integrated development environment, complimentary to XPS, that is used for C/C++ embedded software application creation and verification. Xilinx EDK tools are able to design a system using embedded MicroBlaze™ soft processor cores implemented using FPGA fabric, and/or PowerPC™ (PPC) hard processor cores, i.e., the fixed CPU cores incorporated into FPGA fabric. The MicroBlaze soft processor core has access to a high-speed serial interface called the Fast Simplex Link (FSL) which is an on-chip interconnect that provides a high-performance data channel between the MicroBlaze processor and the surrounding FPGA fabric. Similarly, the PowerPC hard processor core provides high-performance communication channels through the processor local bus (PLB) and on-chip memory (OCM) interfaces.

To use EDK, integrated software environment (ISE) [52] must be installed as well. ISE is the foundation for Xilinx FPGA logic design, which includes tools related to embedded processor systems and their design. Because FPGA design can be an involved process, Xilinx has provided ISE that allow the designer to circumvent some of this complexity such as constraints entry, timing analysis, logic placement and routing, and device programming have all been integrated into ISE.

A simplified design flow for an embedded design using Xilinx EDK tools is showed in Figure 3.9 [18]. The design enables the integration of both hardware and software components of an embedded system. Typically, the ISE FPGA development software runs behind the scene. The XPS tools make function calls to the utilities provided by the ISE software. XPS is used primarily for embedded processor hardware system development, where specification of the microprocessor, peripherals, and the interconnection of these components, along with their respective property assignments takes place. Simple software development can also be accomplished from within XPS, but for more complex application development and debug, Xilinx recommends using the SDK tool. Verifying the correct functionality of hardware platform can be accomplished by running the design through a

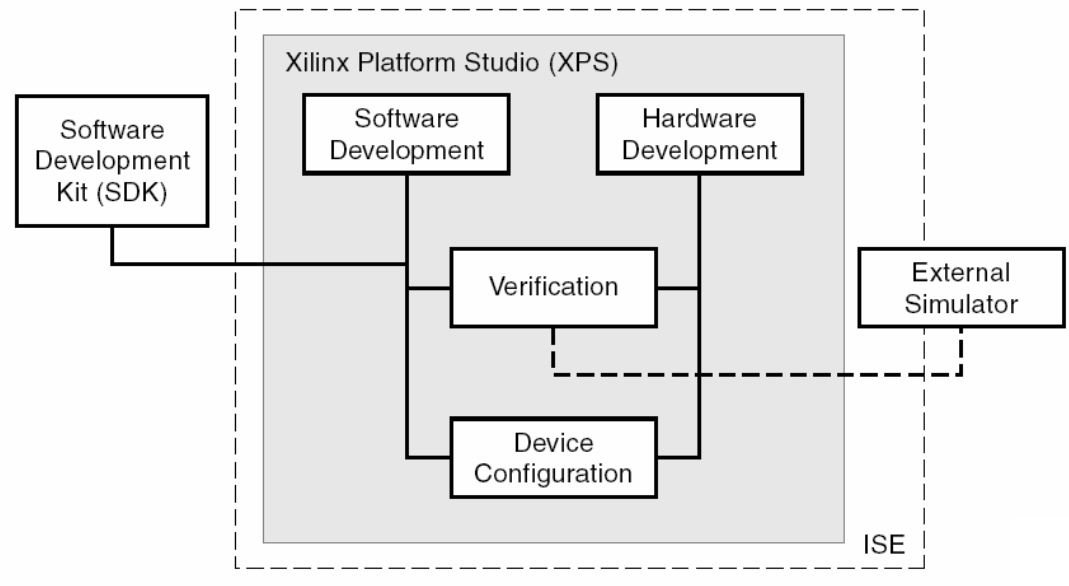

Figure 3.9: Basic embedded design process flow [18].

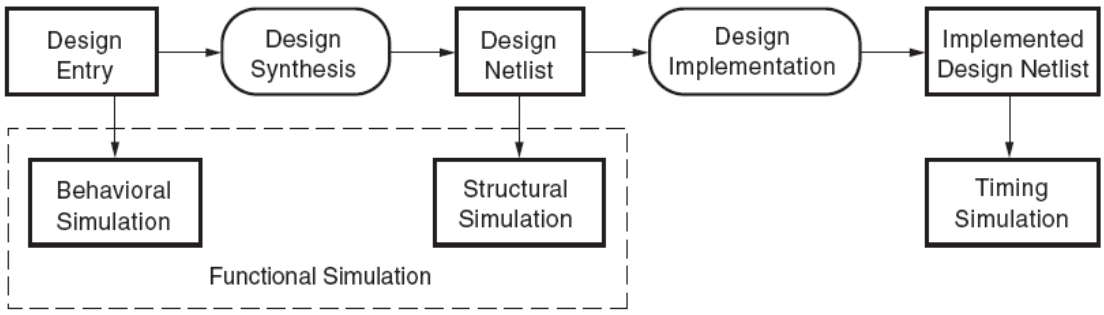

Figure 3.10: EDK design simulation stages [18].

hardware description language (HDL) simulator. XPS facilitates three types of simulation including behavioral, structural and timing-accurate. Verification through behavioral, structural, and timing simulation can be performed at specific points in design process, as illustrated in Figure 3.10 [18]. After completing the design, FPGA bitstream along with the software executable and linkable format file (ELF) are downloaded into target board to configure the target device.

The Xilinx EDK design is implemented in Xilinx Virtex-II Pro XC2VP30 based platform FPGA. The HW/SW codesign architecture for Jacobi, Gauss-Seidel and CG is shown in Figure 3.11. This architecture includes the following hardware components:

- PPC 405: The brain of the system, the microprocessor.
- PLB_BUS: The processor local bus. PLB_BUS is the higher hierarchy bus, the one closer to the processor. Primary instruction and data memory are transferred through this bus.
- OPB_BUS: The on-chip-peripheral bus. Slow and non-critical peripheral is attached to this bus.
- MMB: The matrix multiplication block, which implements matrix multiplications of three iterative methods in FPGA.
- PLB_BRAM_IF_CNTRL: The controller for the memory which is attached to the PLB bus.
- PLB_BRAM: Memories for storing data and instructions.
- PLB2OPB_BRIDGE: This bridge connects the PLB and the OPB bus in a master-slave (PLB-OPB) schema. This is a one-way bridge. Therefore, as an OPB-PLB schema is needed an opb2plb_bridge will be required.
- OPB_UART: Universal asynchronous receiver/transmitter, this is attached to the OPB bus and allows the design system to display information on PC.
- DDR_CLOCK_MODULE_REF: The design system requires several different clocks, for the bus, for the CPU, for peripherals, etc.
- PROC_SYS_RESET: The system has different types of resets (i.e. chip reset, system reset, core reset, etc).

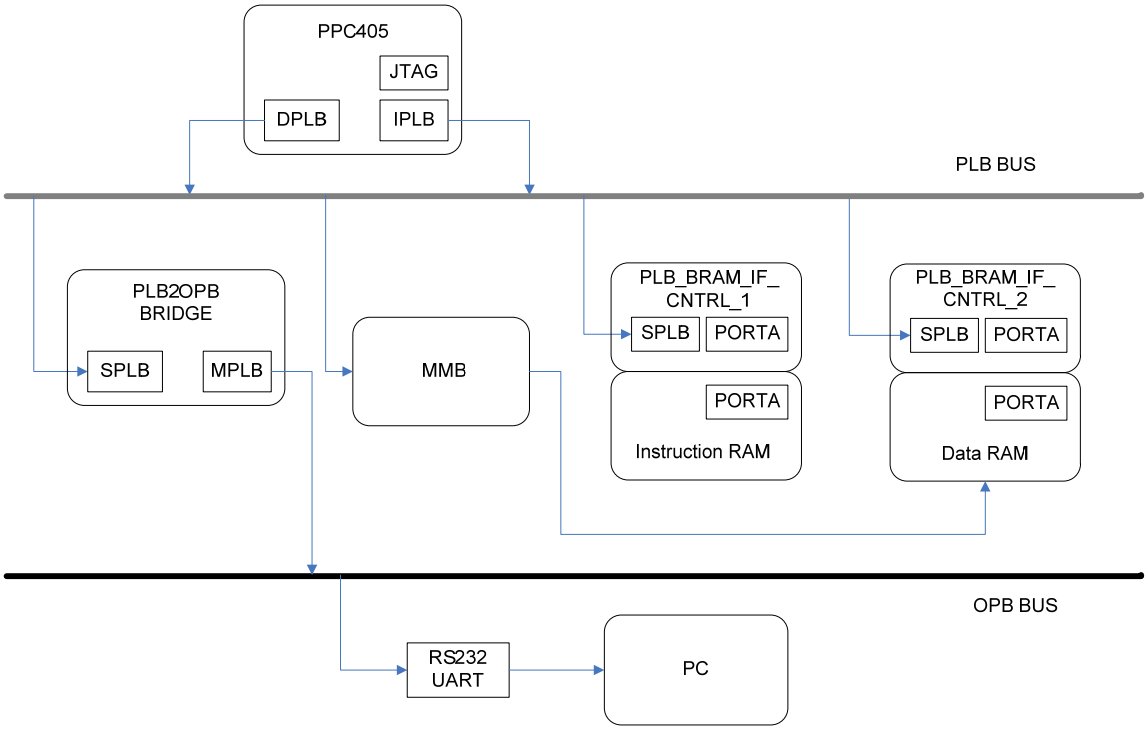

Figure 3.11: EDK design architecture of Jacobi, Gauss-Seidel, and CG.

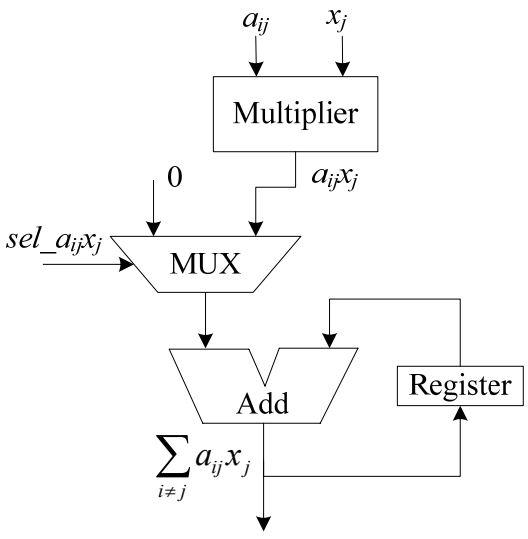

Figure 3.12: Hardware architecture of MMB

As described in Section 3.1.1, the key to a fast implementation of three iterative methods is a fast implementation of matrix multiplication. Hence, Xilinx EDK HW/SW codesign of three iterative methods has matrix multiplications implemented in hardware, and the rest operations implemented in software and stored in instruction RAM. The MMB is able to execute matrix-vector multiplication, vector-vector multiplication and vector-scalar multiplication in one block. Figure 3.12 shows the hardware architecture of MMB including one single precision floating point multiplier, one accumulator containing one single precision floating point adder, and five 32-bit user control registers (not shown in Figure 3.12) which can be accessed by both software and hardware. Functions of five user control registers are listed in Table 3.1. As showed in Figure 3.11, one port of MMB is connected to PLB bus, and another port is connected to data RAM to fetch data. Thus, MMB is only able to fetch one of inputs at a time. Therefore, a global signal, $sel\_a_{ij}x_j$, shown in Figure 3.12 is used to determine when to select $a_{ij}x_j$ so that only $a_{ij}x_j$ or 0 is selected. The MMB computes $a_{ij}\times x_j$, and then adds $a_{ij}x_j$ to accumulate register, where the accumulate register is initialized to zero. The MMB accumulates the products of inputs fed every two cycle. After $2n$ cycles, one element of resulted vectors will be obtained and stored into memory until all of inputs are processed. In order to simplify the design, MMB shares the same clock as PLB bus and is able to run with a max frequency of 41.2MHz due to the limitation of floating point adder, thus 25 MHz is chosen for PLB bus clock frequency selections and frequency of PPC is set to100 MHz.

The write function: MYIP_mWriteReg(BaseAddress, RegOffset, Data) and read function: MYIP_mReadReg(BaseAddress, RegOffset) are used as interfaces between software and hardware, where BaseAddress is the base address of MMB on PLB bus , RegOffset is the register address offset in MMB and Data is 32-bits data written into five user register. The write function writes the base address of matrix A and x, the numbers of rows and columns of matrix A, the base address of matrix result, and the status of flag into five user registers respectively. The read function reads the status of flag for PPC to decide when the software process will start to run.

Table 3.1 Functions of five user control registers for CG

User Control Register	Function
Register_1	Store the base address of matrix A.
Register_2	Store the base address of matrix $\boldsymbol{x}$ (column vector).
Register_3	Store the numbers of rows and columns of matrix $\boldsymbol{A}$ (high 16 bits for row and low 16 bits for column);
Register_4	Store the base address of matrix result;
Register_5	Store the status of flag, i.e. start or end of matrix multiplication.

The overall EDK HW/SW codesign flow of three iterative methods is summarized as follows: PPC runs software implemented operations at the beginning. When matrix multiplication starts, PPC writes five user control registers with base addresses of matrices, numbers of rows and columns, and status of flag. Once flag is set to start, MMB fetches data from data RAM and start multiplication. At the same time, a clock counter begins to count clock cycles and PPC keeps checking the flag register in MMB which shows the status of matrix multiplication. Once matrix multiplications are finished, the flag is set to end, and PPC continues to do other jobs implemented in software until another matrix multiplication occurs. Once all of the evaluation process has been done, PPC reads the register in the clock counter and prints the total time displayed in the number of clock cycles. All three iterative methods are designed in a similar way which use the same MMB, but have different software descriptions corresponding to each method.

So far, FPGA-based hardware architectures, LNS-based hardware designs, and Xilinx EDK HW/SW codesigns of Jacobi, combined Jacobi and Gauss-Seidel and CG have been discussed. Most iterative methods aim at parallel processing and are able to find solution in fewer steps compared to direct methods, but accuracy is not guaranteed for limited iterations. Selected direct methods such as PIE and WZ are able to achieve the required accuracy and solve matrices in parallel compared to GE and LU but mainly target on dense matrices. In Chapter 4, FPGA-based hardware architectures and Xilinx EDK based HW/SW codesign of

WZ factorization is presented followed by the reordering technique dealing with large sparse matrices.

Chapter 4

General Hardware Architectures and HW/SW Codesign of Direct Methods

Since mid-1950s some of direct methods solve matrices by transferring them into block forms [23, 32]. Instead of solving a matrix one element at a time, the matrix is regrouped and solved as sub-blocks. By doing this, matrix can be factorized and solved from two sides simultaneously. WZ factorization and PIE are two of these methods which are more suitable for parallel computation. The goal of this chapter is to provide FPGA-based hardware implementations of WZ factorization. Single unit and scalable hardware architectures of WZ factorization are proposed and analyzed under different constraints. Xilinx EDK HW/SW codesign of WZ factorization is presented targeting on hardware implementation of the matrix update, followed by the reordering technique extended to WZ factorization for solving large sparse systems.

4.1 Alternative Methods of WZ factorization and PIE

As described in chapter 2, Doolittle's method and Crout's method are two alternatives to LU factorization. Two methods are slightly different, where the Doolittle's method returns a unit lower triangular matrix and an upper triangular matrix, while the Crout's method returns a lower triangular matrix and a unit upper triangular matrix. Two methods decompose matrices in a similar process but with different sequence. Similarly, ZW factorization and X factorization can be used as alternatives of WZ factorization and PIE respectively, where matrices are factorized in a similar way but from different directions.

4.1.1 ZW Factorization

An alternative of WZ factorization is to decompose matrix in a reverse way into ZW form, called ZW factorization. Let $A = Z'W'$, where

$$Z' = \begin{bmatrix} 1 & z_{12} & z_{13} & .. & z_{1,(n+1)/2} & .. & z_{1,n-2} & z_{1,n-1} & 0 \\ & 1 & z_{23} & & .. & & z_{2,n-2} & 0 & \\ & & 1 & .. & z_{(n-1)/2,(n+1)/2} & .. & 0 & & \\ & 0 & . & & 1 & & . & 0 & \\ & & 0 & .. & z_{(n+3)/2,(n+1)/2} & .. & 1 & & \\ & 0 & z_{n-1,3} & & .. & & z_{n-1,n-2} & 1 & \\ 0 & z_{n,2} & -z_{n,3} & .. & z_{n,(n+1)/2} & .. & z_{n,n-2} & z_{n,n-1} & 1 \end{bmatrix} \tag{4.1}$$

and

$$W' = \begin{bmatrix} w_{11} & & & 0 & & & w_{1,n} \\ w_{21} & w_{22} & & & & w_{2,n-1} & w_{2,n} \\ .. & .. & .. & & .. & .. & .. \\ w_{(n+1)/2,1} & w_{(n+1)/2,2} & .. & w_{(n+1)/2,(n+1)/2} & .. & w_{(n+1)/2,n-1} & w_{(n+1)/2,n} \\ .. & .. & .. & & .. & .. & .. \\ w_{n-1,1} & w_{n-1,2} & & 0 & & w_{n-1,n-1} & w_{n-1,n} \\ w_{n,1} & & & & & & w_{n,n} \end{bmatrix} \tag{4.2}$$

ZW factorization solves two columns and two rows from the middle of matrices Z' and W' toward outside instead of solving them from outsides of matrices toward inside. The evaluation procedure is distinguished into two cases.

Case1: If n is odd, at the first evaluation stage, the elements of $(n+1)/2_{th}$ row of matrix W' are obtained by:

$$w_{(n+1)/2,j} = a_{(n+1)/2,j} \quad \text{for } j = 1, 2, ..., n\text{-}1, n. \tag{4.3}$$

The elements of $(n+1)/2_{th}$ column of matrix Z' are evaluated by

$$z_{i,(n+1)/2} = \frac{a_{i,(n+1)/2}}{w_{(n+1)/2,(n+1)/2}}, \tag{4.4}$$

where $i = 1,\ 2,...,\ (n-1)/2,\ (n+3)/2,...,\ n\text{-}1,\ n$ and $i \neq (n+1)/2$. The remaining matrix is then updated by

$$a_{ij} = a_{ij} - z_{i,(n+1)/2} w_{(n+1)/2,j}, \tag{4.5}$$

where $i, j = 1, 2, 3,..., n$ and $i, j \neq (n+1)/2$. At the second evaluation stage, the elements of $(n\text{-}1)/2_{th}$ and $(n\text{+}3)/2_{th}$ rows of matrix W' are obtained by:

$$w_{(n-1)/2,j} = a_{(n-1)/2,j} \text{ and } w_{(n+3)/2,j} = a_{(n+3)/2,j}, \tag{4.6}$$

where $j = 1,\ 2,...,\ n\text{-}1,\ n$ and $j \neq (n+1)/2$. The elements of $(n\text{-}1)/2_{th}$ and $(n\text{+}3)/2_{th}$ columns of matrix Z' are evaluated by solving $(n\text{-}3)$ sets of 2×2 linear equations given by

$$z_{i,(n-1)/2} w_{(n-1)/2,(n-1)/2} + z_{i,(n+3)/2} w_{(n+3)/2,(n-1)/2} = a_{i,(n-1)/2}$$

and (4.7)

$$z_{i,(n-1)/2} w_{(n-1)/2,(n+3)/2} + z_{i,(n+3)/2} w_{(n+3)/2,(n+5)/2} = a_{i,(n+3)/2},$$

where $i = 1,\ 2,...,\ n\text{-}1,\ n$ and $i \neq (n\text{-}1)/2,\ (n+1)/2$ and $(n+3)/2$. The remaining matrix is updated by

$$a_{ij} = a_{ij} - z_{i,(n-1)/2} w_{(n-1)/2,j} - z_{i,(n+3)/2} w_{(n+3)/2,j}, \tag{4.8}$$

where $i, j = 1, 2,..., n\text{-}1, n$ and $i, j \neq (n-1)/2,\ (n+1)/2$ and $(n+3)/2$. For following stages, this process repeats until the matrix Z' and W' are found in the forms of Equations 4.1 and 4.2.

Case 2: If n is even, at first evaluation stage, two rows of W' and two columns of Z' (i.e. $n/2_{th}$ and $(n\text{+}2)/2_{th}$ rows, and columns) are obtained by using Equations 4.6 and 4.7. This process repeats for the following stages until the elements of first and last rows, and columns of matrix W' and Z' are found.

During the solution process, $Ax = b$ can be rewritten as $(Z'W')x = b$. Two related and simpler linear systems of forms $Z'y = b$ and $W'x = y$ are required to be solved. Vector y is obtained by solving $Z'y = b$ and final solution x is obtained by solving $W'x = y$. The

solution process begins by solving elements of y in pairs from the middle of vector y. For odd case, the process starts with $y_{(n\text{-}1)/2}$ and $y_{(n+3)/2}$. In general, at i_{th} stage,

$$y_i = b_i \text{ and } y_{n\text{-}i+1} = b_{n\text{-}i+1}, \tag{4.9}$$

where $i = 1,\ 2,...,\ n\text{-}1, n$ and $i \neq (n+1)/2$. Vector b is then updated by Equation 4.10.

$$b_j^{'} = b_j - z_{ji} y_i - z_{j,n-i+1} y_{n-i+1}, \tag{4.10}$$

where $j = 1,\ 2,...,\ i-2,\ i-1$ and $j = n-i+2,\ n-i+3,...,\ n-1,\ n$. The final solution x is solved from the top and bottom of matrix W', which is evaluated in pairs by solving (n-1)/2 sets of 2×2 linear equations ending with $x_{(n\text{-}1)/2}$ and $x_{(n+3)/2}$. The last equation to be solved is

$$x_{(n+1)/2} = \frac{y_{(n+1)/2}}{w_{(n+1)/2,(n+1)/2}} \tag{4.11}$$

Even case is similar as odd case, where x is evaluated by solving n/2 sets of 2×2 linear equations ending with $x_{n/2}$ and $x_{(n+2)/2.}$ An example of ZW is given in Appendix A.

4.1.2 X Factorization

Another modification scheme of PIE method is named as X factorization. The final matrix X is showed as follows.

$$X = \begin{bmatrix} x_{11} & & & 0 & & & x_{1n} \\ & x_{22} & & & & x_{2,n-1} & \\ & & x_{33} & & x_{3,n-2} & & \\ 0 & & & \cdots & & & 0 \\ & & x_{n-2,3} & & x_{n-2,n-2} & & \\ & x_{n-1,2} & & & & x_{n-1,n-1} & \\ x_{n1} & & & 0 & & & x_{nn} \end{bmatrix} \tag{4.12}$$

Matrix X is achieved by taking product of matrix Z' in the form of Equation 4.13 with matrix Z in the form of 2.14, where matrix Z is obtained by PIE method.

$$Z' = \begin{bmatrix} 1 & -z_{12} & -z_{1,3} & \dots & -z_{1,n-2} & -z_{1,n-1} & 0 \\ 0 & 1 & -z_{2,3} & \dots & -z_{2,n-2} & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & 1 & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & -z_{n-1,3} & \dots & -z_{n-1,n-2} & 1 & 0 \\ 0 & -z_{n2} & -z_{n3} & \dots & -z_{n,n-2} & -z_{n,n-1} & 1 \end{bmatrix} \tag{4.13}$$

By transforming matrix Z into X form, final solution x can be obtained by solving multiple sets of 2×2 equations simultaneously. The cross diagonal elements of matrix X are same as those in matrix Z. By equalizing $Z'Z$ with X, matrix Z' is obtained and used to update vector b', where vector b' is the updated vector b from PIE method. The relationship between PIE and X factorization is showed as follows: First, multiply both sides of Equation 2.1 with matrix W to obtain Equation 4.14.

$$WAx = Wb \tag{4.14}$$

As described in PIE scheme, $WA = Z$, substitute this equation into Equation 4.14 to obtain Equation 4.15

$$Zx = b' \tag{4.15}$$

where $b' = Wb$. Then multiply both sides of Equation 4.15 with Z' to obtain the following equation.

$$Z'Zx = Z'b' \tag{4.16}$$

Substitute $Z'Z = X$ into Equation 4.16 to obtain Equation 4.17

$$Xx = b'' \tag{4.17}$$

where $b'' = Z'b'$. $Z'Z$ can be written in Equation 4.18.

$$Z'Z = \begin{bmatrix} 1 & -z'_{12} & -z'_{1,3} & \dots & -z'_{1,n-2} & -z'_{1,n-1} & 0 \\ 0 & 1 & -z'_{2,3} & \dots & -Z'_{2,n-2} & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & 1 & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 \\ 0 & 0 & -z'_{n-1,3} & \dots & -z'_{n-1,n-2} & 1 & 0 \\ 0 & -z'_{n2} & -z'_{n3} & \dots & -z'_{n,n-2} & -z'_{n,n-1} & 1 \end{bmatrix} \tag{4.18}$$

$$* \begin{bmatrix} z_{11} & z_{12} & . & . & . & z_{1,n-1} & z_{1,n} \\ & z_{22} & . & . & . & z_{2,n-1} & \\ & & & . & & & \\ 0 & & & z_{(n+1)/2,(n+1)/1} & & 0 & \\ & & & . & & & \\ & z_{n-1,2} & & . & & z_{n-1,n-1} & \\ z_{n,1} & z_{n,2} & . & . & . & z_{n,n-1} & z_{n,n} \end{bmatrix}$$

Considering the case when n is odd, at the first evaluation stage, the elements of 2_{nd} and $(n-1)_{th}$ columns of matrix Z' are obtained by solving two sets of 2×2 equations given by

$$\begin{aligned} z'_{12}\, z_{22} + z'_{1,n-1}\, z_{n-1,2} &= z_{12} \\ z'_{12}\, z_{2,n-1} + z'_{1,n-1}\, z_{n-1,n-1} &= z_{1,n-1} \end{aligned} \tag{4.19}$$

and

$$\begin{aligned} z'_{n,2}\, z_{22} + z'_{n,n-1}\, z_{n-1,2} &= z_{n,2} \\ z'_{n,2}\, z_{2,n-1} + z'_{n,n-1}\, z_{n-1,n-1} &= z_{n,n-1} \end{aligned} . \tag{4.19}$$

The remaining elements of Z are updated using Z' values from the previous step. For following evaluation stages, this procedure repeats until the elements of middle column of matrix Z' are found. An example of X factorization is given in Appendix A.

4.2 FPGA-based Hardware Architectures of WZ factorization

As described in chapter 2, WZ factorization solves matrices by transferring them into block forms, where the matrix is regrouped and solved as sub-blocks. In FPGA-based hardware design of WZ factorization, pairs of elements of matrix W can be solved in serial by single

unit architecture and multiple pairs of matrix W can be solved in parallel by a scalable architecture.

4.2.1 Single Unit Architecture

Figure 4.1 shows the single unit hardware architecture of WZ factorization. This architecture consists of a 2×2 solver and one update block which are used to solve a pair of W values and update one row of the remaining matrix A_{n-2i} respectively. The 2×2_solver contains three Wsolvers, where sub/adder functioned as subtractor in these Wsolvers. The update block contains several Aupdate units. Each Aupdate unit contains one Wsolver and one sub/adder functioned as an adder. Each Aupdate unit is used to update certain elements of the A_{n-2i} based on the position of Aupdate unit. Hardware architectures of Wsolver and Aupdate unit are shown in Figure 4.2, where i represents each distinct evaluation stage for i =1, 2,…,(n+1)/2, j represents elements of the first and last columns of matrix W for j =i+1, i+2,…,n-i, and k represents row elements of the remaining matrix A_{n-2i} for k =i+1, i =2,…,n-i.

In this architecture, the 2×2_solver applies Cramer's rule under the non-singularity constraint imposed for their determinants. Three Wsolvers compute a pair of W values simultaneously and fed them into the update block. Ideally, the update block consists of (n-2i) Aupdate units. It is unlikely to implement this ideal case in reality, since the number of Aupdate units will increase significantly as matrix size increases. Therefore, the number of Aupdate units should be determined according to the available hardware resource. Assume two ram modules are allocated to 2×2_solver and Aupdate respectively. During the initialization phase, matrix A can be stored into matrix Z to save memory and accelerate the calculation process. By doing so, the elements of first and last rows of matrix Z are not necessary to be evaluated and the process of the update of the A_{n-2i} is carried out directly in matrix Z. There are six different inputs for the 2×2_solver, and five inputs for each Aupdate unit where two of them are W values coming from the 2×2_solver. The modification block could start loading data once the 2×2_solver finishes loading data from memory. Only one Wsolver's processing time is counted since three Wsolvers are processed simultaneously. Assume each Wsolver and divider take α and β cycles to process respectively. Thus the 2×2

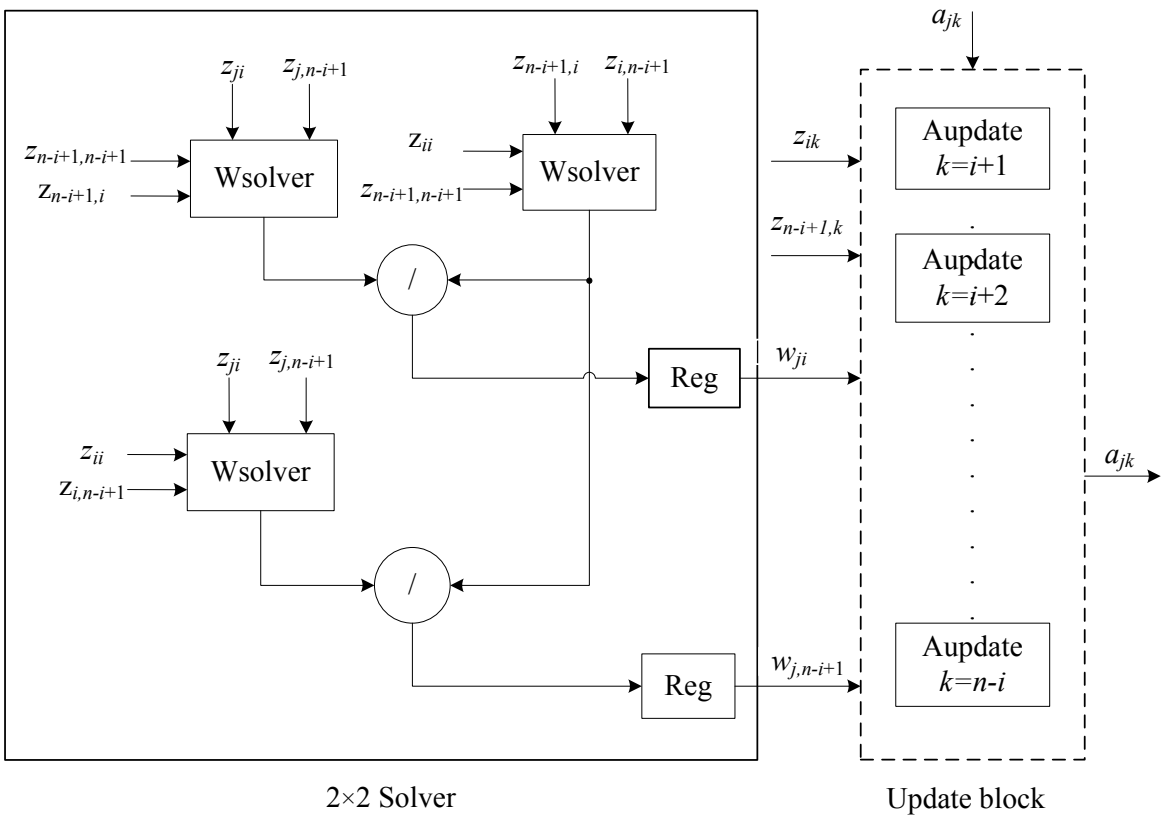

Figure 4.1: Block diagram of a single unit for WZ factorization method.

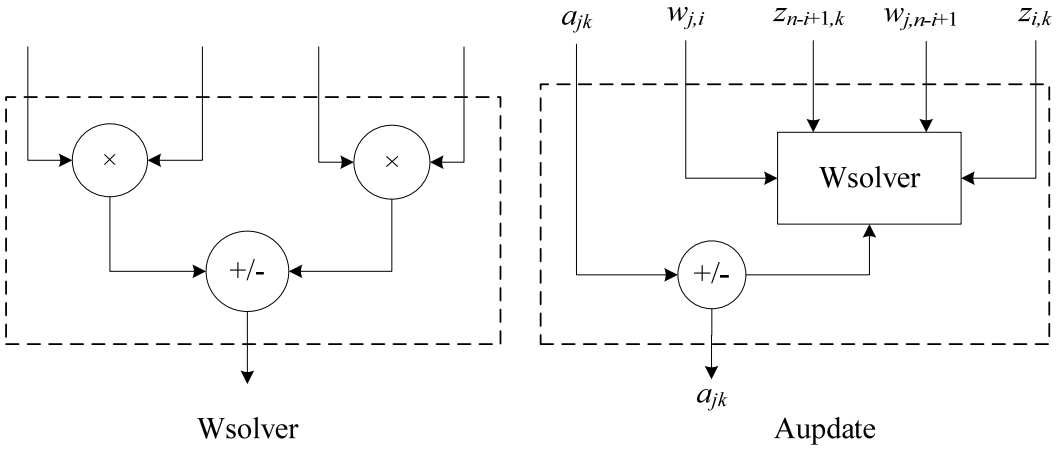

Figure 4.2: Block diagrams of Wsolver and Aupdate.

_solver requires (α+β) cycles to process. Memory is equally divided into λ portions satisfying the bandwidth required for one 2×2_solver. The loading cycle for p Aupdate units is $3p/\lambda$. By equalizing these two items, p can be evaluated as

$$(\alpha+\beta) = {}^{3p}\!/_{\lambda} \Rightarrow p = {}^{\lambda(\alpha+\beta)}\!/_{3}. \tag{4.20}$$

From the data distribution aspect, four of inputs data of the 2×2_solver are located at four corners of matrix Z (e.g. z_{11}, z_{n1}, z_{n1}, z_{nn} for first stage). The other two are located at the first and last columns and same row as W values located. For example, if w_{ji} and $w_{j,\ n-i+1}$ are evaluated by 2×2_solver, the rest two inputs are located at j_{th} row of matrix Z (e.g. z_{ji} and z_{jn} for the first stage). Inputs data to each Aupdate are located at the same column of the element which required to be updated. For example, in order to update a_{ik}, the inputs data are retrieved from k_{th} column of matrix Z (e.g. z_{1k} and z_{nk} for first stage). Therefore, memory can be divided into 4 portions according to data distribution and each portion is able to store a quarter of matrix Z.

4.2.2 Scalable Architecture

To achieve complete parallel process, n-2 WZ_solvers can be parallized in WZ hardware architecture. Each WZ_solver is responsible for evaluating a pair of W values and updating elements of one row of the remaining matrix A_{n-2i}. To extend the parallelism, n-2 Aupdate units can be parallized in each WZ_solver for the first iteration. For the following iterations, both the numbers of WZ_solvers and Aupdate units will be decreased by 2, thus some of the hardware resource are wasted at later stages. Considering the available hardware resource, several possible cases are proposed as follows, where single unit is denoted as WZ_solver here.

1. 1 WZ_solver with 1 2×2_solver and 1 Aupdate unit. Pairs of W values are solved in serial by 2×2_solver and elements of one row of the remaining matrix A_{n-2} are also updated in serial by Aupdate unit.

2. 1 WZ_solver with 1 2×2_solver and k (k<n-2) Aupdate units. A subset of one row of A_{n-2i} is updated in parallel at a time which requires $\lceil (n-2i)/k \rceil$ number of times to update one row of the remaining matrix A_{n-2i}. But pairs of W values are solved in serial by 2×2_solver.

3. l (l<n–2) WZ_solvers with 1 2×2_solver and 1 Aupdate in each WZ_solver. 1 WZ_solver is corresponding to solve one pair of W values and update one row of the remaining matrix A_{n-2i}. Inside each WZ_solver, the elements of one row of A_{n-2i} are updated in serial, but a subset of W are solved in parallel at a time which requires $\lceil (n-2i)/l \rceil$ number of times to obtain two columns of matrix W.

4. p (p<n-2) WZ_solvers with 1 2×2_solver and q (q<n-2) Aupdate units in each WZ_solver. A subset of one row of A_{n-2i} is updated in parallel inside each WZ_solver, which requires $\lceil (n-2i)/q \rceil$ number of times to process. A subset of W values are also solved in parallel at a time among p WZ_solvers which requires $\lceil (n-2i)/p \rceil$ number of times to obtain two columns of matrix W.

Case 4 is considered here for scalable hardware architecture. In order to evenly distribute the data, assume the memory allocated to 2×2 solver is divided into η portions and the memory allocated to the update block is divided in to δ portions. Assume each Wsolver and divider take α and β cycles to process respectively. As showed in Figure 4.3, first WZ_solver has six inputs data for 2×2_solver and three inputs data for each Aupdate unit. Each extra WZ_solver adds two more inputs data for 2×2_solver and one more input data for each Aupdate unit. The numbers shown in Figure 4.3 represent the numbers of inputs data. The numbers of WZ_solvers and Aupdate units (i.e. p and q) are obtained as follows. By equalizing the number of inputs data of 2×2_solvers with memory portions η, p can be evaluated as:

$$(6+2(p-1)) = \eta \;\Rightarrow\; p = {(\eta - 4)}/{2} \tag{4.21}$$

where $6+2(p-1)$ is the number of inputs data of 2×2_solver.

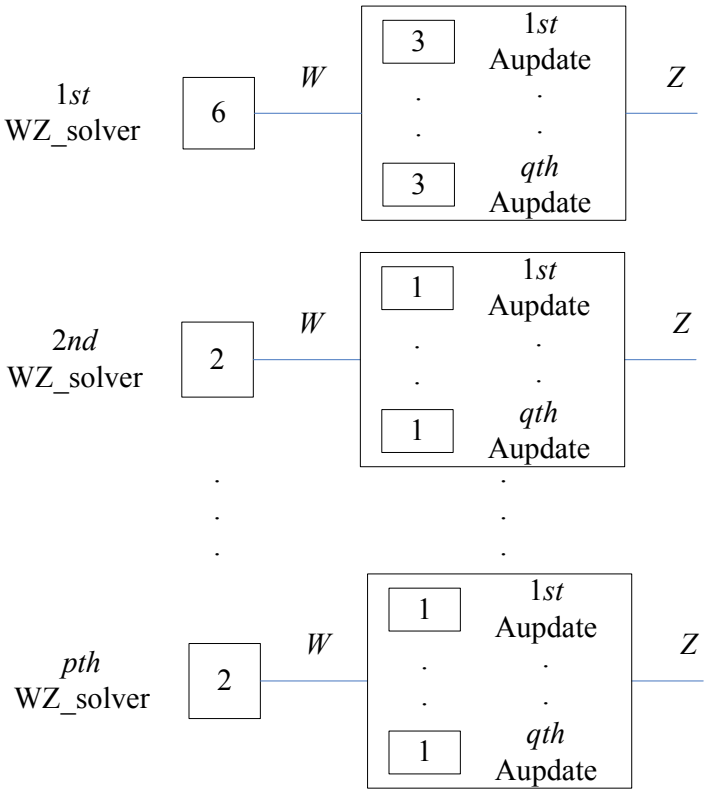

Figure 4.3: Scalable architecture of WZ factorization(case 4).

By equalizing the total number of inputs data of Aupdate units with the loading cycle for q Aupdate units, q can be evaluated as:

$$(\alpha+\beta)=(p-1+3)q/\delta \;\Rightarrow\; q=(\alpha+\beta)\delta/(p+2) \tag{4.22}$$

where $(p-1+3)q$ is the number of inputs data of Aupdate units. Substitute Equation 4.21 into Equation 4.22, q is obtained by

$$q=2(\alpha+\beta)\delta/\eta \tag{4.23}$$

In this architecture, both parts of memory are initialized with matrix Z and are divided into η and δ portions respectively, which make data distribution more dynamical according to the requirements. The evaluation of next pairs of W values requires updating the remaining matrix A_{n-2i} in the previous iteration. This dependence hinders the parallelization of the WZ

factorization. This drawback can be improved by adjusting data input sequences. During the update process, control signals are used to pick the Z values required to be updated first. In this case, the elements of first and last columns in A_{n-2i} should be updated first since those values are required by 2×2 solvers for next iteration.

4.3 Xilinx EDK HW/SW Codesign of WZ Factorization

As described in chapter 3, Xilinx EDK tools for HW/SW codesign enable the integration of both hardware and software components of an embedded system. Xilinx EDK HW/SW codesign architecture for WZ factorization is shown in Figure 4.4, which has the same architecture as HW/SW codesign of CG except the hardware implemented block, i.e. the update of one row of the remaining matrix A_{n-2i} instead of MMB is implemented in hardware. The Update block shown in Figure 4.5 consists of one single precision floating point multiplier, one single precision floating point adder/subtractor, and three 32-bit user control registers (not shown in Figure 4.5) which can be accessed by both hardware and software. Functions of three user control registers are listed in Table 4.1. As shown in Figure 4.4, Update block is only able to fetch one of inputs at a time. Therefore, a global signal, *sel_wz*, shown in Figure 4.5 is used to determine when to select *wz* so that only *wz* or 0 is selected. The Update block computes *wz*, and then subtracts *wz* from a_{ij} and stores the result, a'_{ij}, into accumulate register. The second global signal, *sel_*a_{ij}, is used to determine when to select the accumulated result a'_{ij}, and subtracts second *wz* from a'_{ij}.

Table 4.2 Functions of three user control registers for WZ

User Control Register	**Function**
Register_1	Store the base address of the remaining matrix A_{n-2i}, which is same as the base address of updated results
Register_2	Store the number of columns of the remaining matrix A_{n-2i}.
Register_3	Store the status of flag, i.e. start or end of matrix multiplication.

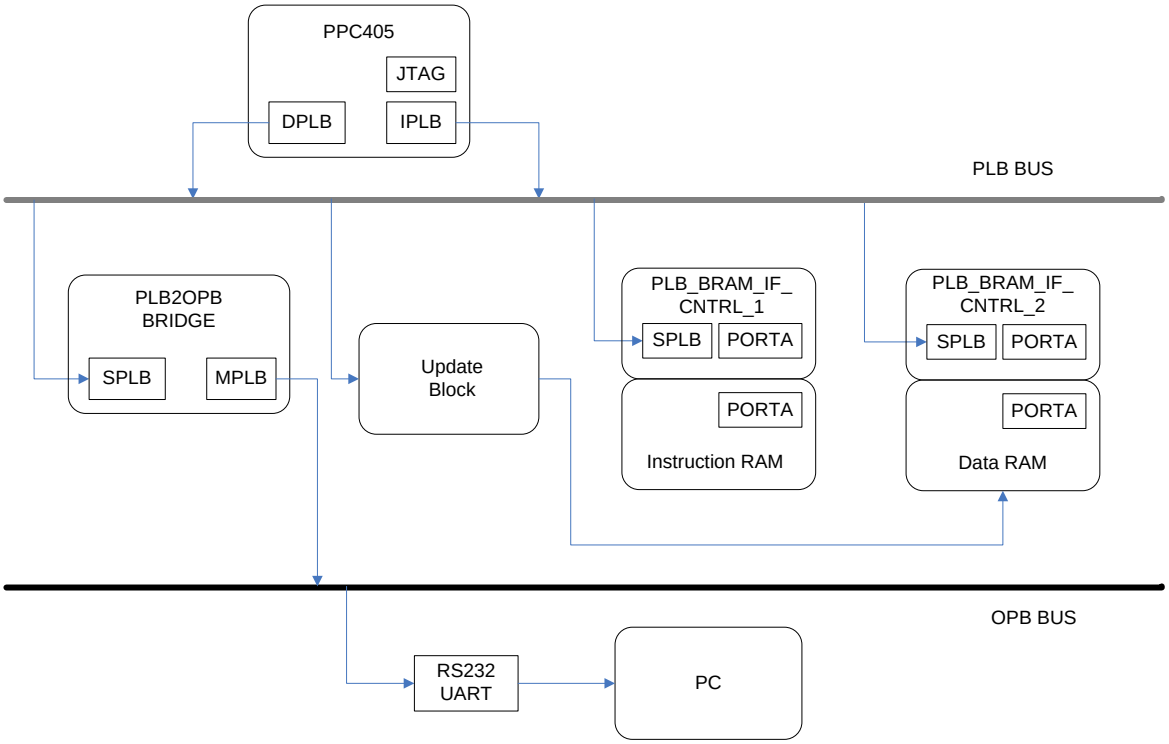

Figure 4.4: EDK design architecture of WZ factorization

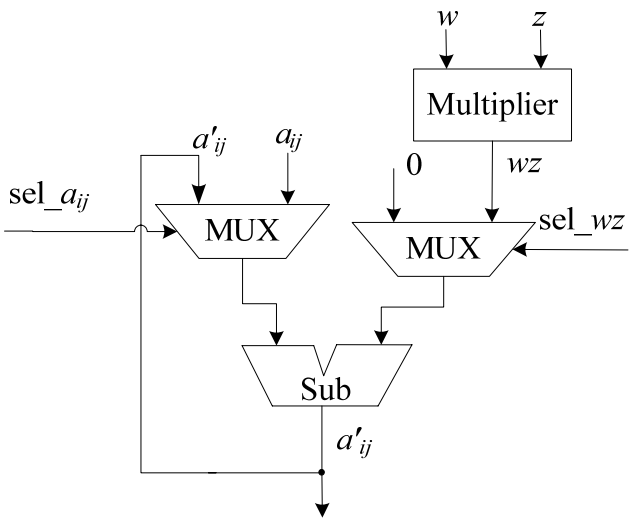

Figure 4.5: Hardware architecture of Update block

The write function: MYIP_mWriteReg(BaseAddress, RegOffset, Data) and read function: MYIP_mReadReg(BaseAddress, RegOffset) are used as interfaces to software, where BaseAddress is the base address of the Update block on PLB bus, RegOffset is the user register address offset in the Update block and Data is 32-bits data written into three user registers. Write function writes the base address of the remaining matrix A_{n-2i}, the number of columns of the remaining matrix A_{n-2i}, and the status of flag into three user registers respectively. Read function reads the status of flag for PPC to decide when the software implemented processes will start to run.

The overall EDK HW/SW codesign flow of WZ factorization is summarized as follows: PPC runs software implemented operations at the beginning. When the row update starts, PPC writes three user control registers with the base address of A_{n-2i}, the number of columns of A_{n-2i}, and the status of flag. Once flag is set to start, Update block fetches data from data RAM and start update. At the same time, a clock counter begins to count clock cycles and PPC keeps checking the flag register in update block. Once the update is finished, flag will be set to end, and PPC continues to do other jobs implemented in software until another row update occurs. Once all of the evaluation process has been done, PPC reads the register in the clock counter and prints the total time displayed in clock cycles.

4.4 Reordering Techniques for Sparse Matrix

If a matrix is dense, the best choice is probably to factor the matrix and solve the equation by back substitution. The time spent on factoring a dense matrix is roughly equivalent to the time spent on solving the matrix iteratively. But for applications like power network and circuit simulation, the larger the network is, the more sparse the matrix. For example, the non-zero elements in a 3000×3000 power network matrix occupy only about two percent of total elements. Even though a sparse matrix offers the advantage of reduced storage space for data storage, factoring sparse matrix produces more nonzeros than matrix itself, resulting in several fill-ins, which might be impossible to implement due to limited memory, and will be

time consuming as well. Iterative methods are memory-efficient and run quickly with sparse matrices, but direct methods are still preferred choices if more accurate results are desired.

A dynamic data structure [53] is discussed as follows to take care of the fill-ins during factorizations of direct methods, where a sparse matrix can be reordered into bordered-diagonal-block (BDB) form shown in Figure 7 to reduce the number of fill-ins. The most widely used reordering techniques are minimum degree and minimum fill-in [54]. The idea is to generate a permutation of the original matrix so that the permuted matrix results in a stable solution that also increases parallelism. As showed in Figure 7, A_{ij}'s are matrix blocks; A_{ii}'s are referred to as the diagonal blocks; A_{in} and A_{nj} are called right border blocks and bottom border blocks, respectively, where $i, j \in [1, n-1]$ and A_{nn} is known as the last block. The blocks A_{ii}, A_{in}, and A_{ni} are said to form a 3-block group [55], where $i \in [1, n-1]$ and $n \le N$. Since all non-border off diagonal blocks contain only 0's, there will be no fill-ins in these blocks during factorization and the resulting factorized matrix keeps the same BDB structure. In this BDB form, there is no data dependence among the factorization of the 3-block groups until the last block. Hence, the factorization of the 3-block groups can be carried out independently from each other and no inter-processor communication is required during this procedure. In order to factor the last block A_{nn}, pairs of blocks from right border and bottom border are multiplied in parallel to produce $A^*_{nj} = A_{nj} A_{jn}$, where $j \in [1, n-1]$. The resulting products can be stored in the bottom blocks and the summation of these products is required to factor the last block. The summation of these products is carried out along a binary tree in parallel and the results are sent to the processor assigned to the last diagonal block. Upon above description, the BDB matrix algorithm shows distinct advantages for parallel implementation.

Same technique can be applied to WZ factorization shown in Figure 8. This BDB-based WZ factorization involves four steps: (1) WZ factorization of the independent blocks. (2) Multiplication of the right and bottom border blocks to generate the partial sums. (3) The accumulation of the partial results for the last diagonal block. (4) WZ factorization of the last diagonal block using the accumulated partial results from the previous steps.

$$\begin{bmatrix} A_{11} & 0 & \dots & 0 & A_{1n} \\ 0 & A_{22} & \dots & 0 & A_{2n} \\ .. & 0 & \dots & 0 & .. \\ 0 & 0 & \dots & A_{n-1,n-1} & A_{n-1,n} \\ A_{n1} & A_{n2} & \dots & A_{n,n-1} & A_{n,n} \end{bmatrix}$$

Figure 7: Sparse matrix in BDB form

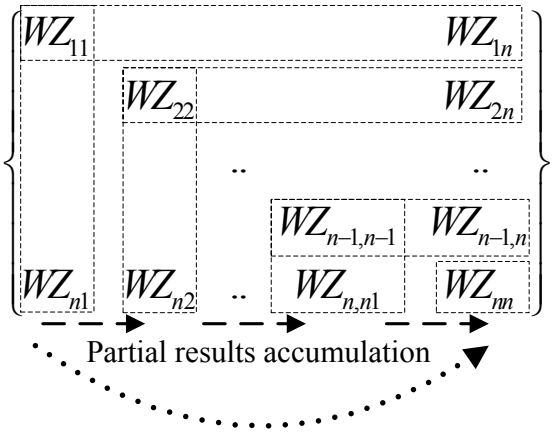

Figure 8: Parallel WZ factorization of a sparse BDB matrix

This chapter discussed FPGA-based hardware architectures and Xilinx EDK HW/SW codesign of WZ factorization. So far, FPGA-based general hardware architectures of three iterative methods and WZ factorization have been discussed, and Xilinx EDK HW/SW codesign of three iterative methods and WZ factorization have been done. Performance analysis and comparison of each method are showed in next chapter.

Chapter 5

Performance Analysis

After realization of FPGA-based general hardware architectures, LNS based hardware designs, and Xilinx EDK HW/SW codesigns of selected matrix solving methods, performance analysis of each method are discussed under their own category (i.e., iterative methods or direct methods) based on the results of Matlab simulations and Xilinx EDK HW/SW codesigns. Convergence analysis of LNS-based Jacobi processor is also given to show that how the simplified error correction circuit for logarithm/antilogarithm conversion is related to the convergence of Jacobi method.

5.1 Performance Analysis of Iterative Methods

5.1.1 Matlab Comparison of Jacobi, Gauss-Seidel and Conjugate Gradient

Single-processor computer based Matlab simulations were performed to evaluate three iterative methods. This computer has IBM IntelliStation Z Pro with a 3.6 GHz Intel Xeon processor, 2MB L2 cache and 2.75 GB of system memory. Symmetric, positive-definite linear systems are used for testing. Those matrices are generated by Poisson equation [56], which is a partial differential equation with broad utility in electrostatics, mechanical engineering and theoretical physics. The Matlab simulation results for the three methods are showed in Figures 5.1, 5.2 and 5.3 with different specifications of matrices and tolerance. Figure 5.1 shows that given a specific matrix, CG always takes less numbers of iterations to converge for any tolerance values. The result also verifies that CG converges in at most n steps, where n is the size of matrix. Figure 5.2 shows that CG always converges at a faster

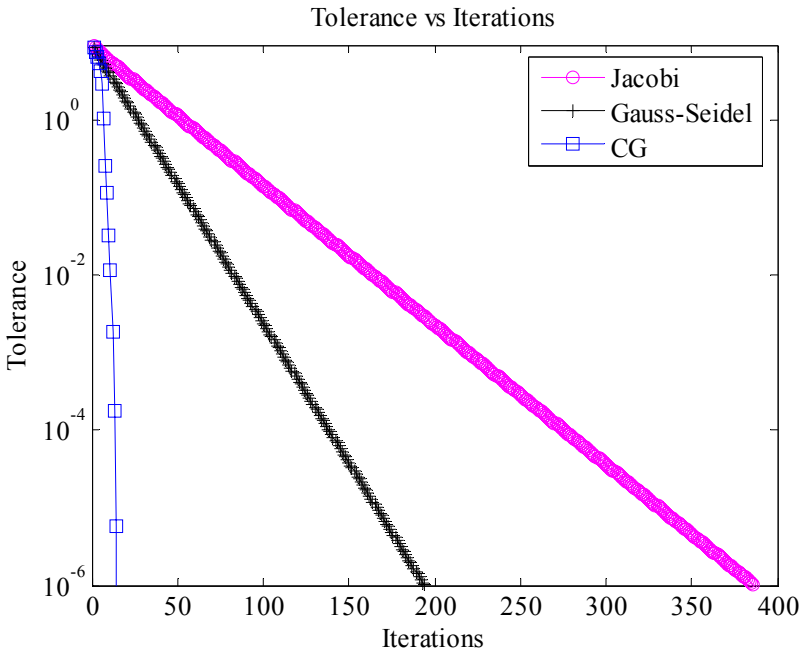

Figure 5.1: Number of iterations required for solving specific size of linear systems for different tolerance values according to Jacobi, GS and CG methods.

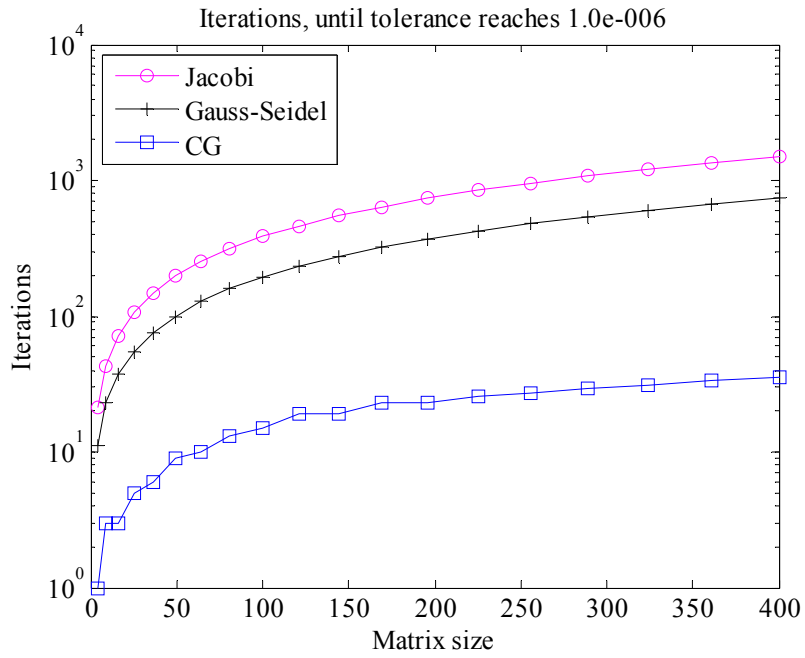

Figure 5.2: Number of iterations required for solving different size of linear systems for specified tolerance value according to Jacobi, GS and CG methods.

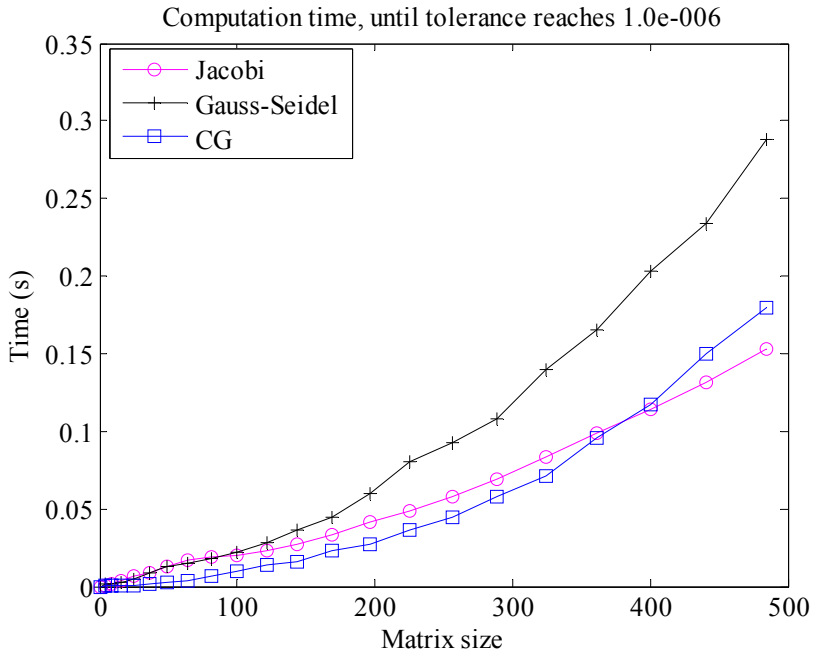

Figure 5.3: Total computation time required for solving different size of linear systems for specified tolerance values according to Jacobi, GS and CG methods.

rate for a given tolerance and different size of matrix. The result also shows that Gauss-Seidel takes about half numbers of iterations of Jacobi method taken to converge. Therefore, in term of iteration, CG is a better choice to solve given linear systems. However, the total computation times of the three methods indicate different results. A Matlab function named as *tic-toc* is used to calculate the average computation time of the three iterative methods for different matrices in terms of CPU time. Figure 5.3 shows that CG takes less computation time than Jacobi at the beginning, but requires more time as the size of matrix grows over 400. This result is caused by the increasing number of matrix-vector multiplications as described in Chapter 2, which is the dominant factor of the evaluation processes of three methods as matrix increases, and matrix multiplication is more dominant compared with Jacobi and Gauss-Seidel method. Therefore, this part has been implemented in hardware to improve overall performance.

5.1.2 Convergence Analysis of LNS-based Jacobi Processor

Jacobi method will always converge if A is diagonally dominant [57], i.e.,

$$\sum_{j \neq i} \left| a_{ij} \right| < a_{ii} \tag{5.1}$$

Jacobi sometimes converges even if this condition is not satisfied. It is necessary, however, that the diagonal terms in the matrix are greater (in magnitude) than the other terms. When LNS is used, logarithm and antilogarithm conversions introduce additional errors. It is important to know whether these additional errors affect the convergence of Jacobi method. Matlab simulations were carried out to study how error introduced by logarithm conversion in a LNS-based Jacobi processor could affect the convergence. The matrices generated by Matlab that satisfy Equation 5.1 are chosen, i.e. the matrices are all convergent according to Jacobi method. Firstly, non-diagonal values are randomly generated within a range of [10 20]; then diagonal values are chosen according Equation 5.1. For performance comparison, these diagonal values are scaled up to 10 times and 100 times of the original one. Equations of linear systems based on the chosen matrices are then solved by using Jacobi method with and without using LNS.

Figure 5.4 shows the results of Matlab simulations of matrices with and without using LNS. Two types of error correction approaches are used with LNS; one is 6-region error correction according to [21] and the other is 8-region error correction according to [58]. Figure 5.5 shows similar Matlab simulation results with 10-times larger diagonal values than the ones used in Figure 5.4. It should be noted that LNS with 6 regions error correction has accuracy between 10^{-2} and 10^{-3} while LNS with 8 regions error correction has accuracy between around 10^{-3}. By comparing the numbers of iterations in Figures 5.4, 5.5 and 5.6, it shows that larger diagonal values result in less number of iterations to obtain solutions, which is consistent with Equation 5.1. Since data less than 10^{-8} can not be represented in binary format with limited range of the fraction part of logarithm, for some cases, LNS-based Jacobi processor is not able to converge for tolerance less than 10^{-8}. The limited range of the fraction part also causes degraded accuracy. In order to improve the convergence for smaller

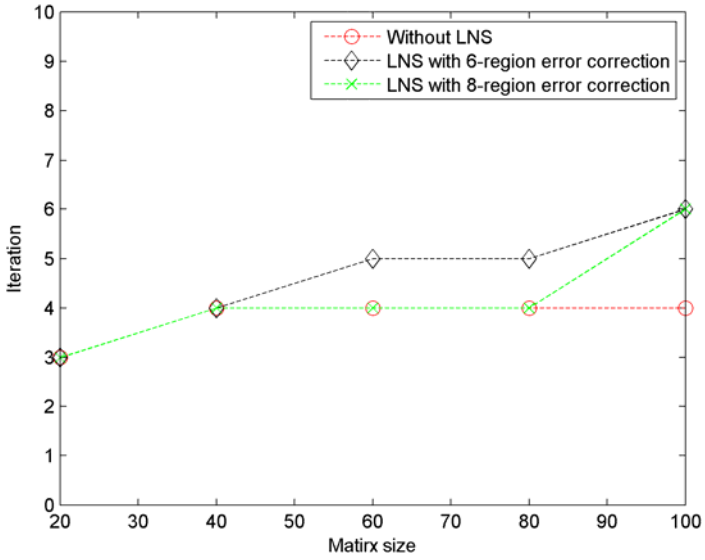

Figure 5.4 Number of iterations required for solving different size of linear systems for different diagonal values according to Jacobi method with and without using LNS. Scale factors for diagonal values are ×1.

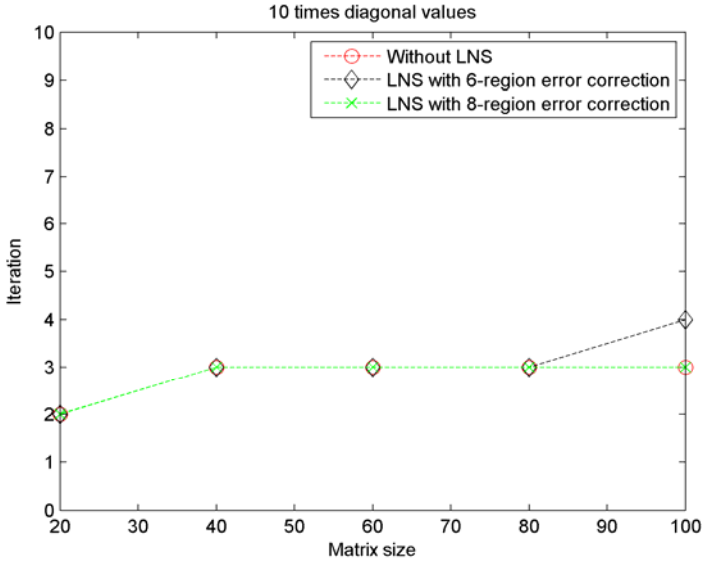

Figure 5.5 Number of iterations required for solving different size of linear systems for different diagonal values according to Jacobi method with and without using LNS. Scale factors for diagonal values are ×10.

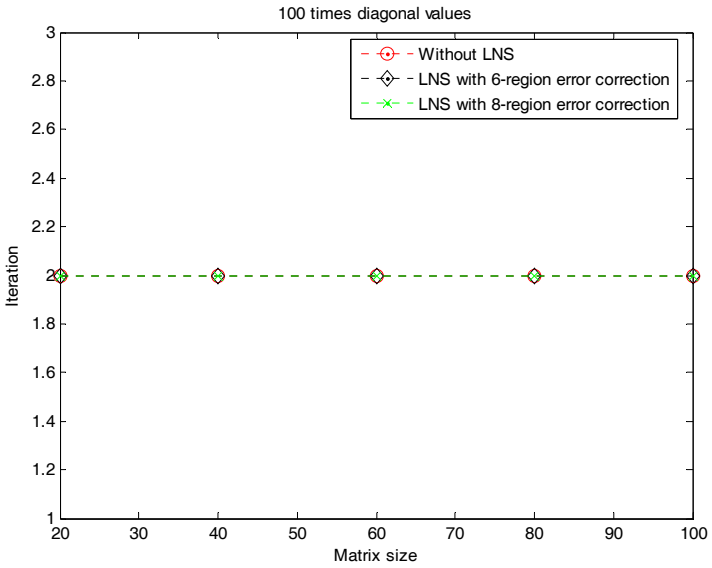

Figure 5.6 Number of iterations required for solving different size of linear systems for different diagonal values according to Jacobi method with and without using LNS. Scale factors for diagonal values are ×100.

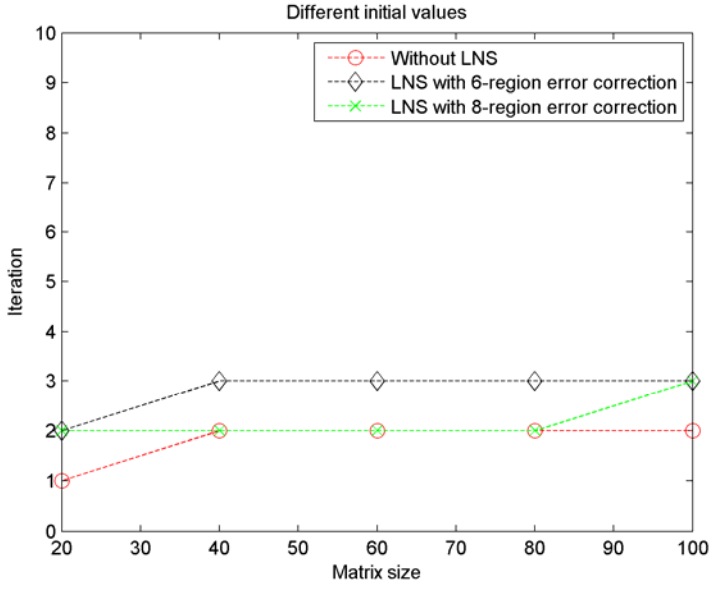

Figure 5.7 Number of iterations required for solving different size of linear systems according to Jacobi method with and without using LNS under different initial values.

tolerances and increase the data accuracy, the range of the fraction part of logarithm can be increased. Figure 5.7 shows that LNS have similar iterations as multiplication when different initial values for x are applied. It can be seen that different initial values for x do not have significant impact on convergence.

The relationship between of convergence and LNS can be expressed in the following mathematic form. Let $e_i^m = x_i^m - x_i$ be the difference between the i_{th} component of the exact solution x_i and the i_{th} component of the m_{th} iterate, $m \geq 0$.

$$\left|e_i^m\right| = \left| -\sum_{j \neq i} \frac{a_{ij}}{a_{ii}} (x_i^{m-1} - x_i) - \sum_{j \neq i} \frac{\varepsilon_j^m}{a_{ii}} + \varepsilon_i^m \right| \tag{5.2}$$

where $x_i^{m-1} - x_i = e_i^{m-1}$, ε_j^m is the error cause by the logarithm and antilogarithm conversions associated the multiplication of a_{ij} and x_j at m_{th} iteration, and ε_i^m is the error caused by the logarithm and antilogarithm division associated with $1/a_{ii}$ at m_{th} iteration. Let $\| \mathbf{e}^{m-1} \|_\infty = \max_{1 \leq i \leq n} \{ | e_i^{m-1} | \}$ and $\| \boldsymbol{\varepsilon}_j^m \|_\infty = \max_{j \neq i} \{ | \varepsilon_j^m | \}$. Then

$$\left|e_i^m\right| \leq \sum_{j \neq i} \left| \frac{a_{ij}}{a_{ii}} \right| \left\| \mathbf{e}^{m-1} \right\|_\infty + \left| \frac{(n-1) \left\| \boldsymbol{\varepsilon}_j^m \right\|_\infty}{a_{ii}} \right| + \left| \varepsilon_i^m \right| \tag{5.3}$$

Let

$$K = \max \left| \sum_{j \neq i} \frac{a_{ij}}{a_{ii}} \right| \tag{5.4}$$

Then Equation 5.3 becomes

$$\left|e_i^m\right| \leq K \left\| \mathbf{e}^{m-1} \right\|_\infty + \left| \frac{(n-1) \left\| \boldsymbol{\varepsilon}_j^m \right\|_\infty}{a_{ii}} \right| + \left| \varepsilon_i^m \right| \tag{5.5}$$

The first term is same as the sufficient condition for Jacobi method to converge without using LNS; i.e. if $K < 1$, $\mathrm{e}^m \to 0$ as $m \to \infty$. For example, e_i^1 will be multiplied by $\sum_{j \neq i} a_{ij} / a_{ii}$

for m–1 times at the m_{th} iteration. However, every iteration also generates a new second term and third term. At the m_{th} iteration, the accumulated errors caused by logarithm and antilogarithm conversions can be expressed as:

$$\sum_{k=1}^{m} K^{m-k}\left(\left|\frac{(n-1)\left\|\boldsymbol{\varepsilon}_j^k\right\|_\infty}{a_{ii}}\right| + \left|\varepsilon_i^k\right|\right) \tag{5.6}$$

Equation 5.6 shows the errors accumulate with the number of iterations (m) and is also proportional to the size of matrix (n). In order for LNS-based Jacobi method to absolutely converge, both Equation 5.1 and 5.6 must be satisfied. The accumulated errors expressed by Equation 5.6 can be kept less than the convergence tolerance by including proper error correction circuits in the logarithm and antilogarithm converters. Asume

$$\|\varepsilon\|_\infty = \left\|\varepsilon_j^k\right\|_\infty = \left\|\varepsilon_i^k\right\|_\infty = \max\left|\varepsilon_i^k\right| \tag{5.7}$$

Equation 5.6 can be simplified as:

$$\begin{cases} m\|\varepsilon\|_\infty\left(n/\left|a_{ii}\right|+1\right) & \text{when } K \to 1 \\ \|\varepsilon\|_\infty\left(n/\left|a_{ii}\right|+1\right) & \text{when } K \to 0 \end{cases} \tag{5.8}$$

Equation 5.8 matches the Matlab simulations in Figure 5.4, Figure 5.5 and Figure 5.6. When m and $\|\varepsilon\|_\infty$ are small, the numbers of iterations are similar for Jacobi method with or without using LNS. If current error correction algorithms where $\|\varepsilon\|_\infty$ is large, LNS must be used with caution when Jacobi method needs more iterations to converge.

5.1.3 Xilinx EDK Implementation of Three Iterative Methods

Xilinx EDK implementation was performed to show the performance of the three iterative methods, where the three iterative methods were implemented in pure software running on PPC and also implemented in HW/SW codesign using PPC and FPGA. The comparison of speeds of Jacobi, Gauss-Seidel and CG are shown in Figures 5.8, 5.9 and 5.10 respectively. Since different microprocessors might have different frequency specifications, clock cycle is

used to represent the speed instead of using actual time within this HW/SW codesign. In this HW/SW codesign, MMB block is implemented in hardware while the rest is unchanged (i.e., implemented in PPC based software). As matrix increases, the computation time of the software implemented designs grows exponentially, but HW/SW codesigns are almost linear lines with comparable smaller slopes. These results also indicate that all three iterative methods are matrix multiplication dominant, since only the MMB is implemented in hardware. Figure 5.1 shows that CG runs faster than Jacobi method in HW/SW codesign for a given tolerance (i.e., 10^{-6} in this case) , since both methods are matrix multiplication dominant and CG takes less number of iterations to converge compared to Jacobi. However, this result is also dependent on the matrix given for testing. If a matrix is highly diagonal dominant, where the diagonal values are 10 times or even 100 times larger the sum of non-diagonal values, Jacobi will take less number of iterations to converge and the difference of clock cycles between Jacobi and CG within HW/SW codesigns will also be reduced.

Three methods implemented in software and HW/SW codesign are also compared for same number of iterations (i.e. 6 iterations in this case). Figure 5.1 shows that CG runs slower than

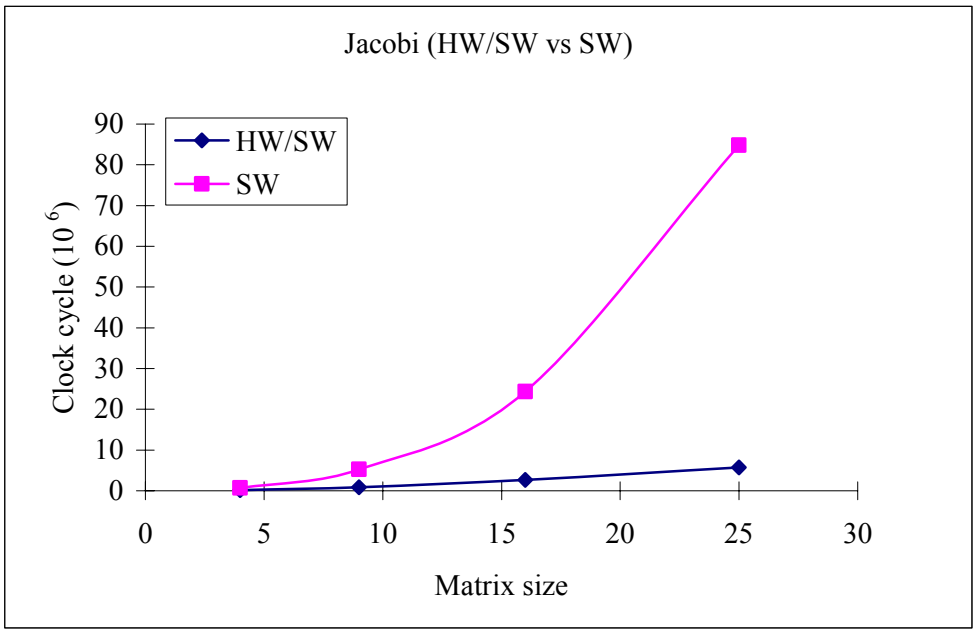

Figure 5.8: Speed comparison of EDK SW design and HW/SW codesign of Jacobi method

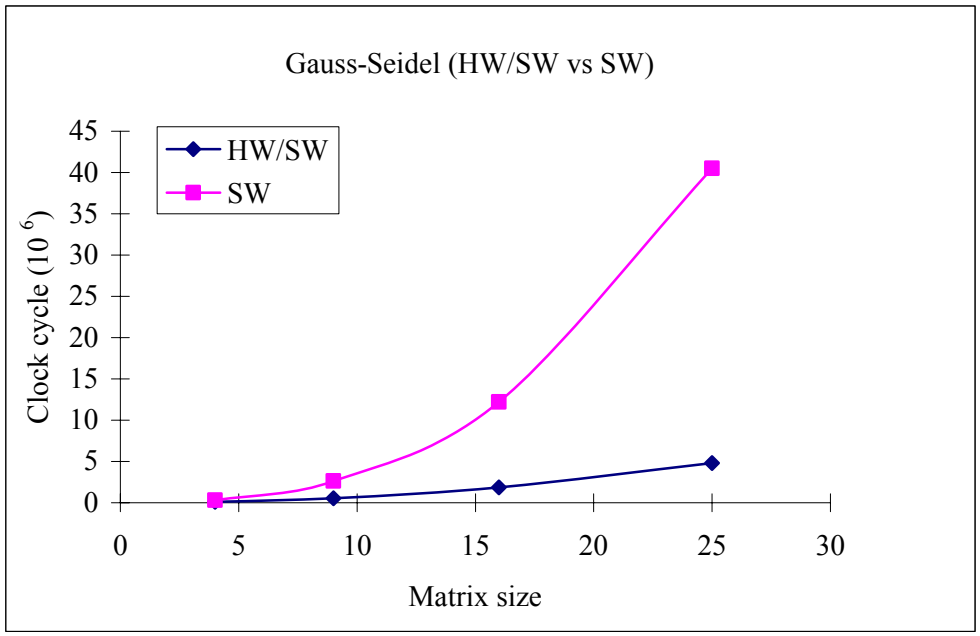

Figure 5.9: Speed comparison of EDK SW design and HW/SW codesign of GS method

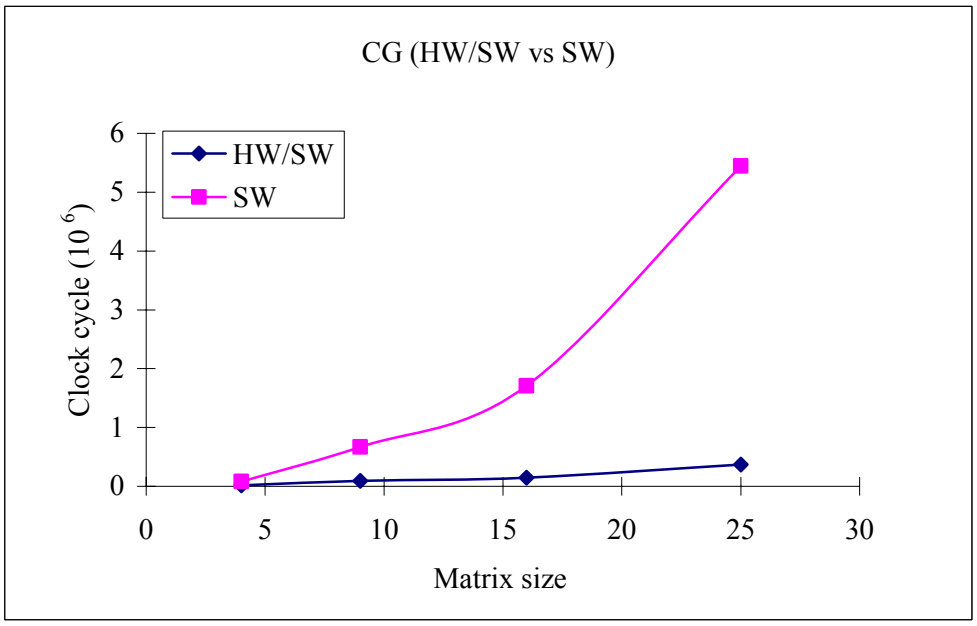

Figure 5.10: Speed comparison of EDK SW design and HW/SW codesign of CG method

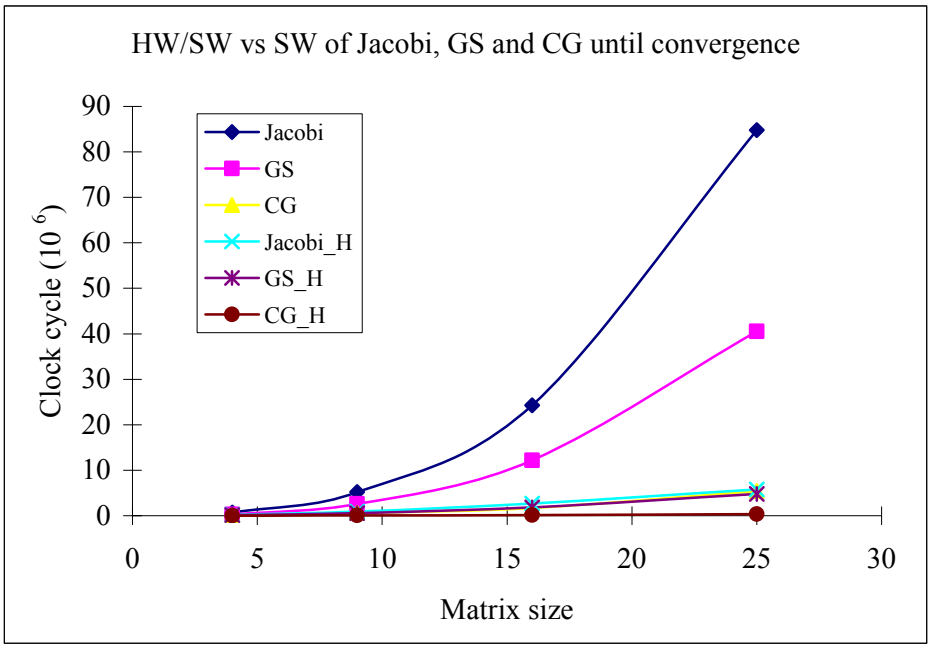

Figure 5.11: Speed comparison of Jacobi, Gauss-Seidel and CG method implemented in EDK SW design and HW/SW codesign for different iterations until convergence.

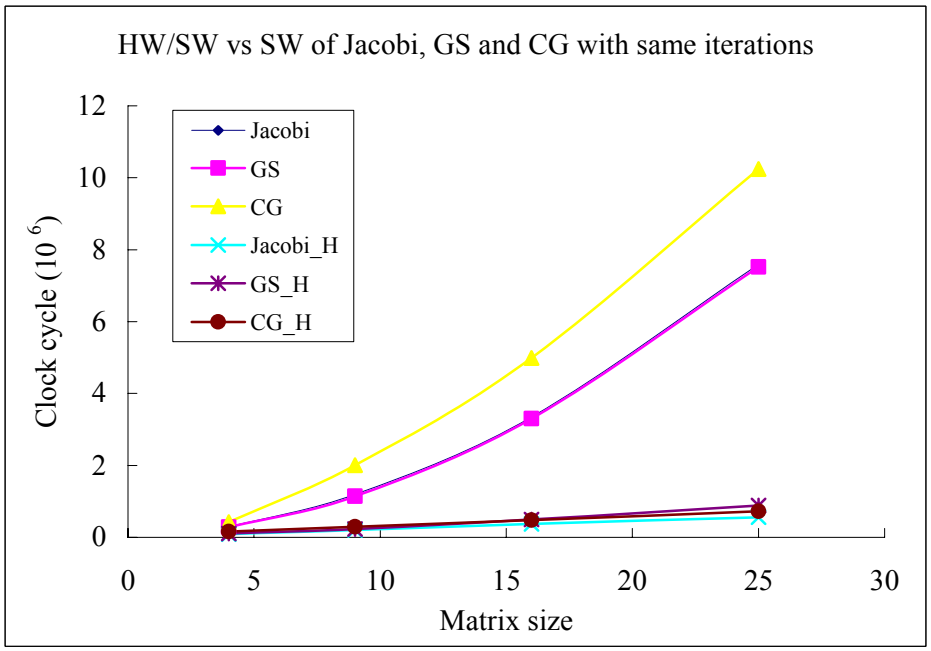

Figure 5.12: Speed comparison of Jacobi, Gauss-Seidel and CG method implemented in EDK SW design and HW/SW codesign for same numbers of iterations.

both Jacobi and Gauss-Seidel in software, but the differences of clock cycles among three methods are apparently reduced in HW/SW codesign. This result also indicates that CG is more matrix multiplication dominant compared to other two iterative methods.

5.1.4 Memory Consideration

Memory is one of the most important concerns in processing large linear systems. With proposed hardware architectures of iterative methods, memory space is proportional to the size of linear system. For example, for Jacobi method, every JPU must have read access to memories of vector b, coefficient matrix A, vector x and write access to memory of new vector x. For smaller size systems, the simplest form of the design is based on using build-in FPGA block RAMs (BRAM), which allows the circuit to run at maximum frequency. For example, the XC5VLX50 FPGA has 96×18 Kb BRAM which can provide enough memory space for a system with 160 equations. With four XC5VLX50 FPGAs, the maximum system size will be doubled.

Every JPU requires read access to certain rows in matrix A and corresponding elements from vector b based on the position of JPU. These elements can be loaded once into the appropriate FPGAs and used until the end of the process. Every FPGA also should have an identical image of the vector x. However, new elements of vector x (depending on the number of JPUs) are generated after an iteration. In this case, using a shared bus and an arbitration method, new elements of x can be distributed and shared among FPGAs during the next iteration. Assuming there are four FPGAs with 8 JPUs in total, at the end of each iteration eight new elements of vector x will be created. Eight write accesses to the shared bus will be required to distribute the new elements.

In Xilinx EDK HW/SW codesigns of selected iterative methods, on-chip dual-port BRAM is used for storing instructions and data in order to simplify the design. For larger systems, where internal BRAM is not adequate, using single port external memory with large number of JPUs is able to compare the performance of larger size matrices. However, data needs to be fetched from external memory through PLB bus, and also PLB bus is always occupied by PPC for checking the status of MMB block at the same time. This situation causes time

confliction due to limited ports of external memory. Nevertheless, there are other ways to use external memory. One of them is to use interrupt signal of PPC to check the status of MMB block, but will take more time for exploring EDK and debugging.

5.2 Performance Analysis of Direct Methods

5.2.1 Matlab Comparison of LU and WZ

The well-known LU factorization is one of the most commonly used algorithms to solve linear systems on sequential computers nowadays. Unlike WZ factorization, LU algorithm essentially processes elimination and factorization serially. For an $n \times n$ matrix, it takes $(n\text{-}1)$ steps to factorize in LU, but only $n/2$ steps if n is even and $(n\text{-}1)/2$ steps if n is odd in WZ.

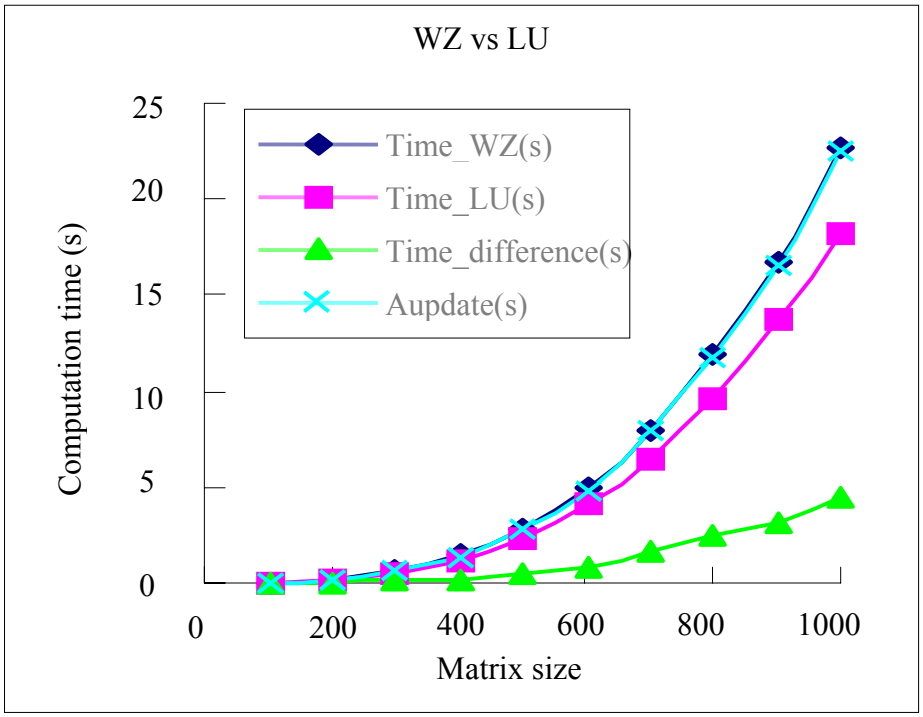

Figure 5.13: Computation time for WZ and LU factorization, time difference between WZ, and LU and computation time for the update of $A_{n\text{-}2i}$ in WZ.

Hence, WZ is more suitable for parallel computation compared to LU. Research made by Yalamov [15, 33] showed that WZ based matrix solving method has obtained a 20% gain in execution time compared to LU method when processed on a parallel computer.

In single-processor computer based Matlab simulations, however, WZ doesn't exhibit any advantage since every step is done in serial. The Matlab function function *tic-toc* is again used to calculate the average computation times of WZ and LU in terms of CPU time. Figure 5.1 shows that the computation time of both methods is increased exponentially but WZ increases faster than LU as matrix increases. The time difference between them is also increased exponentially as matrix increases. The simulation results also shows that the major computation time cost in WZ is from the update of A_{n-2i}, which takes up to more than 90% of total time and is close to 99% when matrix increases over 300. Therefore, it is expected that the performance will be improved if the update of A_{n-2i} is implemented in hardware with HW/SW codesign.

5.2.2 Xilinx EDK Implementation of WZ Factorization

Xilinx EDK implementation was also performed to show the performance of WZ factorization, where WZ factorization was implemented in pure software running on PPC and also implemented in HW/SW codesign using PPC and FPGA. The comparison of speeds between these two implementations is shown in Figure 5.1. In this figure, clock cycle is again used to represent the speed instead of using actual time. In this HW/SW codesign, Aupdate block is implemented in hardware while the rest is unchanged (i.e., implemented in software). The matrices generated based on Poisson equation are also used in these simulations. When matrix size is small (e.g. less than 9), the time difference of HW/SW codesign and software implementation is not obvious. As matrix size increases, the computation time of both designs increase exponentially, the software implementation increases faster than HW/SW codesign, and the time difference between two implementations also increases exponentially. The simulation results show that the performance of WZ improves when the update of A_{n-2i} is implemented in hardware, which

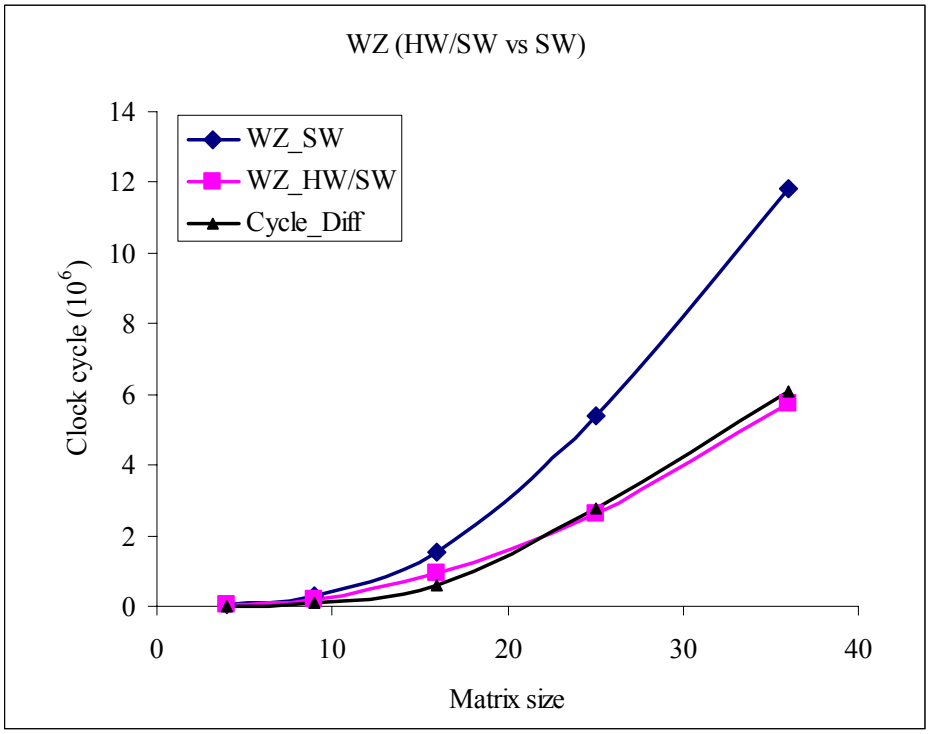

Figure 5.14: Speed comparison of EDK based SW design and HW/SW codesign of WZ factorization

indicates that the update of the remaining matrix A_{n-2i} is the most time consuming part in WZ factorization.

In this HW/SW codesign of WZ factorization, only the inner loop k, i.e., the update of one row of the remaining matrix A_{n-2i} as described in Chapter 4, is implemented in hardware. The performance will be improved if loop j is also implemented in hardware, which means the update of the entire remaining matrix A_{n-2i} is processed in hardware. The performance will be further improved if loop i is also replaced by hardware. This HW/SW codesign is also tested with other matrices, where matrices A is randomly generated within range of [10 20] and vector b is generated from matrix A and a given vector x. The simulation results are similar with Figure 5.1. The implementation with these matrices shows the generality of this design.

This chapter discussed performance comparison of selected iterative methods and direct method under each category. The most time consuming parts, i.e. the matrix multiplication of

iterative methods and the update of WZ factorization, are implemented in hardware with HW/SW codesigns. The EDK implementation results show that HW/SW codesigns achieve significant improvement over pure software design. Convergence analysis of LNS-based Jacobi processor shows the error caused by logarithm and antilogarithm conversion accumulates as the number of iterations grows. The accumulated error is also proportional to the size of a linear system.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis focuses on the development of parallel processing of matrix solving algorithms and hardware architectures for the selected algorithms including Jacobi, Gauss-Seidel, CG and WZ factorization. Several matrix solving methods were discussed targeting on their parallel property. The FPGA-based general hardware architectures of these methods were proposed considering the hardware resource. The performance comparisons of these methods were analyzed based on the results of Matlab simulations and Xilinx EDK HW/SW codesign simulations. For iterative methods, Matlab simulations were performed to compare the Jacobi, Gauss-Seidel and CG methods. The simulation results indicate that the key to a fast implementation of the three methods is a fast implementation of matrix multiplication. The simulation results also show that CG method takes less number of iterations for any given tolerance. For different sizes of matrices, CG method also takes less number of iterations to reach a given tolerance compared with other two methods. Thus, in terms of iterations, CG is a better algorithm to solve given linear systems. However, the total computation time of the three methods indicates the different results. The simulation results show that CG takes more computation time than other two methods as matrix size increases over certain size (e.g. 400×400 in this case), since matrix-vector multiplication is a more dominant factor in CG method than in the other two methods. In order to improve the speed of matrix multiplication, two approaches are used. The first approach is to use LNS instead of matrix multiplication. In LNS, fast multiplication and division operations can be achieved by using addition and subtraction operations on the logarithms of the input data. Convergence analysis of a LNS

based Jacobi processor was discussed. Two major factors were identified and considered in this convergence analysis. Firstly, in any hardware architecture for the Jacobi iterative method, only a set of unknowns can be processed in parallel due to the constraint of hardware resources. Secondly, the conversions of logarithm-to-floating point and floating-to-logarithm introduce additional error. On the other hand, the convergence analysis also shows that the proper error correction algorithm does not necessarily to be highly accurate for a linear system with considerably smaller matrix or extremely diagonally dominated matrix. The second approach is to implement matrix multiplications of the three methods in hardware. The three iterative methods were implemented with Xilinx EDK HW/SW codesign where MMB is implemented in hardware and the rest is unchanged (i.e. implemented in software). The EDK testing results show that, as the size of matrix increases, the computation time of software implemented designs grow exponentially, but the computation time of HW/SW codesigns are relatively linear as compared to software implementation, which proves that all three methods are matrix-multiplication dominant. CG runs faster than Jacobi method in the HW/SW codesign for a given tolerance, due to that fact that matrix multiplications dominate the computation time of all three methods while CG requires less number of iterations to converge comparing to other two methods. The EDK testing results also show that with software implementation, CG method is slower than both Jacobi and GS methods for the same number of iterations. However, the differences in computation time are improved significantly in HW/SW codesign, which is evidence that CG method is more matrix-multiplication dominant comparing to the other two methods.

For direct methods, FPGA-based hardware architecture and Xilinx EDK HW/SW codesign of WZ factorization are also presented. Single unit and scalable architectures of WZ factorization are proposed and analyzed under different constraints. Matlab simulations were performed to compare the performance of LU and WZ. The simulation results show that the WZ factorization runs faster than the LU factorization on parallel processors but slower on single processor. The simulation results also indicate that the most time consuming part of WZ comes from the update of the remaining matrix A_{n-2i}. Hence, the matrix update is implemented in hardware with Xilinx EDK HW/SW codesign of WZ factorization to

improve the overall performance. The EDK simulation results show that the performance of HW/SW codesign is apparently improved over pure software implementation.

6.2 Suggestions for Future Work

The implementation of the selected matrix solving methods remains for future work. Some of future work is suggested as follows.

1. Performance comparison of LU and WZ on FPGA-based design: In order to compare LU with WZ in hardware implementation, hardware architecture of LU needs to be designed in a similar way as WZ. Figure 6.1 shows one way to implement hardware architecture of LU factorization. This architecture has one L_decomposer and one U_decomposer to solve matrices L and U respectively. Two decomposers process alternatively to decompose a row and column of L and U respectively. At the beginning, the memory is initialized with matrix A, where all three matrices can be stored in matrix A. During the factorization, the modified elements in matrix A are deleted and replaced with matrices L and U. The diagonal of matrix L or U contains all 1's and is not stored explicitly. From Equations 2.7, all nonzero elements on the preceding rows and columns have to be available before the k_{th} loop step begins. This relationship hinders the parallel property of LU, although the elements of k_{th} row or column can be calculated in parallel, which means multiple multipliers can be paralleled at the input. The numbers of multipliers and adders need to be determined according to the available hardware resource for pure FPGA-based hardware implementation. Considering available hardware resource, the comparison of LU and WZ can be simplified in HW/SW codesign. Matlab simulation of LU factorization shows that the most time consuming part comes from the multipliers and accumulators of L_decomposer and U_decomposer shown in Figure 6.1 respectively. Therefore, those parts can be implemented in hardware with HW/SW codesign.

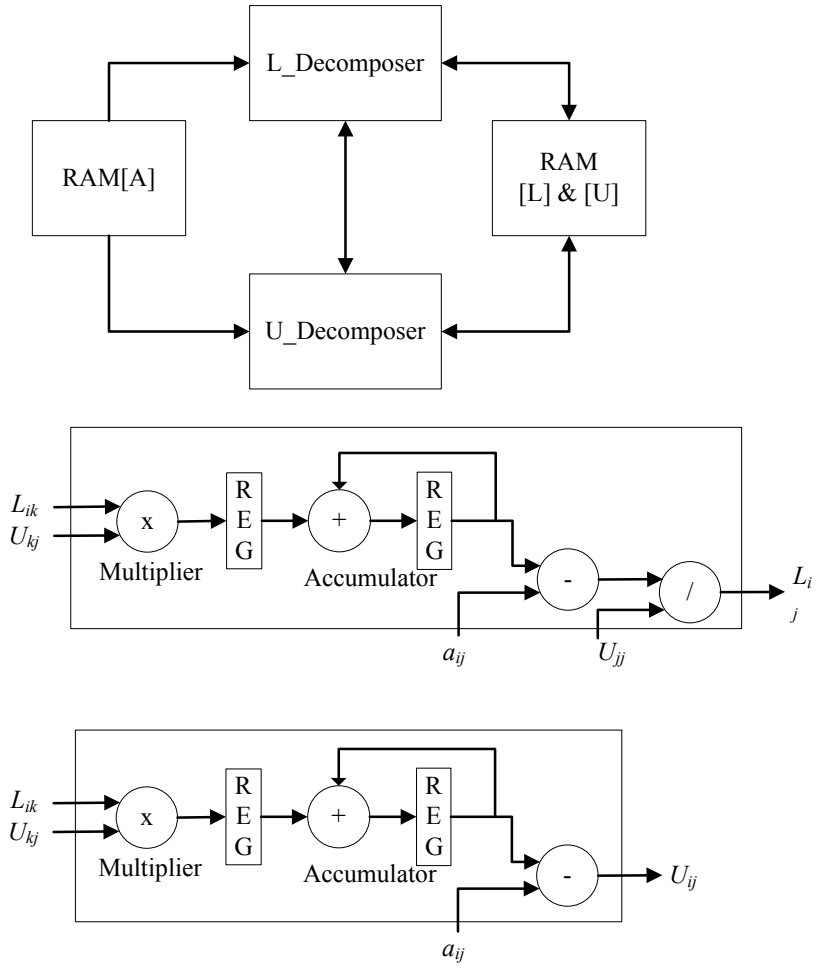

Figure 6.1: General hardware architecture of LU factorization

2. FPGA-based hardware implementation of the proposed hardware architectures of selected iterative methods and direct methods. In particular, multi-FPGA implementation of matrix solving methods according to TMD architecture can be further investigated. The implementation results should be compared with the performance of other multiprocessor-based computing.

3. HW/SW implementation of circuit simulators. The most time consuming parts of circuit simulation are matrix solving and model evaluation. These parts can be implemented in hardware with HW/SW codesign.

4. Hardware implementation of reordering techniques. The reordering of sparse matrix into dense matrix is usually implemented in software for simplicity purpose. If the available hardware resource allows, these techniques can be implemented in hardware to improve the overall performance.

References

[1] Q.K. Zhu, *Power Distribution Network Design for VLSI*, Wiley-IEEE, 2004.

[2] J. Ogrodzki, *Circuit Simulation Methods and Algorithms*, CRC Press, 1994.

[3] A. Ghali, A.M. Nevill and T.G. Brown, *Structural Analysis: A Unified Classical and Matrix Approach*, Taylor & Francis, 2003.

[4] J. Schewel, *Configurable Computing: Technology and Applications*, SPIE, 1998.

[5] R. Hartenstein and H. Grünbacher, *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable*, Springer, 2000.

[6] W. Badawy and G. Jullien*, System-on-chip for Real-time Applications*, Springer, 2002.

[7] "International Technology Roadmap for Semiconductors 2007 Edition Design," The International Technology Roadmap for Semiconductors, 2007.

[8] K.E. Emam, J-N. Drouin and W. Melo, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press, 1997.

[9] P.-Y. Chung and I.N Hajj, "Parallel solution of sparse linear systems on a vector multiprocessor computer," *IEEE International Symposium on Circuits and Systems*, pp. 1577-1580, May. 1990.

[10] K. Compton and S. Hauck, *Reconfigurable Computing: A Survey of Systems and Software*, ACM Computing Survey, pp. 171-210, Apr. 2002.

[11] R. Tessier and W. Burleson, "Reconfigurable computing and digital signal processing: a survey," *Journal of VLSI Signal Processing*, pp. 7-27, Feb. 2001.

[12] G.H. Golub and C.F.V. Loan, *Matrix Computations*, Johns Hopkins University Press, 1996.

[13] R.A. Horn and C.R. Johnson, *Matrix Analysis*, Cambridge University Press, 1985.

[14] D.J. Evans and R. Abdullah, "The parallel implicit elimination (PIE) method for the solution of linear systems," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 4, No. 1-2, pp. 153-162, 1994.

[15] P. Yalamov and D.J. Evans, "The WZ matrix factorisation method," *Parallel Computing*, vol. 21, pp. 1111-1120, 1995.

[16] I.S Duff, A.M. Erisman, and J.K. Reid, *Direct Methods for Sparse Matrice*, Oxford University Press, Oxford, England, 1990.

[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, 1985.

[18] *EDK Concepts, Tools, and Techniques: A Hands-on Guide to Effective Embedded System Design*, Xilinx support: Embedded Development Kit (EDK) 9.2i Documentation, Jan. 2007.

[19] J.N. Mitchell, Jr., "Computer multiplication and division using binary logarithms," *IEEE Transactions on Electronic*, vol. 11, pp. 512-517, Aug. 1962.

[20] S.L. SanGregory, R.E. Siferd, C. Brother, and D. Gallagher, "A fast, low-power logarithm approximation with CMOS VLSI implementation," *In proceeding of Midwest Symposium on Circuits and Systems*, pp. 388-391, Aug. 1999.

[21] K.H. Abed and R. Siferd, "CMOS VLSI implementation of a low-power logarithmic converter," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1421-1433, Nov. 2003.

[22] K.H. Abed and R.E. Siferd, "VLSI implementation of a low-power antialgorithmic converter," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1221-1228, Sep. 2003.

[23] D.J. Evans, "Parallel strategies for linear systems of equations," *International Journal of Computer Mathematics*, vol. 81, No. 4, pp. 417-416, Apr. 2004.

[24] K.E. Emam, J-N. Drouin and W. Melo, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society Press, 1997.

[25] R.M. Kielkowski, *Inside SPICE*, 2^{nd} *edition*, McGraw-Hill, Inc, 1998.

[26] A. Robbins and W.C. Miller, *Circuit Analysis: Theory and Practice*, Thomson Delmar Learning, 2003.

[27] G.A. Meurant, *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*, Society for Industrial and Applied Mathematics, 2006.

[28] J.R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," *Carnegie Mellon University*, Aug. 1994.

[29] M.R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of Research of the National Bureau of Standards*, vol. 49, Dec. 1952.

[30] V. Faber and T. Manteuffel, "Necessary and sufficient conditions for the existence of a conjugate gradient method," *SIAM Journal of Numerical. Analysis*, vol. 21, pp. 315-339, 1984.

[31] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *Journal of Research of the National Bureau of Standards*, vol. 45, pp. 255-282, 1950.

[32] D.J. Evans, "Implicit matrix elimination (IME) schemes," *International Journal of Computer Mathematics*, pp, 229-237. 1993.

[33] R. Asenjo, M. Ujaldón, and E.L. Zapata, "Parallel WZ factorization on mesh multiprocessors," *Microprocessing and Microprogramming*, vol. 38, pp. 319-326, Sep. 1993.

[34] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Survey*, pp. 171–210, Apr. 2002.

[35] R. Tessier and W. Burleson, "Reconfigurable computing and digital signal processing: a survey," *Journal of VLSI Signal Processing*, pp. 7–27, Feb. 2001.

[36] A. Patel, C. A. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow, "A scalable FPGA-based multiprocessor," *In proceeding of the* 14th *Annual IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 111-120, Apr. 2006.

[37] M.A.S.D. Fuentes, *A Parallel Programming Model for A Multi-FPGA Multiprocessor Machine*, M.Sc. thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, 2006.

[38] *Xilinx Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.* http://www.xilinx.com/support/documentation/virtex-ii_pro.htm.

[39] *The Infiniband Architecture Specification R1.2*, Technical report, InfiniBand Trade Association, Oct 2004. http://www.infinibandta.org.

[40] L. Zhuo and V.K. Prasanna, "Hardware/Software co-design for matrix computations on reconfigurable computing systems," *In proceeding of Parallel and Distributed Processing Symposium*, pp. 23-30, Mar. 2007.

[41] Cray XD1. http://www.cray.com/downloads/cray_xd1_datasheet.pdf.

[42] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Technical Report, UT-CS-94-230, 1994.

[43] X. Wang and S.G. Ziavras, "Exploiting mixed-mode parallelism for matrix operations on the HERA architecture through reconfiguration," *IEE Proceedings Computers and Digital Techniques*, vol 153, pp. 249-260, Jul. 2006.

[44] L. Zhuo and V.K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on FPGAs," 18th *International Parallel and Distributed Processing Symposium*, pp. 92–101, Apr. 2004.

[45] J. Liang, R. Tessier, and O. Mencer, "Floating point unit generation and evaluation for FPGAs," 11th *Annual IEEE Symp. on Field Programmable Custom Computing Machines*, pp. 185–194, Apr. 2003.

[46] G.R. Morris and V.K. Prasanna, "An FPGA-based floating-point Jacobi iterative solver," *In proceeding of* 8th *International Symposium on Parallel Architecture, Algorithms and Networks* (ISPAN 2005), pp. 7-9, Dec. 2005.

[47] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington, "A sparse matrix library in C++ for high performance architectures," *Proceedings of the Second Object Oriented Numerics Conference*, pp. 214-218, 1994.

[48] R. Shahnaz, A. Usman, and I.R. Chughtai, "Review of storage techniques for sparse matrices," 9th *International Multitopic Conference*, pp. 1-7, Dec. 2005.

[49] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1994.

[50] Y. Saad, *SPARSKIT: A Basic Toolkit for Sparse Matrix Computations*, Technical Report, Computer Science Department, University of Minnesota, Jun. 1994.

[51] P. Huang, D.H.-Y. Teng, K. Wahid, and S.-B. Ko, "Convergence aAnalysis of Jacobi iterative method using logarithmic number system," 7th *IEEE/ACIS International Conference on Computer and Information Science*, pp. 27-32, May. 2008.

[52] Xilinx ISE 10.1 Design Suite Software Manuals and Help. http://www.xilinx.com/support/software_manuals.htm

[53] W. Xiaofang and S.G. Ziavras, "Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines," *Concurrency and Computation: Practice & Experience,* vol 16, No. 4, pp. 319 – 343, 2004.

[54] D. Koester, S. Ranka, and G.C. Fox, "A parallel Gauss-Seidel algorithm for sparse power systems matrices," *Proceedings of Supercomputing* , pp. 184-193, Nov. 1994.

[55] W. Xiaofang and S.G Ziavras, "A configurable multiprocessor and dynamic load balancing for parallel LU factorization," *Parallel and Distributed Processing Symposium, Proceedings* 18th *international,* pp. 234-242, Apr. 2004.

[56] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *Journal of Research of the National Institute of Standards and Technology*, vol. 45, pp. 255-282, 1950.

[57] R.W. Shonkwiller and L. Lefton, *An Introduction to Parallel and Vector Scientific Computing*, Cambridge University Press, 2006.

[58] H.-J Kim, B.-G. Nam, J.-H. Sohn, J.-H. Woo, and H.-J. Yoo, "A 231-MHz, 2.18-mW 32-bit logarithmic arithmetic unit for fixed-point 3-D graphics system," *IEEE Journal of Solid State Circuits*, vol. 41, no. 11, pp. 2373-2381, Nov. 2006.

[59] I. Bravo, P. Jimenez, M. Mazo, J.L. Lazaro, J.J. de las Heras, and A. Gardel, "Different proposals to matrix multiplication based on FPGAs," *IEEE International Symposium on*, pp. 1709-1714, Jun. 2007.

Appendix A
Numerical Examples

1. Parallel Implicit Elimination (PIE)

Consider the following 5×5 dense matrix.

$$A=\begin{bmatrix} 6 & 1 & 1 & 2 & 1 \\ 1 & 8 & 2 & 1 & 2 \\ 1 & 1 & 5 & 1 & 1 \\ 2 & 1 & 1 & 6 & 1 \\ 1 & 2 & 1 & 1 & 6 \end{bmatrix}$$

Stage 1: $i = 1$

Step 1: The first and last rows of matrix Z are obtained as follows:

$$z_{11}=a_{11}=6,\ \ z_{12}=a_{12}=1,\ \ z_{13}=a_{13}=1,\ \ z_{14}=a_{14}=2,\ \ z_{15}=a_{15}=1.$$
$$z_{51}=a_{51}=1,\ \ z_{52}=a_{52}=2,\ \ z_{53}=a_{53}=1,\ \ z_{54}=a_{55}=1.\ \ z_{55}=a_{55}=6.$$

Step 2: Solve three sets of 2×2 linear systems to eliminate elements of first and last columns in matrix Z.

For $j=2$:

$$\left.\begin{aligned} a_{11}w_{21}+a_{51}w_{25}&=a_{21}\\ a_{15}w_{21}+a_{55}w_{25}&=a_{25}\end{aligned}\right\}\rightarrow\left.\begin{aligned} 6w_{21}+w_{25}&=1\\ w_{21}+6w_{25}&=2\end{aligned}\right\}\rightarrow\begin{aligned} w_{21}&=\tfrac{4}{35}\\ w_{25}&=\tfrac{11}{35}\end{aligned}$$

For $j=3$:

$$\left.\begin{aligned} a_{11}w_{31}+a_{51}w_{35}&=a_{31}\\ a_{15}w_{31}+a_{55}w_{35}&=a_{35}\end{aligned}\right\}\rightarrow\left.\begin{aligned} 6w_{31}+w_{35}&=1\\ w_{31}+6w_{35}&=1\end{aligned}\right\}\rightarrow\begin{aligned} w_{31}&=\tfrac{5}{35}\\ w_{35}&=\tfrac{5}{35}\end{aligned}$$

For $j=4$:

$$\left.\begin{aligned} a_{11}w_{41}+a_{51}w_{45}&=a_{41}\\ a_{15}w_{41}+a_{55}w_{45}&=a_{45}\end{aligned}\right\}\rightarrow\left.\begin{aligned} 6w_{21}+w_{25}&=2\\ w_{21}+6w_{25}&=1\end{aligned}\right\}\rightarrow\begin{aligned} w_{41}&=\tfrac{11}{35}\\ w_{45}&=\tfrac{4}{35}\end{aligned}$$

Step 3: Update $A_{n\text{-}2}$

For $j = 2$ and $k = 2,\ 3,\ 4$

$$\left.\begin{aligned} a_{22} &= a_{22} - w_{21}z_{12} - w_{25}z_{52} \\ a_{23} &= a_{23} - w_{21}z_{13} - w_{25}z_{53} \\ a_{24} &= a_{24} - w_{21}z_{14} - w_{25}z_{54} \end{aligned}\right\} \rightarrow \begin{aligned} a_{22} &= \tfrac{254}{35} \\ a_{23} &= \tfrac{55}{35} \\ a_{24} &= \tfrac{16}{35} \end{aligned}$$

For $j = 3$ and $k = 2,\ 3,\ 4$

$$\left.\begin{aligned} a_{32} &= a_{32} - w_{31}z_{12} - w_{35}z_{52} \\ a_{33} &= a_{33} - w_{31}z_{13} - w_{35}z_{53} \\ a_{34} &= a_{34} - w_{31}z_{14} - w_{35}z_{54} \end{aligned}\right\} \rightarrow \begin{aligned} a_{32} &= \tfrac{20}{35} \\ a_{33} &= \tfrac{165}{35} \\ a_{34} &= \tfrac{20}{35} \end{aligned}$$

For $j = 4$ and $k = 2,\ 3,\ 4$

$$\left.\begin{aligned} a_{42} &= a_{42} - w_{41}z_{12} - w_{45}z_{52} \\ a_{43} &= a_{43} - w_{41}z_{13} - w_{45}z_{53} \\ a_{44} &= a_{44} - w_{41}z_{14} - w_{45}z_{54} \end{aligned}\right\} \rightarrow \begin{aligned} a_{42} &= \tfrac{16}{35} \\ a_{43} &= \tfrac{20}{35} \\ a_{44} &= \tfrac{184}{35} \end{aligned}$$

Stage 2: $i = 2$

Step 1: The 2^{nd} and 4^{th} rows of matrix Z are obtained as follows:

$$z_{22} = a_{22} = \tfrac{254}{35},\ \ z_{23} = a_{23} = \tfrac{55}{35},\ \ z_{24} = a_{24} = \tfrac{16}{35}.$$
$$z_{42} = a_{52} = \tfrac{16}{35},\ \ z_{43} = a_{43} = \tfrac{20}{35},\ \ z_{44} = a_{45} = \tfrac{184}{35}.$$

Step 2: Solve one set of 2×2 linear system to eliminate elements of 2^{nd} and 4^{th} columns in matrix Z.

For $j = 3$:

$$\left.\begin{aligned} a_{22}w_{32} + a_{42}w_{34} &= a_{32} \\ a_{24}w_{32} + a_{44}w_{34} &= a_{34} \end{aligned}\right\} \rightarrow \left.\begin{aligned} \tfrac{254}{35}w_{31} + \tfrac{16}{35}w_{35} &= \tfrac{20}{35} \\ \tfrac{16}{35}w_{31} + \tfrac{184}{35}w_{35} &= \tfrac{20}{35} \end{aligned}\right\} \rightarrow \begin{aligned} w_{32} &= \tfrac{762}{10541} \\ w_{34} &= \tfrac{17}{166} \end{aligned}$$

Step 3: Update $A_{n\text{-}4}$

For $j = 3$ *and* $k = 3$

$$a_{33} = a_{33} - w_{32}a_{23} - w_{34}a_{43} \rightarrow a_{33} = \tfrac{47879}{10541}$$

The matrices Z is obtained as follows

$$Z=\begin{bmatrix} 6 & 1 & 1 & 2 & 1 \\ & \frac{254}{35} & \frac{55}{35} & \frac{16}{35} & \\ & & \frac{47879}{10541} & & \\ & \frac{16}{35} & \frac{20}{35} & \frac{184}{35} & \\ 1 & 2 & 1 & 1 & 6 \end{bmatrix}$$

2. WZ Factorization

Consider the following 4×4 dense matrix.

$$A=\begin{bmatrix} 5 & 1 & 2 & 1 \\ 1 & 6 & 1 & 2 \\ 2 & 1 & 7 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix} \text{ and } b=\begin{bmatrix} 9 \\ 10 \\ 11 \\ 8 \end{bmatrix}$$

Factorize matrix A into W and Z in the forms of Equations 2.21 and 2.22.

Stage1: i=1

Step 1: Obtain first and last rows of matrix Z as follows:

$$z_{11}=a_{11}=5,\ \ z_{12}=a_{12}=1,\ \ z_{13}=a_{13}=2,\ \ z_{14}=a_{14}=1,$$
$$z_{41}=a_{41}=1,\ \ z_{42}=a_{42}=2,\ \ z_{43}=a_{43}=1,\ \ z_{44}=a_{44}=4.$$

Step 2: Solve two sets of 2×2 linear systems to obtain elements of first and last columns of matrix W.

For $j=2$:

$$\left.\begin{aligned} z_{11}w_{21}+z_{41}w_{24}&=a_{21} \\ z_{14}w_{21}+z_{44}w_{24}&=a_{24} \end{aligned}\right\} \rightarrow \left.\begin{aligned} 5w_{21}+w_{24}&=1 \\ w_{21}+4w_{24}&=2 \end{aligned}\right\} \rightarrow \begin{aligned} w_{21}&=\tfrac{2}{19} \\ w_{24}&=\tfrac{9}{19} \end{aligned}$$

For $j=3$:

$$\left.\begin{aligned} z_{11}w_{31}+z_{41}w_{34}&=a_{31} \\ z_{14}w_{31}+z_{44}w_{34}&=a_{34} \end{aligned}\right\} \rightarrow \left.\begin{aligned} 5w_{31}+w_{34}&=2 \\ w_{31}+4w_{34}&=1 \end{aligned}\right\} \rightarrow \begin{aligned} w_{31}&=\tfrac{7}{19} \\ w_{34}&=\tfrac{3}{19} \end{aligned}$$

Step 3: Update $A_{n\text{-}2}$

For $j = 2$ and $k = 2,\ 3$

$$\left.\begin{aligned} a_{22} &= a_{22} - w_{21}z_{12} - w_{24}z_{42} \\ a_{23} &= a_{23} - w_{21}z_{13} - w_{24}z_{43} \end{aligned}\right\} \rightarrow \begin{aligned} a_{22} &= \tfrac{94}{19} \\ a_{23} &= \tfrac{6}{19} \end{aligned}$$

For $j = 3$ and $k = 2,\ 3$

$$\left.\begin{aligned} a_{32} &= a_{32} - w_{31}z_{12} - w_{34}z_{42} \\ a_{33} &= a_{33} - w_{31}z_{13} - w_{34}z_{43} \end{aligned}\right\} \rightarrow \begin{aligned} a_{32} &= \tfrac{6}{19} \\ a_{33} &= \tfrac{116}{19} \end{aligned}$$

The matrices W and Z are obtained as follows

$$W = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \frac{2}{19} & 1 & 0 & \frac{9}{19} \\ \frac{7}{19} & 0 & 1 & \frac{3}{19} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } Z = \begin{bmatrix} 5 & 1 & 2 & 1 \\ 0 & \frac{94}{19} & \frac{6}{19} & 0 \\ 0 & \frac{6}{19} & \frac{116}{19} & 0 \\ 1 & 2 & 1 & 4 \end{bmatrix}$$

Solution procedure of system: $Wy = b$:

Stage 1: Obtain y_1 and y_4,

$$y_1 = b_1 = 9,$$
$$y_4 = b_4 = 8.$$

Then update the remaining elements of vector b,

$$j = 2: \quad b_2 = b_2 - w_{21}y_1 - w_{24}y_4 \rightarrow b_2 = \tfrac{100}{19},$$
$$j = 4: \quad b_3 = b_3 - w_{31}y_1 - w_{34}y_4 \rightarrow b_3 = \tfrac{122}{19}.$$

Stage 2:

$$y_2 = b_2 = \tfrac{100}{19},$$
$$y_3 = b_3 = \tfrac{122}{19}.$$

Solution procedure for system: $Zx = y$

Stage 1: n is even,

$$\left.\begin{array}{l} z_{22}x_2 + z_{23}x_3 = y_2 \\ z_{32}x_2 + z_{33}x_3 = y_3 \end{array}\right\} \rightarrow \left.\begin{array}{l} \frac{94}{19}x_2 + \frac{6}{19}x_3 = \frac{100}{19} \\ \frac{6}{19}x_2 + \frac{116}{19}x_3 = \frac{122}{19} \end{array}\right\} \rightarrow \begin{array}{l} x_2 = 1 \\ x_3 = 1 \end{array}$$

Then, update the remaining elements of vector y

$$j = 1: \ y_1 = y_1 - x_2 z_{12} - x_3 z_{13} \rightarrow y_1 = 6,$$
$$j = 4: \ y_4 = y_4 - x_2 z_{42} - x_3 z_{13} \rightarrow y_4 = 5.$$

Stage 2:

$$\left.\begin{array}{l} z_{11}x_1 + z_{14}x_4 = y_1 \\ z_{41}x_1 + z_{44}x_4 = y_4 \end{array}\right\} \rightarrow \left.\begin{array}{l} 5x_1 + x_4 = 6 \\ x_1 + 4x_4 = 5 \end{array}\right\} \rightarrow \begin{array}{l} x_1 = 1 \\ x_4 = 1 \end{array}$$

The final solution of x is $x = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}^{\mathrm{T}}$.

3. ZW Factorization

Consider the same 4×4 dense matrix as showed in the numerical example of WZ factorization.

$$A = \begin{bmatrix} 5 & 1 & 2 & 1 \\ 1 & 6 & 1 & 2 \\ 2 & 1 & 7 & 1 \\ 1 & 2 & 1 & 4 \end{bmatrix}$$

Factorize matrix A into Z' and W' in the forms of Equations 4.1 and 4.2.

Stage1: i=1

Step 1:The 2^{nd} and 3^{rd} rows of matrix W' are obtained as follows:

$$w_{2,1} = a_{2,1} = 1, \ \ w_{2,2} = a_{2,2} = 6, \ \ w_{2,3} = a_{2,3} = 1, \ \ w_{2,4} = a_{2,4} = 2;$$
$$w_{3,1} = a_{3,1} = 2, \ \ w_{3,2} = a_{3,2} = 1, \ \ w_{3,3} = a_{3,3} = 7, \ \ w_{3,4} = a_{3,4} = 1.$$

Step 2: Solve two sets of 2×2 linear equations to obtain elements of first and last columns of matrix Z'.

For $j=1$:

$$\left.\begin{aligned} w_{22}z_{12}+w_{32}z_{13}&=a_{12}\\ w_{23}z_{12}+w_{33}z_{13}&=a_{13}\end{aligned}\right\}\rightarrow\left.\begin{aligned} 6z_{12}+z_{13}&=1\\ z_{12}+7z_{13}&=2\end{aligned}\right\}\rightarrow\begin{aligned} z_{12}&=\tfrac{5}{41}\\ z_{13}&=\tfrac{11}{41}\end{aligned}$$

For $j=4$:

$$\left.\begin{aligned} w_{22}z_{42}+w_{32}z_{43}&=a_{42}\\ w_{23}z_{42}+w_{33}z_{43}&=a_{43}\end{aligned}\right\}\rightarrow\left.\begin{aligned} 6z_{42}+z_{43}&=2\\ z_{42}+7z_{43}&=1\end{aligned}\right\}\rightarrow\begin{aligned} z_{42}&=\tfrac{13}{41}\\ z_{43}&=\tfrac{4}{41}\end{aligned}$$

Step 3: Update the remaining matrix.

For $j=1$ and $k=1,\ 4$

$$\left.\begin{aligned} a_{11}&=a_{11}-z_{12}w_{21}-z_{13}w_{31}\\ a_{14}&=a_{14}-z_{12}w_{24}-z_{13}w_{34}\end{aligned}\right\}\rightarrow\begin{aligned} a_{11}&=\tfrac{178}{41}\\ a_{14}&=\tfrac{20}{41}\end{aligned}$$

For $j=4$ and $k=1,\ 4$

$$\left.\begin{aligned} a_{41}&=a_{41}-z_{42}w_{21}-z_{43}w_{31}\\ a_{44}&=a_{44}-z_{42}w_{24}-z_{43}w_{34}\end{aligned}\right\}\rightarrow\begin{aligned} a_{41}&=\tfrac{20}{41}\\ a_{44}&=\tfrac{134}{41}\end{aligned}$$

The matrices Z' and W' are obtained as follows

$$Z'=\begin{bmatrix} 1 & \frac{5}{41} & \frac{11}{41} & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & \frac{13}{41} & \frac{4}{41} & 1\end{bmatrix}\ \text{and}\ W'=\begin{bmatrix} \frac{178}{41} & 0 & 0 & \frac{20}{41}\\ 1 & 6 & 1 & 2\\ 2 & 1 & 7 & 1\\ \frac{20}{41} & 0 & 0 & \frac{134}{41}\end{bmatrix}$$

The solution part is very similar to WZ method. Refer to the solution part of the numerical example as showed in WZ factorization.

4. X Factorization

Considering the matrix Z obtained from the numerical example of PIE.

$$Z=\begin{bmatrix} 6 & 1 & 1 & 2 & 1 \\ & \frac{254}{35} & \frac{55}{35} & \frac{16}{35} & \\ & & \frac{47879}{10541} & & \\ & \frac{16}{35} & \frac{20}{35} & \frac{184}{35} & \\ 1 & 2 & 1 & 1 & 6 \end{bmatrix}$$

The matrix X is obtained as $X=\begin{bmatrix} 6 & & & & 1 \\ & \frac{254}{35} & & \frac{16}{35} & \\ & & \frac{47879}{10541} & & \\ & \frac{16}{35} & & \frac{184}{35} & \\ 1 & & & & 6 \end{bmatrix}$.

Solve matrix Z' in order to update the vector b'.

Stage1: i=1

Step 1: The 2^{nd} and 4^{th} columns of matrix Z' are obtained by solving two sets of 2×2 linear equations.

For $j=1$:

$$\left.\begin{aligned} z'_{12}z_{22}+z'_{14}z_{42}&=z_{12}\\ z'_{12}z_{24}+z'_{14}z_{44}&=z_{14}\end{aligned}\right\}\rightarrow\left.\begin{aligned} z'_{12}\tfrac{254}{35}+z'_{14}\tfrac{16}{35}&=1\\ z'_{12}\tfrac{16}{35}+z'_{14}\tfrac{184}{35}&=2\end{aligned}\right\}\rightarrow\begin{aligned} z'_{12}&=\tfrac{532}{4648}\\ z'_{14}&=\tfrac{1772}{4648}\end{aligned}$$

For $j=5$:

$$\left.\begin{aligned} z'_{52}z_{22}+z'_{54}z_{42}&=z_{52}\\ z'_{52}z_{24}+z'_{54}z_{44}&=z_{54}\end{aligned}\right\}\rightarrow\left.\begin{aligned} z'_{52}\tfrac{254}{35}+z'_{54}\tfrac{16}{35}&=2\\ z'_{52}\tfrac{16}{35}+z'_{54}\tfrac{184}{35}&=1\end{aligned}\right\}\rightarrow\begin{aligned} z'_{52}&=\tfrac{1232}{4648}\\ z'_{54}&=\tfrac{777}{4648}\end{aligned}$$

Step 2: Update the remaining matrix of *Z*.

For $k=1$

$$z_{13}=z_{13}-z'_{12}z_{23}-z'_{14}z_{43}=1-\tfrac{532}{4648}(\tfrac{55}{35})-\tfrac{1772}{4648}(\tfrac{20}{35})=\tfrac{9798}{16268}$$

For $k=5$

$$z_{53}=z_{53}-z'_{52}z_{23}-z'_{54}z_{43}=1-\tfrac{1232}{4648}(\tfrac{55}{35})-\tfrac{777}{4648}(\tfrac{20}{35})=\tfrac{7938}{16268}$$

Stage 2: i=2

The 3^{rd} column of matrix Z' is obtained by

$$z'_{13} = z_{13} / z_{33} = \frac{9798}{16268}\left(\frac{10541}{47879}\right) = \frac{103280718}{491621572}$$
$$z'_{23} = z_{23} / z_{33} = \frac{55}{35}\left(\frac{10541}{47879}\right) = \frac{579755}{491621572}$$
$$z'_{43} = z_{43} / z_{33} = \frac{20}{35}\left(\frac{10541}{47879}\right) = \frac{210820}{491621572}$$
$$z'_{53} = z_{53} / z_{33} = \frac{7938}{16268}\left(\frac{10541}{47879}\right) = \frac{83674458}{491621572}$$

The matrices Z' is obtained as follows

$$Z' = \begin{matrix} 1 & \frac{532}{4648} & \frac{103280718}{491621572} & \frac{1772}{4648} & 0 \\ 0 & 1 & \frac{579755}{491621572} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{210820}{491621572} & 1 & 0 \\ 0 & \frac{1232}{4648} & \frac{83674458}{491621572} & \frac{777}{4648} & 1 \end{matrix}$$

Once matrix Z' is solved, vector b' is updated into b''. Then final solution of x is evaluated by solving 2 sets of 2×2 linear equations simultaneously.