8-2009

Parallel Processing Architecture for Solving Large Scale Linear Systems

Arun Nagari
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Arun Nagari entitled "Parallel Processing Architecture for Solving Large Scale Linear Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Itamar Arel, Major Professor

We have read this thesis and recommend its acceptance:

Fangxing Li, Hairong Qi

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

To the Graduate Council:
I am submitting herewith a thesis written by Arun Nagari entitled "Parallel Processing Architecture for Solving Large Scale Linear Systems". I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Itamar Arel, Major Professor

We have read this thesis and
recommend its acceptance:

Fangxing Li

Hairong Qi

Acceptance for the Council:

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

Parallel Processing Architecture for Solving Large Scale Linear Systems

A Thesis
Presented for the
Degree
Master of Science
The University of Tennessee, Knoxville

Arun Nagari
August 2009

Dedication

This thesis is dedicated to my parents, Govindarajulu Nagari and Lavanya Bharathi Nagari, and to my sister Silpa Nagari. For your love, support, and encouragment in helping me achieve my goals, I thank you.

Acknowledgments

I would first like to thank the Almighty for giving me the opportunity to work on this project and pursue my masters at University of Tennessee - Knoxville. There are many people whom I would like to acknowledge at this time for their help and support.

First of all, I must thank my advisor, Dr. Itamar Arel, for expending the time and effort to both get me started and guide me down the path towards the completion of my studies. I would not be where I am without his knowledge and understanding.

I would also like to express my thanks to all involved in my academic career for influencing me and helping me achieve my goals. I thank my teachers and professors throughout my studies for their support and teaching. Thank you to Junkyu Lee, Siddartha Janga Gautam, Ravi Chiravuri, Ankit Master, Derek Rose, and Cuibi Lu for assisting me in my work in various ways.

Finally, I thank my family for their love, and for always supporting and ecouraging me.

Abstract

Solving linear systems with multiple variables is at the core of many scientific problems. Parallel processing techniques for solving such system problems has have received much attention in recent years. A key theme in the literature pertains to the application of Lower triangular matrix and Upper triangular matrix(LU) decomposing, which factorizes an $N \times N$ square matrix into two triangular matrices. The resulting linear system can be more easily solved in $O(N^2)$ work. Inherently, the computational complexity of LU decomposition is $O(N^3)$. Moreover, it is a challenging process to parallelize. A highly-parallel methodology for solving large-scale, dense, linear systems is proposed in this thesis by means of the novel application of Cramer's Rule. A numerically stable scheme is described, yielding an overall computational complexity of $O(N)$ with N^2 processing units.

Contents

Chapter 1

Introduction

1.1 Background of Solving Linear Systems

Solving a system of linear equations is a fundamental problem in many scientific and engineering applications, including electric power network analysis, circuit simulation, aircraft design and structural analysis[8]. LU decomposition, in which a matrix is decomposed as a product of a lower and upper triangular matrix, is a common method used to solve a system of linear equations. In applications where matrices have thousands of elements, LU factorization demands exhaustive computations. When real time operations need to be performed, a reduced execution time is of the utmost importance. For this purpose, extensive research is directed towards the application of parallel processing techniques for LU factorization of linear systems[8]. Parallel supercomputers have achieved great success in solving computation-intensive problems, but have drawbacks such as high price/performance ratios, programming complexities, and maintenance costs[7].

On the other hand, with the rapid evolution of Field Programmable Gate Arrays (FPGA) into multi-million-gate System on a Programmable Chip (SOPC) computing platforms, it is now possible to integrate a large number of computational resources onto one silicon die, which speeds the process of concurrently solving linear systems[7][8][4]. Such efforts pertain to both fine-grained, as well as coarse, parallelism.

In this thesis, a novel approach for solving large-scale linear systems, using Cramer's Rule, is presented[6]. This approach yields a highly parallelizable design that can be directly realized in hardware. The computational complexity of the proposed method is $O(N)$, given N^2 processors.

1.2 Motivation

Power system distribution is one area in which solving a system of linear equations is crucial. The distribution networks of a power system are generally hierarchical with limited number of high-voltage lines transmitting electricity to connected local networks. Reliability, in this type of system, is ensured by feeding the highly interconnected local networks from multiple high voltage sources. Power grids have a graphical representation, expressed as matrices, in which graph nodes and matrix diagonal elements represent electrical bus, while graph edges, or non-zero off-diagonal elements, represent the electrical transmission lines. Load flow analysis is a critical task in power systems. Numerical methods such as the Jacobi, Gauss-Seidel, or Newton-Raphson are employed to solve the load flow equations[1]. The power system analysis is the basis from which the load flow calculations stems. The results of the calculations are used to estimate the operation of a power system under a set of conditions. Any of the varied input conditions can change since the systems are not static. In the event of scheduled or unscheduled equipment outages, the ability to quickly perform the load flow calculations allows engineers to be more confident about safety, reliability, and economic operation.

The existing technology is inadequate in that it does not allow real-time or dynamic analysis of these systems. The computational complexity can require hours to complete. Much research has been performed in this area and can be summarized as : developing an algorithm, improving software efficiency, and adding custom hardware parallelism. The impetuous for my research is to improve the performance of electrical power system applications in order to provide real-time power system control and decision making support by solving a system of linear equations.

1.3 Thesis Outline

Chapter 2 discusses several of the existing methods to solve a system of linear equations and their mathematical concepts. It also includes descriptions of the other methods currently being used to solve large scale linear systems. A brief explaination of the disadvantages of these methods is also addressed.

In Chapter 3 the algorithm developed in this thesis is discussed. It includes a detailed explaination of the algorithm with emphasis on the mathematical concepts used, its design, numerical stability, and computational complexity. A software simulation is helpful since a design can be more easily created in a software rather than in a FPGA, ensureing its verification and stability.

The algorithm was implemented in MATLAB for verification. The algorithm has been tested for stability with the fixed point arithmatic in MATLAB.

Chapter 4 focuses on the hardware architecture used to implement the developed algorithm. The algorithm has been implemented in VHDL and the Vertex2Pro family of FPGAs targeted to implement the hardware design. Each of the different components used has been discussed in detail. The synthesis results are provided. Finally Chapter 5 includes the results and comparision of the algorithm with other existing methods and algorithms.

Chapter 2

Literature Review

2.1 Overview

The purpose of a load flow computation is to determine the numerical values of the voltage magnitude and phase angles at load buses, voltage phase angle and reactive power at generator buses, and real and reactive power at the slack bus of a power system transmission network. Popular methods to perform the load flow are Jacobi, Gauss-Seidel and Newton-Rapson. Each method has its own advantages and disadvantages. The Jacobi and Gauss-Seidel methods use the known quatities of the power system and are easier to understand. Though their implementation is simple, the process needs significantly more iterations to converge to a solution. The Jacobi method requires more iterations than the Gauss-Seidel method. Newton's method is much complex than either the Jacobi or Gauss-Seidel method. The number of calculations per iteration is increased, requiring additional hardware. However, less iterations are needed to converge to a solution. The rate of convergence of Newton's method is quadratic whereas the rate of convergence for the Jocobi and Gauss-Seidel models is linear.

2.2 Existing methods

2.2.1 Gauss-Seidel and Jacobi Methods

The Jacobi and Gauss-Seidel methods are comparable. They have a similar output for any given input. The difference, however, is how newly calculated voltage data is handled in further calculations. The Jacobi method produces a solution vector for a constant set of inputs, unlike the Gauss-Seidel method, which utilizes the most readily existing value. The solution, via Gauss-Seidel, uses the following two equations, Eqs1 and 2

$$V_i^{(v+1)}=\frac{1}{Y_{ii}}\left[\left(\frac{S_i}{V_i^v}\right)^* - \sum_{k=l}^{k<i} Y_{ik}V_k^{v+1} - \sum_{k=i+1}^{NZ} Y_{ik}V_k^v\right] \tag{1}$$

$$S_i = P_i + jQ_i, \tag{2}$$

until the change in each element in the voltage vector, V, between iterations is less than 10^{-4} per unit. In Eqs.1 and 2, where Y is the bus admittance matrix (Y-bus), P is a vector of real power injections and Q is a vector of reactive power injection and the difference between generation and demand. All elements of the Y matrix and the V vector are complex quantities. First, the reactive power is estimated for the Gauss-Seidel iteration shown in Equation 3.

$$Q_i^v = V_i^v \sum_{k=1}^{NZ} (Y_{ik}V_k^v)^* . \tag{3}$$

When the voltage magnitudes and angles converge to a solution, reactive power becomes an output of the calculation.

2.2.2 Newton's Method

The objective of the power flow analysis is to determine steady-state voltages on all buses in a given network, and to derive from them the real and reactive power flows into each line and transformer. In most network buses, the active and reactive powers are specified, and can be evaluated by the following equations, for a network with N buses.

$$P_i = \sum_{k=1}^{N} |y_{ik}V_iV_k| \cos(\theta_{ik} + \delta_k - \delta_i) \tag{4}$$

$$Q_i = \sum_{k=1}^{N} |y_{ik}V_iV_k| \sin(\delta_i - \delta_k - \theta_{ik}) . \tag{5}$$

Here P_i, Q_i, and V_i are the active power, reactive power, and the complex voltage at bus i, respectively, with

$V_i = |V_i| \angle\delta_i$

$V_k = |V_k| \angle\delta_k$

$y_{ik} = |y_{ik}| \angle\theta_{ik} = g_{ik} + jb_{ik}$

$i, k \in [1, N]$

where y_{ik} is an element of the bus admittance matrix. If N_g is the number of voltage controlled buses in the system, then we have to solve $(2N - N_g - 2)$ equations. After expanding the obtained equations into a Taylor series, the following linear equations are produced. The linear equations need to be solved iteratively until the difference between $\Delta\delta$ and ΔV is smaller than the tolerance.

$$\begin{bmatrix} J^{11} & J^{12} \\ J^{21} & J^{22} \end{bmatrix} \begin{bmatrix} \Delta\delta \\ \frac{\Delta V}{|V|} \end{bmatrix} = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{6}$$

The Jacobian matrix, J, is evaluated at every iteration by the following equations

J11:

$$\frac{\partial P_i}{\partial \delta_j} = |V_i|\,|V_j|\,(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) \qquad j \neq i \tag{7}$$

$$\frac{\partial P_i}{\partial \delta_j} = -\,|V_i| \sum_{i=1}^{N} |V_j|\,(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) \qquad j \neq i \tag{8}$$

J12:

$$|V_j| \frac{\partial P_i}{\partial |V_j|} = |V_i|\,|V_j|\,(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij}) \qquad j \neq i \tag{9}$$

$$|V_j| \frac{\partial P_i}{\partial |V_j|} = 2\,|V_i|^2 g_{ii} + |V_i| \sum_{i=1}^{N} |V_j|\,(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij}) \qquad j \neq i \tag{10}$$

J21:

$$\frac{\partial Q_i}{\partial \delta_j} = -\,|V_i|\,|V_j|\,(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij}) \qquad j \neq i \tag{11}$$

$$\frac{\partial Q_i}{\partial \delta_j} = |V_i| \sum_{i=1}^{N} |V_j|\,(g_{ij}\cos\delta_{ij} + b_{ij}\sin\delta_{ij}) \qquad j \neq i \tag{12}$$

J22:

$$|V_i| \frac{\partial Q_i}{\partial V_j} = |V_i|\,|V_j|\,(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) \qquad j \neq i \tag{13}$$

$$|V_i| \frac{\partial Q_i}{\partial V_j} = |V_i| \sum_{i=1}^{N} |V_j|\,(g_{ij}\sin\delta_{ij} - b_{ij}\cos\delta_{ij}) - 2V_i^2 b_{ii} \quad j \neq i \tag{14}$$

To derive the difference at each iteration from the above linear equations, two types of methods are used. One is the direct method and the other is the iterative method. LU factorization, followed by forward reduction and backward substitution, is the most widely used method. In LU factorization, a sequence of Gaussian eliminations are applied to arrive at the form $PJ = LU$, where L is the lower triangular matrix, U is the upper triangular matrix and P is the Permutation matrix. The solution can then be obtained from the following equations

$$Lw = \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{15}$$

$$U \begin{bmatrix} \Delta\delta \\ \frac{\Delta V}{|V|} \end{bmatrix} = w \tag{16}$$

In direct methods, it is difficult to extract substatial parallalism while maintaining low inter-processor communication overhead. The computational complexity of LU factorization is shown by $O(M^3)$, where M stands for the order of the matrix. Hence, Newton's method is preferable to over direct methods.

2.3 FPGA Technology

FPGA stands for Field Programeable Gate Array, which are reusable logic devices. It is a combination of a number of Logic-cells. Each logic cell is a combination of a look-up table, a D flip-flop, and a 2-to-1 multiplexer. The look-up table is similar to a small RAM and typically has 4 inputs. Arrays of logic cells contain logic gates, along with memory blocks, to form the underlying flexible fabric for FPGA integrated circuits. Codes written in Hardware Description Language(HDL), such as VHDL, are synthesized and mapped to the devices, facilitating the designer to model the FPGA's functionality. FPGA device density could br found in a multitude of logic gates with operational clock rates in the tens of Megahertz in 2002. Today, the FPGA device densities are in millions of logic gates with synthesized logic capable of running at rates up to, and exceeding 500MHz. Apart from an increase in logic density, the inclusion of high performance embedded arithmetic units, and large amounts of memory, high speed processor cores have facilitated the growth of FPGA integrated circuits beyond a simple prototyping device. Today, high performance floating point computations are now feasible on a FPGA.

Chapter 3

Designed Algorithm

The algorithm employs Cramer's method for sloving a system of linear equations and Chio's Pivotal Condensation Process to develop a novel algorithm for solving large scale linear systems.

3.1 Concurrency in Solving Linear systems

3.1.1 Revisiting Cramer's Rule

Cramer's Rule expresses the solution to a system of simultaneous linear equations in terms of ratios of the determinants. For a linear system in the form $Ax = b$, where $A = [a_{ij}]$ is an invertible $N \times N$ matrix with $|a_{11}| > 0$, Cramer's Rule states that

$$x_i = \frac{\det(A_i)}{\det(A)}, (i = 1, 2, 3, ..., n) \tag{17}$$

where A_i is the matrix formed by replacing the i^{th} column of A by the column vector b[6]. Although Cramer's Rule provides an elegant way to solve a consistent system of equations, it is often viewed as highly impractical because it is computationally too expensive for large systems. Comparing Cramer's Rule to the Gaussian elimination method or other iterative methods accentuate this argument. Given that the most efficient method of calculating the determinant of an $N \times N$ matrix is $O(N^3)$, Cramer's Rule is generally perceived as an $O(N^4)$ method, limiting its scalability.

The widely acknowledged theory stated above is challenged in this thesis by introducing two relevant contributions. The first is a numerically stable and efficient method for calculating determinants in a parallel processing system, while the second is a framework for concurrently obtaining $\det(A_i)$. This thesis shows that, for a system with P parallel processing units, the overall computational complexity of the new method is reduced to $O(N^3/P)$. Moreover, the method's inherently small communication requirements render this method highly attractive for large-scale parallel processing platforms.

3.2 Proposed Architecture

3.2.1 Chio's Pivotal Condensation Process

The conventional approach of determining the computation of an $N \times N$ matrix is to express the elements of any one row or column, and the corresponding cofactors of the elements, as a linear combination of N determinants on the order $N-1$. For higher order matrices, the process becomes prohibitively lengthy. An alternative to the conventional method of determinant evaluation is to use a condensation method[3][2]. In condensation methods, an initial matrix of order N is successively reduced by one order per iteration until the basic order of 2×2 is reached. This method decreases the time required for determinant computation, as compared to the standard method.

Let $A = [a_{ij}]$ be an $N \times N$ matrix with $|a_{11}| > 0$, and D denote the matrix obtained by replacing each element a_{ij} in A by the term $\begin{vmatrix} a_{11} & a_{1j} \\ a_{i1} & a_{ij} \end{vmatrix}$, it can be shown that $|A| = |D|/(a_{11}^{N-2})$.

$$|A| = (1/a_{11}^{N-2}) \times \begin{vmatrix} \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1N} \\ a_{21} & a_{2N} \end{vmatrix} \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{31} & a_{32} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1N} \\ a_{31} & a_{3N} \end{vmatrix} \\ \vdots & \vdots & \cdots & \vdots \\ \begin{vmatrix} a_{11} & a_{12} \\ a_{N1} & a_{N2} \end{vmatrix} & \begin{vmatrix} a_{11} & a_{13} \\ a_{N1} & a_{NN} \end{vmatrix} & \cdots & \begin{vmatrix} a_{11} & a_{1N} \\ a_{N1} & a_{NN} \end{vmatrix} \end{vmatrix} \tag{18}$$

Let the reduction of an order k matrix to the order $k-1$ be defined as one *iteration*. Applying $N-2$ iterations will reduce the determinant matrix to 2×2. The above process exhibits attractive attributes in the context of parallel processing. First, it is clear that in each iteration, all 2×2 elements can be calculated independently. This suggests that given $(N-1)^2$ processing units, the process of calculating the determinant is reduced to $O(N)$. Moreover, and more important from a distributed-processing standpoint, the communication involved is simply a broadcast from the first column to all other columns. This suggests a communications complexity of $O(N)$ with clear independence between the right-most, $N-1$ columns.

3.2.2 Parallel Application of Cramer's Rule

Consider N simultaneous linear equations, $Ax = a_c$, of the form

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ a_{31} & a_{32} & \cdots & a_{3N} \\ \vdots & \vdots & \cdots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix} \times \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} a_{c1} \\ a_{c2} \\ a_{c3} \\ \vdots \\ a_{cN} \end{bmatrix} \tag{19}$$

The solution of such a system is addressed by means of an efficient utilization of Cramer's Rule. The core calculation performed by each processing unit will be the determinant of a 2×2 matrix; will remain the case throughout the process. Custom hardware can be designed to optimize the core calculation and further reduce aggregate computation time.

The reduction of computation time is achieved by eliminating the need to independently calculate the different numerators involved in Cramer's Rule. Performing Cramer's Rule traditionally requires computing N, numerators. Prior to applying the algorithm, normalization of the matrix elements is required. This is performed by dividing each of the first column elements with the highest element in that column. These elements are stored and can later be multiplied to obtain the final values. Understanding this method hinges upon realizing that each numerator can be found using the original matrix and the constant column (a_c). Extending this theory, half of the numerators of Cramer's Rule can be derived by replacing, in succession, half of the columns of the original matrix with (a_c). The remaining half of the numerators can be derived by replacing the other half of the original matrix.

This idea is exploited in the new method by mirroring the original matrix to yield two matrices, each of which is used to attain half of the variables. Chio's Condensation Process is used to reduce the matrices, the original and the mirrored matrix, until both are mirrored, creating four matrices. Each of the matrices will be used to find one fourth of the variables. This continues until each matrix is reduced to contain only information pertaining to one variable. The steps comprising the new method are explained below.

Step 1: Mirroring: The first step involves the creation of a new matrix by mirroring, horizontally, the original coefficient matrix. Recall that by interchanging two columns in a matrix, one must be negated for their determinants to be equal. The following illustrates the mirroring

$$\begin{bmatrix} \mathrm{a}_{11} & \mathrm{a}_{12} & \cdots & \mathrm{a}_{1N} \\ \mathrm{a}_{21} & \mathrm{a}_{22} & \cdots & \mathrm{a}_{2N} \\ \mathrm{a}_{31} & \mathrm{a}_{32} & \cdots & \mathrm{a}_{3N} \\ \mathrm{a}_{41} & \mathrm{a}_{42} & \cdots & \mathrm{a}_{4N} \\ \vdots & \vdots & \cdots & \vdots \\ \mathrm{a}_{N1} & \mathrm{a}_{N2} & \cdots & \mathrm{a}_{NN} \end{bmatrix} \Longrightarrow \tag{20}$$

$$\begin{bmatrix} \mathrm{a}_{1N} & \cdots & -\mathrm{a}_{12} & -\mathrm{a}_{11} \\ \mathrm{a}_{2N} & \cdots & -\mathrm{a}_{22} & -\mathrm{a}_{21} \\ \mathrm{a}_{3N} & \cdots & -\mathrm{a}_{32} & -\mathrm{a}_{31} \\ \mathrm{a}_{4N} & \cdots & -\mathrm{a}_{42} & -\mathrm{a}_{41} \\ \vdots & \cdots & \vdots & \vdots \\ \mathrm{a}_{NN} & \cdots & -\mathrm{a}_{N2} & -\mathrm{a}_{N1} \end{bmatrix}$$

Mirroring thus refers to multiple interchanges, resulting in one half of the new matrix being negated with respect to the original matrix. This is an integral step in parallelizing Cramer's Rule. The last half of the columns of the original matrix will be used to find the latter half of the variables. Similarly the last half of the columns of the mirrored matrix, or the first half of the columns of the original matrix, is used to obtain the first half of the variables. Note that by using Chio's process, every column is combined with the first column. Thus, the first column is the only one not combined with itself, resulting in its information being forfeited in the context of this architecture.

Step 2: Column Replacement: The second operation in the process is the replacement of the last column of the original and mirrored matrix with the constant column matrix, and storage of the replaced columns. Using the mirrored matrix as an example, the matrix, along with its stored columns, is shown below.

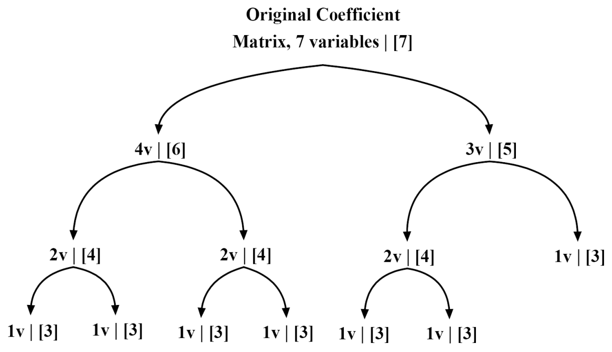


Figure1: Number of variables being solved for vs. matrix set size

$$\begin{bmatrix} a_{1N} & \cdots & -a_{12} & a_{c1} \\ a_{2N} & \cdots & -a_{22} & a_{c2} \\ a_{3N} & \cdots & -a_{32} & a_{c3} \\ a_{4N} & \cdots & -a_{42} & a_{c4} \\ \vdots & \cdots & \vdots & \vdots \\ a_{NN} & \cdots & -a_{N2} & a_{cN} \end{bmatrix} \blacktriangleright \begin{bmatrix} -a_{11} \\ -a_{21} \\ -a_{31} \\ -a_{41} \\ \vdots \\ -a_{N1} \end{bmatrix} \tag{21}$$

The above set of matrices, a larger matrix and its accompanying column matrix, is the fundamental unit and is repeated throughout the architecture. Hereafter, this unit will be referred to as a *matrix set*. The original coefficient matrix and the constant column matrix of the system of equations can be considered the first matrix set. Note that after mirroring, two matrix sets will be present, not just the one depicted above.

Step 3: Variable Assignment: The reduction of matrix sets occurs until a certain size matrix is reached. The appropriate size is determined by the number of variables being solved for by each matrix set. Following mirroring, each matrix set solves for half of the variables. In the case of an odd number N, such as $N = 9$, the original matrix set would be used to solve for $\lceil \frac{N}{2} \rceil = 5$ and the mirrored matrix set for $\lfloor \left(\frac{N}{2}\right) \rfloor = 4$ variables. To compensate for the loss of information inherent in the condensation process, the matrices are only reduced to size $\left(\frac{N}{2} + 2\right)$, 7×7 and 6×6, respectively. This type of calculation is utilized throughout the process to denote at which point reduction is terminated.

The tree diagram in Figure 1 illustrates the relationship between the number of variables being derived and the size of each matrix set, using an original 7×7 matrix as an example. The number followed by v is the number of variables solved for by that particular matrix set, while the number inside the brackets is the order of the matrix set. In this example, 7 total variables exist. The process continues until the final 3×3 matrix sets are produced. Note that the final number of variables in the branches equal 7, and only one variable is derived by any matrix set. All individual matrix sets are independent and can be computed as such.

Step 4: Reduction refers to reducing the individual matrix sets utilizing Chio's Process. The reduction of the accompanying column matrix using Chio's Process is equivalent to treating it as if it were an additional column of the larger matrix. However, the process requires the accompanying column matrix to be stored separately. Once reduced to the appropriate size, steps (1), (3), and (4) are repeated until all matrix sets are reduced to a 3×3 matrix. Replacement of the final columns of the matrices is performed only once – after the first mirroring step.

Chio's Process states that the a_{11} element in the reduced matrix should equal one. Due to the number of multiplications components involved, individual elements tend to expand and can eventually increase to large values. To counteract this problem, after each reducing iteration of the matrix set, the first row of both the large matrix and its accompanying column matrix is interchanged with the row that has the largest-magnitude first value $|a_{i1}|$, with one of the interchanged rows being negated. The negation ensures that the determinant remains the same after interchanging the rows. This process prevents matrix elements from increasing.

To force the first element of the matrix equal to one, the first column of the large matrix is divided by a_{11}. This also forces the multiplying factor, $(1/a_{11}^{N-2})$ equal to 1 in Chio's process because both the denominator (a_{11}) and numerator equal one. A record of each dividing factor (a_{11}) is kept for each branch of matrix sets and is multiplied back in the final step. When the matrix set is reduced to its minimal dimensions (3×3), a list of dividing factors $F = \{F_1, F_2, ..., F_i\}$ will have been formed.

After each mirroring step, two child matrix sets are produced from one parent set. The parent will have a factor list $F_P = \{F_{P1}, F_{P2}, ..., F_{Pi}\}$. Both child matrix sets will have the factor lists, F_A and F_B, respectively and both will begin with F_P. As new factors are added, each separate child set will add factors to its respective list, creating $F_A = \{F_{P1}, F_{P2}, ..., F_{Pi}, F_{A1}, F_{A2}, ..., F_{Ai}\}$ and $F_B = \{F_{P1}, F_{P2}, ..., F_{Pi}, F_{B1}, F_{B2}, ..., F_{Bi}\}$. This pattern continues through out the mirroring and reducing steps. Thus, all branches have lists that share common leading factors between sibling branches, and each unique branch adds its own unique factors.

Step 5: Solving for Variables: Each column of the original coefficient matrix can be traced through the reduction and mirroring phases to the final 3×3 matrices. This dictates which variable is solved for by each final matrix set. To solve for all but the first and last variables, the middle columns of the final 3×3 matrices are replaced with the *negation of* the accompanying column matrix, forming a matrix, Z. If the original coefficient matrix is A, then $\frac{\det(Z)}{\det(A)} \prod [F] = a_{ci}$. The first and last variables, a_{c1} and a_{cN}, are derived without replacing columns in the final matrices. The branch that is always mirrored contains the information of the first variable. If we let that branch's 3×3 matrix be denoted by M. then $\frac{\det(M)}{\det(C)} \prod [F] = a_{c1}$. Similarly, a_{cN} can be computed from the branch that is always the original matrix. This process yields $N + 2$ variables, two will be repeated and can be neglected. The following summarizes the process described:

1. Form a matrix which is the mirror image of the original matrix, A.

2. Replace the last column in each matrix with the constant matrix and store the column that has been replaced.

3. Determine the number of variables, V ,to be derived by each matrix.

4. Reduce each matrix to $(V + 2) \times (V + 2)$ using Chio's Process. Store the dividing factors in the list, F, for each branch. Repeat steps 1, 3, and 4 until all matrices are reduced to the minimal order 3×3.

5. Solve for the variables.

3.2.3 Computational Complexity

An approximation of the number of 2×2 determinants calculated for a given $N \times N$ matrix yields an expression of computational complexity. To simplify, references to determinants hereafter implies 2×2 determinants. To reduce a matrix, using Chio's Process, from an $N \times N$ to an $(N-1) \times (N-1)$ matrix requires $(N - 1)^2$ determinants to be calculated. Recall that an extra column matrix is carried in each matrix set. This requires an extra $N - 1$ determinant to reduce from an $N \times 1$ to an $(N - 1) \times 1$ column matrix. As such, each reduction of one matrix set requires $(N - 1)^2 + (N - 1)$ determinants.

The number of determinants depends on N and the number of mirroring steps performed. The computational complexity expression below was derived assuming that N is a power of 2. This simplifies the calculation since $N/2$ will also be a power of 2, and the number of mirroring steps is exactly $\log_2(N)$. Using an original 8×8 matrix set and Figure 2 as an illustration, the computational complexity expression is obtained. To reduce one 8×8 matrix set to 7×7, $\left(7^2+7\right)$ determinants are required. Similarly, to reduce that matrix set to 6×6, $\left(6^2+6\right)$ determinants are required. So, to reduce both 8×8 matrix sets to 6×6, $2\left(7^2+7+6^2+6\right)$ determinants are required. At this point, both 6×6 matrix sets are mirrored, creating four matrix sets instead of two. This process continues until the final 3×3 order for each matrix set is reached.

The number of determinants required for an $8{\times}8$ matrix, for example, is given by $2\left(7^2+7+6^2+6\right)+4(5^2+5+4^2+4)+8(3^2+3)=492$. Letting $\log_2(N)=G$, this expression can be compacted and generalized using the following equation

$$\sum_{h=\frac{N}{2}+2}^{N-1}\left[2\left(h^2+h\right)\right]+\sum_{k=2}^{G}\left[2^k\sum_{m=\frac{N}{2^k}+2}^{\frac{N}{2^{k-1}}+1}\left[m^2+m\right]\right] \tag{22}$$

Expanding the second summation to include the $k=1$ case and subtracting the resulting factor, the expression becomes

$$\sum_{k=1}^{G}\left[2^k\sum_{m=\frac{N}{2^k}+2}^{\frac{N}{2^{k-1}}+1}\left[m^2+m\right]\right]- \\ 2\left(N^2+N+(N+1)^2+(N+1)\right) \tag{23}$$

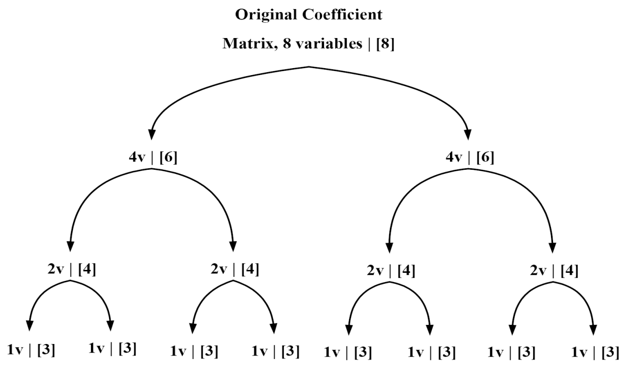


Figure 2 : Tree diagram illustrating the derivation of computational complexity

Rewriting the second summation using the identities $\sum_{i=1}^{t}\left[i^2\right] = \frac{t(t+1)(2t+1)}{6}$ and $\sum_{i=b}^{t}[i] = \frac{(t-b+1)(t+b)}{2}$, the expression becomes

$$\frac{1}{3}\sum_{k=1}^{G}\left[\frac{7N^3}{4^k} + \frac{18N^2}{2^k} + 11N\right] - 4N^2 - 8N - 4, \tag{24}$$

which is equal to exchanging the limits of the summation and negating the k exponents, such that

$$\frac{1}{3}\sum_{k=-G}^{-1}\left[4^k 7N^3 + 2^k 18N^2 + 11N\right] - 4N^2 - 8N - 4 \tag{25}$$

This facilitates the use of the identity $\sum_{i=b}^{t}[i^x] = \frac{x^{t+1}-x^b}{x-1}$. Finally, the expression becomes

$$\frac{7}{9}N^3 + 2N^2 - \frac{133}{9}N + \frac{11}{3}N\log_2(N) - 4, \tag{26}$$

suggesting a complexity of $O(N^3)$ This result is precisely accurate for any N that is a power of 2. The error for any other N is well below 0.01%, for $N > 256$. Because the application of this algorithm is for $N \gg 256$, the computational complexity expression above is sufficiently accurate.

Chapter 4

Hardware Architecture

The hardware architecture used for implementing the aforementioned algorithm is discussed in this chapter. The algorithm is coded in VHDL and processed on a XUP(Xilinx University Program) Vertex 2Pro board. A clock speed of 147MHz was achieved. The data is represented in IEEE754 format.

4.1 Overall block Diagram

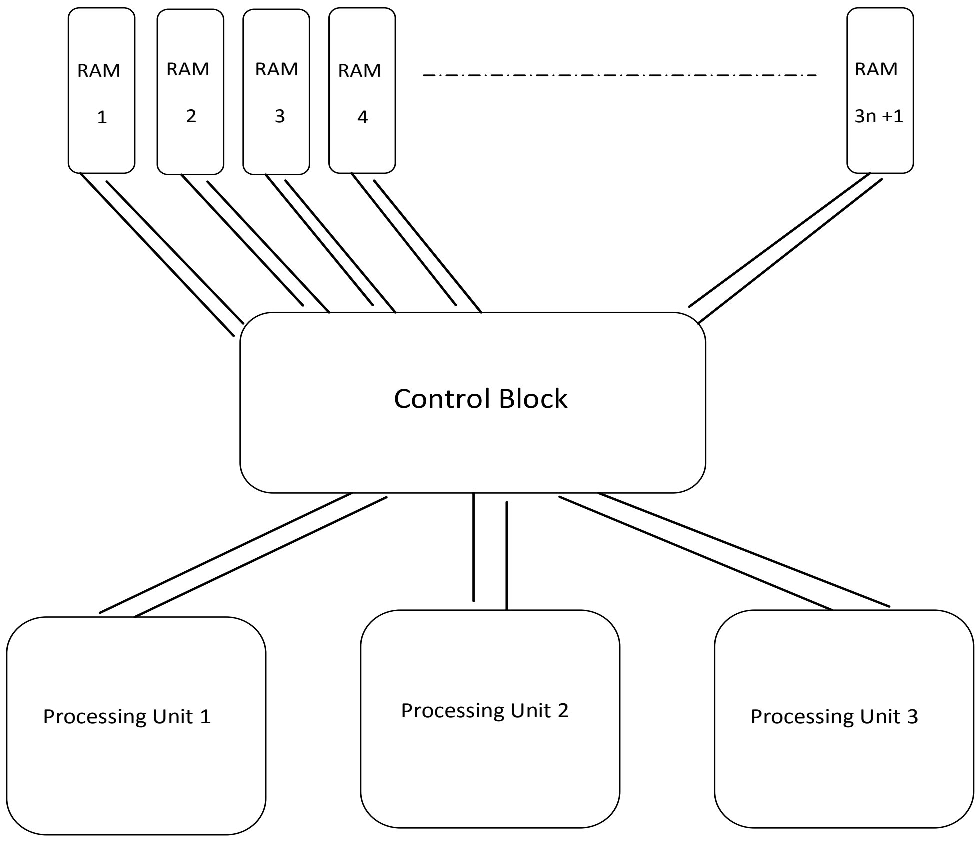


Figure3 : General Block Diagram of the Hardware Architecture

As shown in the above figure, the data is stored in different RAM blocks. Initially only $2N$ RAM blocks are required to store the original matrix and its image matrix. As the algorithm is executed, the data increases by n + 1 RAM blocks, thus $3N + 1$ RAM blocks are needed. The control block accesses the required data from the different blocks of RAMs and forward it to the processing units operating in parallel to the control block. The processed data is now written back to the RAM blocks, again in parallel. Three floating point multipliers, three floating point subtractors and two dividers exist in each processing unit.

4.2 Memory

The memory where the data is stored is divided into block RAMs. The number of RAM blocks can be found by an order N computation. Initially only $2N$ RAM blocks will be filled with data, but as the algorithm progresses the data expands and the correspoding data is stored in the remaining RAM blocks. The matrix data is divided into individual colomns and also stored in RAM blocks. This is so that the required data can be acessed in parallel. The data is initially converted to IEEE 754, 32-bit single-precision format.

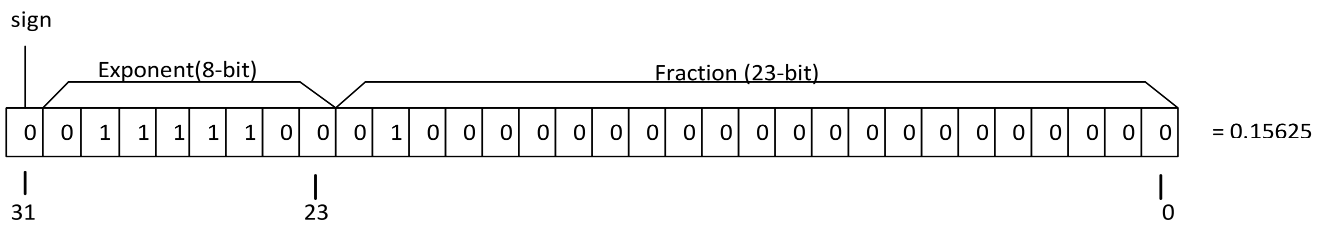
4.2.1 IEEE 754 32-bit single precision format

The IEEE 754 is the most frequently used standard for floating point computation. It defines formats for representing floating-point numbers, including four rounding modes and five exceptions. A single-precision, floating-point number is stored in 32 bits. The exponent is biased by $2^{8-1} - 1 = 127$, in this case (Exponents in the range -126 to $+127$ are representable. See the above explanation to understand why biasing is performed). An exponent of -127 would be biased to the value 0, but this is reserved to encode or show that the value is a denormalized number or zero. An exponent of 128 would be biased to the value 255, but this is reserved to encode an infinity or not a number (NaN).

For normalized numbers, the exponent is the biased exponent and the fraction is the significand without the most significant bit.

The number has the value, v:

$v = (-1)^{sign} \times 2^{exponent - exponent\tilde{\ }bias} \times (1 + fraction)$



32-bit single precision format

4.3 Control Block

The block diagram of a control block used in this architecture is shown below in Figure 5

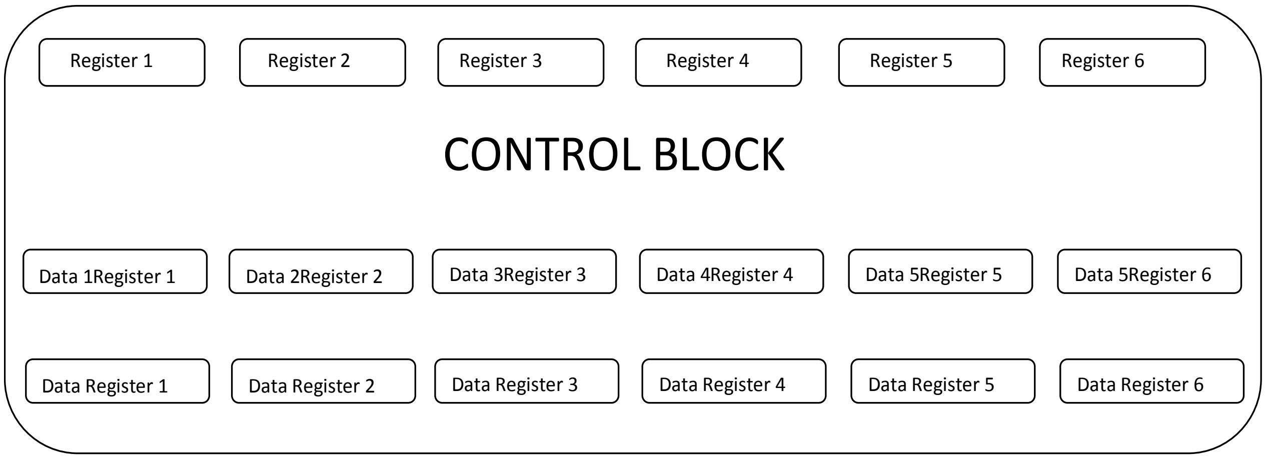


Figure 5 : Block Diagram of Control Block

Several registers in the control block contain the data to be multiplied and subtracted. The data from the control block is forwarded to the three processing units operating parallel to the control block. Computed data is then written back to the RAM blocks. The first register in the block diagram contains the data of the first row of the matrix and the remaining 12 registers contain the data of the remaining two elements of a $2X2$ determinant. The control block mediates the flow of data from the RAM blocks to the processing units, and back to the RAM blocks. The control block depends on a clock count to access the data from the different RAM blocks.

4.4 Components

4.4.1 Processing Unit

Three processing units are operating at the same time in this architecture. The block diagram of a single processing unit is shown below in Figure 6.

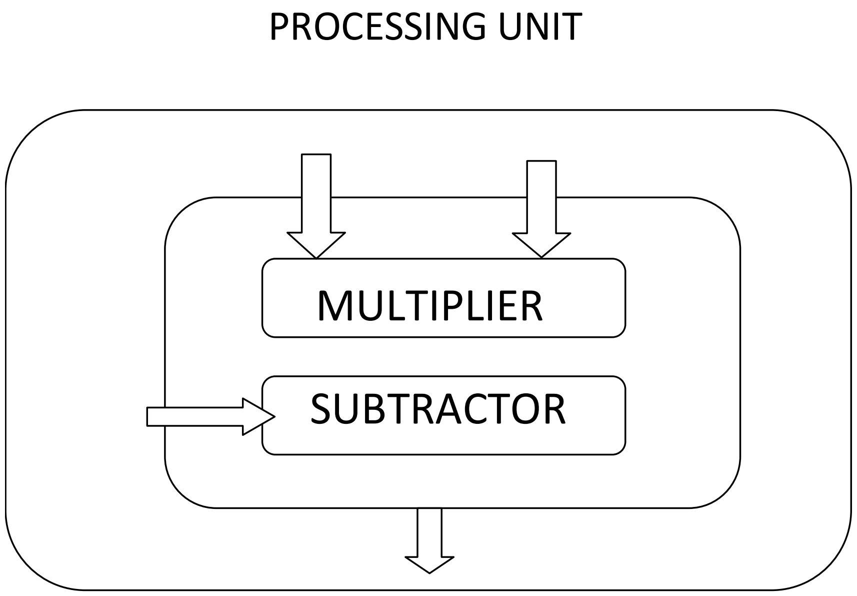
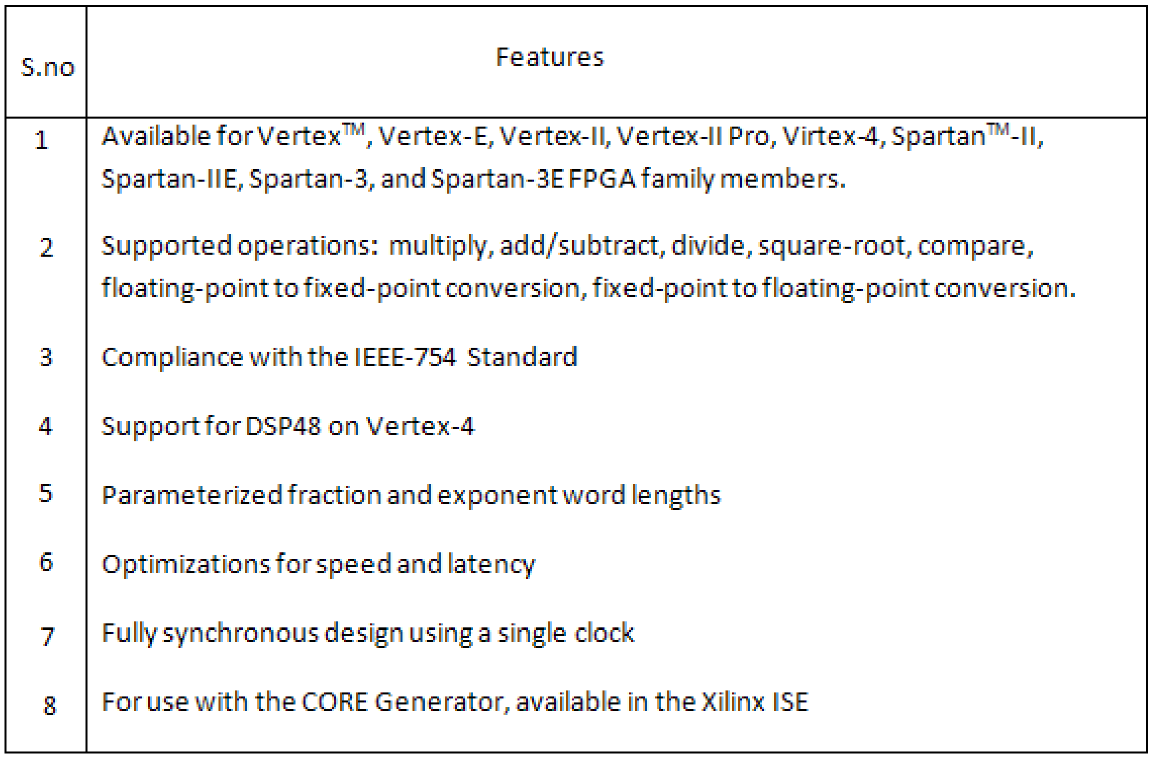


Figure 6 : Processing Unit

The processing unit consists of a multiplier and subtractor. The $2X2$ determinant is computed with a single multiplier and subtractor. The computed values are forwarded to the control block where the data is loaded and then back into the RAM blocks. The multiplier and subtractor are Xilinx coregen floating-point units.

4.4.2 Xilinx Floating Point Operator

The Xilinx Floating-point core provides engineers with a means to perform floating point computations on an FPGA. The core is customizable and allows the optimization of operations, wordlength, latency, and interface. Table 1 summarizes the features and supported operations of a Xilinx Floating-Point Operator.



S.no	Features
1	Available for Vertex^{TM}, Vertex-E, Vertex-II, Vertex-II Pro, Virtex-4, Spartan^{TM}-II, Spartan-IIE, Spartan-3, and Spartan-3E FPGA family members.
2	Supported operations: multiply, add/subtract, divide, square-root, compare, floating-point to fixed-point conversion, fixed-point to floating-point conversion.
3	Compliance with the IEEE-754 Standard
4	Support for DSP48 on Vertex-4
5	Parameterized fraction and exponent word lengths
6	Optimizations for speed and latency
7	Fully synchronous design using a single clock
8	For use with the CORE Generator, available in the Xilinx ISE

Overview:

The Xilinx Floating-point core permits a range of floating-point arithmetic operations to be performed on FPGAs. The operation is specified when the core is generated, and each variant has a common interface. This interface is shown in Figure 1. When a user selects an operation requiring only one operand, the B input is omitted.

IEEE-754 Support:

The Xilinx Floating-point core complies with majority of the IEEE-754 Standard. The deviations require a trade-off between resources versus functionality. Specifically, the core deviates in the following ways:

1) Non-standard Wordlengths

2) Denormalized Numbers

3) Rounding Modes

4) Signalling and Quiet NaNs

5)Non-standard Wordlengths

The Xilinx Floating-point core supports a greater range of fraction and exponent wordlengths than the wordlengths defined in the IEEE-754 Standard.

Standard formats commonly implemented by programmable processors include:

• Single Format - uses 32 bits, with a 24-bit fraction and 8-bit exponent.

• Double Format - uses 64 bits, with 53-bit fraction and 11-bit exponent.

Less commonly implemented standard formats are:

• Single Extended - wordlength extensions of 43 bits and above

• Double Extended - wordlength extensions of 79 bits and above

The Xilinx core support formats with fraction and exponent wordlengths beyond these standard wordlengths.

4.4.3 Parallel Comparator Logic

A parallel comparator block has been developed to increase the speed of the comparison process. Several single comaparators have been joined together to form a parallel comparator block. The block has as inputs of numbers that need to be compared inorder to produce their maximum number, as well as an enable signal. The enable signal is a 32-bit standard logic vector, which has all information regarding the count of numbers to be compared. This block not only provides the maximum number out of a set of numbers, but also provides the address of the maximum number so that the row replacement logic can be performed.

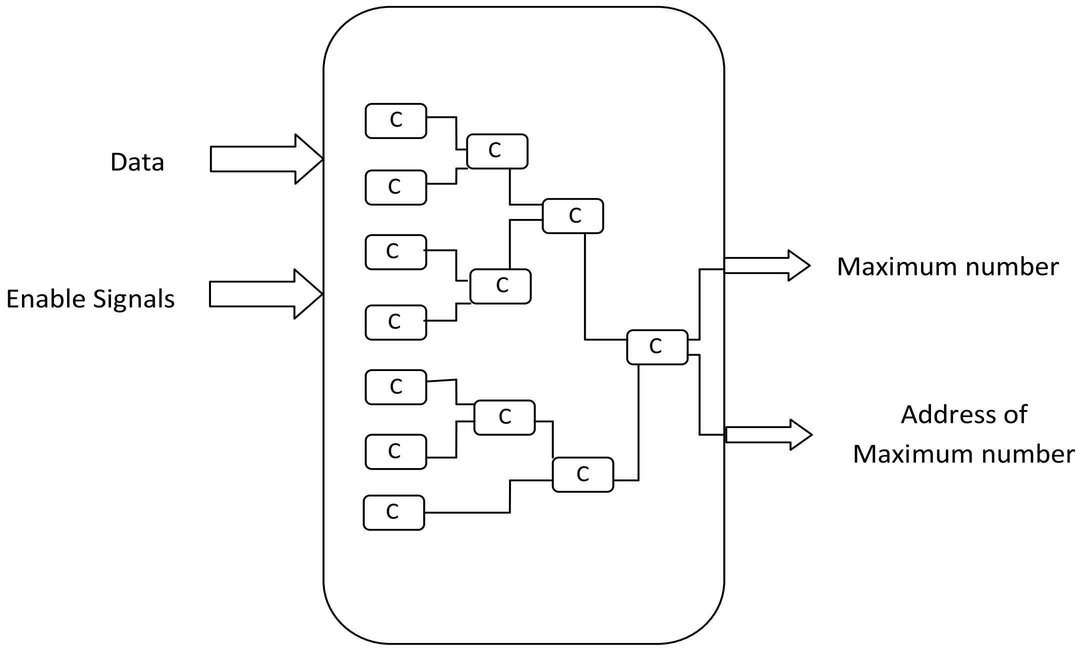
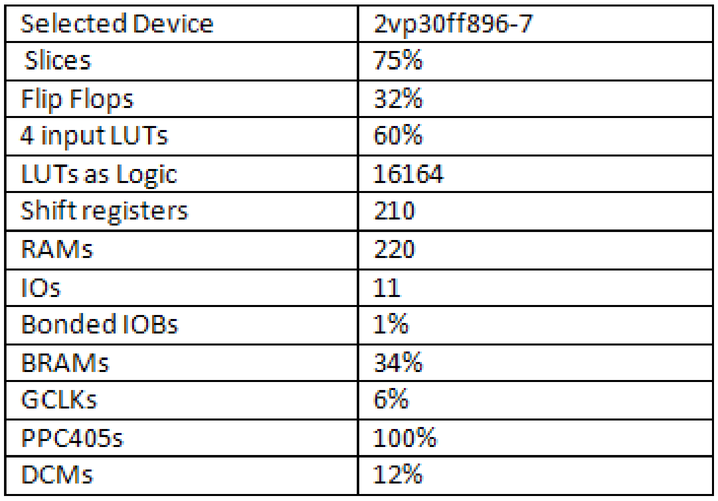


Figure 7 : Parallel Comparator

4.5 Synthesis Results

4.5.1 Device utilization summary

Table_1 : Device_Utilization_Summary



Selected Device	2vp30ff896-7
Slices	75%
Flip Flops	32%
4 input LUTs	60%
LUTs as Logic	16164
Shift registers	210
RAMs	220
IOs	11
Bonded IOBs	1%
BRAMs	34%
GCLKs	6%
PPC405s	100%
DCMs	12%

4.5.2 Timing Constrain

Default period analysis for Clock 'sys_clk_pin' Clock period: 8.541ns (frequency: 117.082MHz)

Minimum period: 8.541ns (Maximum Frequency: 117.082MHz)

Minimum input arrival time before clock: 1.746ns

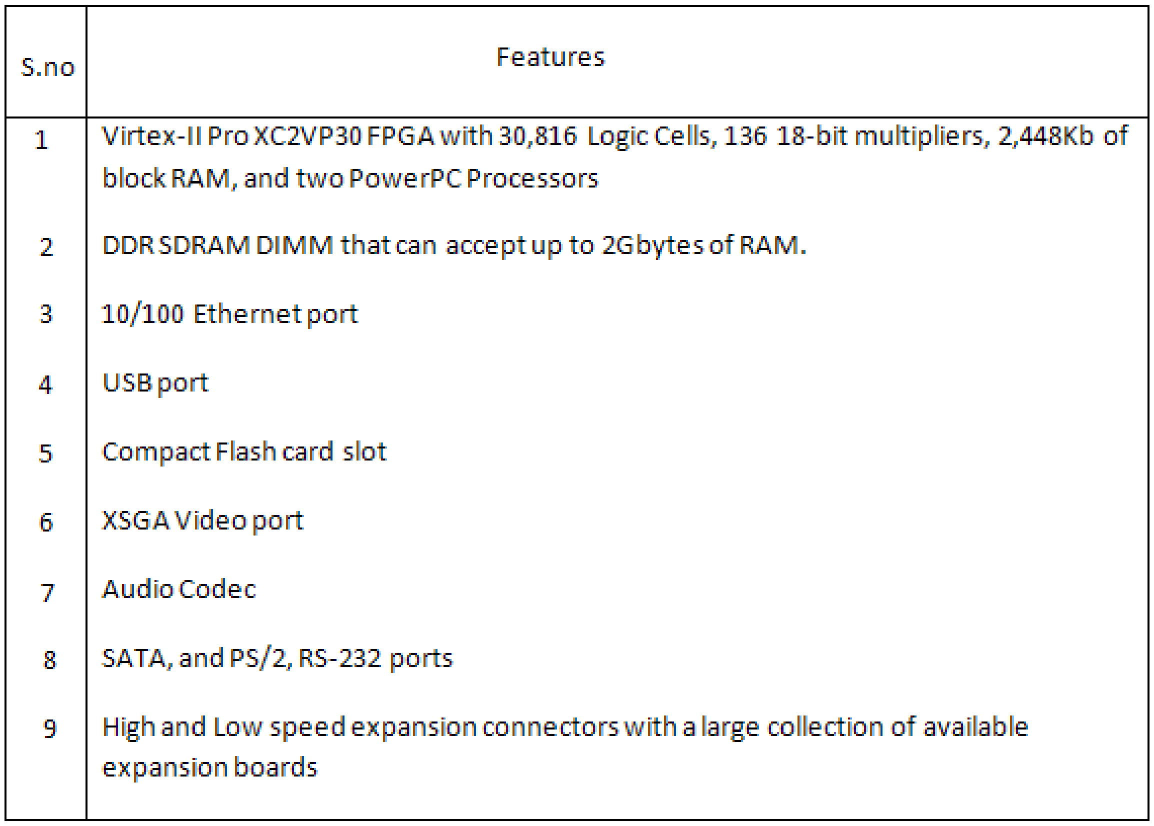
Maximum output required time after clock: 3.293ns

4.6 Hardware

Xilinx University Program (XUP) - XC2VP30 Board

The XUP is designed for engineering education. The XUP features are described as follows:

$Table\_2 \ : \ Xilinx\_University\_Program\_features$



S.no	Features
1	Virtex-II Pro XC2VP30 FPGA with 30,816 Logic Cells, 136 18-bit multipliers, 2,448Kb of block RAM, and two PowerPC Processors
2	DDR SDRAM DIMM that can accept up to 2Gbytes of RAM.
3	10/100 Ethernet port
4	USB port
5	Compact Flash card slot
6	XSGA Video port
7	Audio Codec
8	SATA, and PS/2, RS-232 ports
9	High and Low speed expansion connectors with a large collection of available expansion boards

One of the two PowerPC processors is used to provide input signals to the FPGA and to read data from the FPGA. The SDRAM is used to load an executable file. The USB port provides a communication mechanism between the host PC and the Power PC. For example, the Xilinx Microprocessor Debug (XMD) tool is used to load the executable file to the SDRAM and execute the file through the USB port. The RS-232 port is utilized to verify the results via a HyperTerminal.

Chapter 5

Conclusions

5.1 Comparison with LU Decomposition Method

5.1.1 Elimination of Zero Calculations for Sparse Matrices

In the proposed process, as well as in LU decomposition, several of the calculations yield a result of zero. In performing LU decomposition using systolic arrays, it is impossible to predict the zero results and hence, eliminate its occurrence. However, in the proposed process, any 2×2 matrix with a column or row of zeros yields a determinant of zero. Therefore, that particular determinant computation does not need to be fully executed. Simply checking for a row or column of zeros will allow the skipping of many determinant computations. This will greatly increase the overall speed of the process. Given that the prediction of zeros is not possible in LU decomposition using systolic arrays, the new process is significantly faster.

5.1.2 Number of Dividers

The proposed process includes a step where the first column of a given matrix is divided by its own first element. The largest matrix size is N, and to divide that matrix and its mirror requires, at most, $2N$ dividers. In comparison, LU decomposition using systolic arrays requires multiple dividers to parallelize the procedure. Since dividers are difficult and slow to implement, the difference in required dividers provides a significant increase in speed gain in the new process when compared to LU decomposition.

5.1.3 Pivoting

Pivoting is a critical issue in parallel LU factorization[8]. Pivoting is applied by arranging rows and/or columns of the matrix in order to choose the largest element as the pivot. This step maintains numerical stability during factorization. Pivoting in parallel LU decomposition increases complexity because the arrangement of rows and columns requires greater communication and synchronization between processors[8]. Pivoting also exacerbates the problem of load imbalances. The load imbalance issue can become more prominent if matrices are stored within dynamic data structures[1][8]. Generally, the pivoting process in the LU decomposition algorithm requires the maintainance of a separate matrix (composed of 1's and 0's) that keeps a record of the various row and column interchanges that occurred during the algorithm execution process. However, the method described in this paper requires an array of $2N$ elements, but it does not require a separate pivoting matrix and thus, reduces the memory requirement of the entire system.

5.1.4 Communication Overhead

One critical issue in a parallel implementation of LU factorization is data dependencies. If the matrix elements are stored among processors of a parallel processing platform, each processor has to communicate with the other processors to access the matrix elements. This not only effects the efficiency of parallel algorithms, but also increases the hardware complexity of the machines[8]. This has been a persistent problem in many LU decomposition algorithms. Bounded broadcast is one way to reduce the execution time in a LU decomposition methods[5]. In the proposed process, only the first column of elements in a given matrix is broadcast. The remaining calculations, namely the computations of each 2×2 determinant, are performed independently. Thus, less time is required to broadcast the needed information between the processing elements.

5.2 Conclusions

This thesis describes a methodology for solving large-scale linear systems on a parallel processing platform, using a variation of Cramer's Rule. The proposed architecture has a comparable computational complexity to that of LU decomposition while offering distinctive advantages in terms of parallelism and storage efficiency. Thus, given P parallel processing units, the computational complexity of the method is $O(N^3/P)$. Moreover, the communication overhead, a major challenge in parallel LU factorization schemes, is overcome by the proposed algorithm. For large-scale systems, this architecture can be implemented on hardware platforms, such as FPGAs, to yield a lower cost and performance ratio, when compared to that of supercomputers.

5.3 Relevant Publications

The A Parallel Processing Architecture for Solving Large Scale Linear Systems has appeared in the following publication:

- Arun Nagari, Itamar Elhanany, Ben Thomson, Fangxing Li, Thomas King, "A Parallel Processing Architecture for Solving Large Scale Linear Systems" The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications Las Vegas, Nevada, USA, July, 2008.

Bibliography

[1] O. Y. K. Chen, "A comparison of pivoting strategies for the direct lu factorization." [Online]. Available: citeseer.ist.psu.edu/293110.html

[2] E. Howard, *Elementary Matrix Theory*, 1966.

[3] H.Teimoori, A. M.Bayat, and E.Sarijloo, "A new parallel algorithm for evaluating the determinant of a matrix of order n." September 2005 European Conference on Combinatorics, Graph Theory and applications.

[4] N. Johnson.J.R, Nagvajara.P, "High-performance linear algebra processor using fpga," 2004, drexel University Philadelphia.

[5] D. Kim and S. V. Rajopadhye, "An improved systolic architecture for lu decomposition," pp. 231–238, 2006.

[6] L.Mirsky, *An Introduction to Linear Algebra.* Dover Publications, 1982.

[7] X. Wang and S. Ziavras, "Parallel lu factorization of sparse matrices on fpga-based configurable computing engines," 2004. [Online]. Available: citeseer.ist.psu.edu/wang03parallel.html

[8] X. Wang and S. G.Ziavras, "Parallel direct solution of linear equations on fpga-based machines," in *Parallel and Distributed Processing Symposium, 2003*, 2003.

Vita

Arun Nagari was born in Cudapha, India on Febuary 25, 1985. After graduating from All Saints High School in 2000, he attended The JNT University in Hyderabad. He received his Bachelor of Technology degree in Electronics and Communication Engineering in May of 2006. He then joined The University of Tennessee in August of 2007 to continue his studies, where he received his Master of Science degree in Electrical Engineering in August of 2009.