UNIVERSITY OF BIRMINGHAM

SOLUTION OF PARTIAL DIFFERENTIAL EQUATIONS USING RECONFIGURABLE COMPUTING

By

Jing Hu

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Electrical, Electronic and Computer Engineering
College of Engineering and Physical Sciences
The University of Birmingham
May 2010

The University of Birmingham

Electronic, Electrical and Computer Engineering

Degree of Doctor of Philosophy

in Electronic and Electrical Engineering

Solution of Partial Differential Equations using Reconfigurable Computing

Jing Hu

PhD Thesis

[redacted] [redacted]

[redacted]

Supervisor:	Dr Steven F. Quigley
	Prof Andrew Chan

ABSTRACT

This research undergone is an inter-disciplinary project with the Civil Engineering Department, which focuses on acceleration of the numerical solutions of Partial differential equations (PDEs) describing continuous solid bodies (e.g. a dam or an aircraft wing). Numerical techniques for solutions to PDEs are generally computationally demanding and data intensive. One approach to acceleration of their numerical solutions is to use FPGA based reconfigurable computing boards.

The aim of this research is to investigate the features of various algorithms for the numerical solution of Laplace's equation (the targeted PDE problem) in order to establish how well they can be mapped onto reconfigurable hardware accelerators. Finite difference methods and finite element methods are used to solve the PDE and they are characterized in terms of their operation count, sequential and parallel content, communication requirements and amenability to domain decomposition. These are then matched to abstract models of the capabilities of FPGA-based reconfigurable computing platforms. The performance of different algorithms is compared and discussed. The resulting hardware design will be suitable for platforms ranging from single board add-ins for general PCs to reconfigurable supercomputers such as the Cray XD1. However, the principal aim in this research has been to seek methods that perform well on low-cost platforms.

In this thesis, several algorithms of solving the PDE are implemented on FPGA-based reconfigurable computing systems. Domain decomposition is used to take advantage of the embedded memory within the FPGA, which is used as a cache to store the data for the current sub-domain in order to eliminate communication and synchronization delays between the sub-domains and to support a very large number of parallel pipelines. Using Fourier decomposition, the 32bit floating-point hardware/software design can achieve a speed-up of 38 for 3-D 256×256×256 finite difference method on a single FPGA board (based on a Virtex2V6000 FPGA) compared to a software solution implemented in the same algorithm on a 2.4 GHz Pentium 4 PC which supports SSE2. The 32 bit floating-point hardware-software coprocessor for the 3D tetrahedral finite element problem with 48,000 elements using the preconditioned conjugate gradient method can achieve a speed-up of 40 for a single FPGA board (based on a Virtex4VLX160 FPGA) compared to a software solution.

To my lovely daughter

ACKNOWLEDGEMENTS

I would like to thank:

- My two supervisors Dr. Steven F. Quigley and Prof. Andrew H.C. Chan, for all their outstanding supervision, valuable advice and great guidance throughout my PhD study;
- The CVCP for the Overseas Research Scholarship and University of Birmingham, School of Engineering Scholarship, for giving financial support for this research;
- My colleagues, Dr Sridhar Pammu, Lin A Win, Edward JC Stewart, Dr Abdellatif Abu-Issa, Phaklen Ehkan, Lin Zhang and so on, in room 435 (which was room 439 before the Department refurbishment), for sharing their research experience and knowledge;
- My friends, Dr Qing Liu, Hongwei Hu, Ronghua Zhu and so on, for making the time spent in Birmingham so enjoyable;
- My parents (Yueying Wang and Zhiqun Hu) and my husband (Dr Yebin Shi), last but far from least, for their unwavering support and understanding

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS & ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
BRAM	Block RAM
CE	Calculation Element
CG	Conjugate Gradient
CLB	Configurable Logic Block
CSR	Compressed Storage Row
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor/Processing
EBE	Element-by-Element
FDM	Finite Difference Method
FEA	Finite Element Analysis
FEM	Finite Element Method
FF	Flip-flop
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GPU	Graphic Processing Unit
GS	Gauss Seidel
HW	Hardware
I/O	Input/Output
IP	Intellectual Property
LUT	Look Up Table
ODE	Ordinary Differential Equation

PC	Personal Computer
PCG	Preconditioned Conjugate Gradient
PCI	Peripheral Component Interconnect
PDE	Partial Differential Equation
PE	Processing Element
PLD	Programmable Logic Device
RAM	Random-access Memory
SIMD	Single Instruction Multiple Data
SOR	Successive Over-Relaxation
SW	Software
SRAM	Static Random-access Memory
VHDL	Very high speed integrated circuit Hardware Description Language

Chapter 1

INTRODUCTION

1.1 Introduction

The Finite Difference Method (FDM) is one of the simplest and most straightforward ways of solving Partial Differential Equations (PDEs). The PDE is converted, by transforming the continuous domain of the state variables to a network or mesh of discrete points, into a set of finite difference equations that can be solved subject to the appropriate boundary conditions.

The Finite Element Method (FEM) is a widely used engineering analysis tool to obtain an approximate solution for a given mathematical model of a structure, as well as being used in the approximation of the solutions of partial differential equations. Finite Element Analysis (FEA) is widely applied to model a material or design that will be affected by various environmental factors, such as stress, temperature and vibration. FEA usually requires the solution of a large system of linear equations repeatedly. In industry, there is a requirement to reduce the time taken to obtain results from FEA software. In three dimensions, in order to reduce the memory requirements, iterative solvers are typically used to solve these systems.

Of the iterative algorithms available, the Conjugate Gradient method is one of the most effective iterative approaches to the solution of symmetric positive definite sparse systems of linear equations, as it converges in a finite number of steps.

FDM and FEM are both extremely computationally expensive, especially for 3-D problems. Much effort has therefore been made to explore the degree of parallelism on parallel computer systems in order to achieve a high speed-up of the simulation that is proportional to the number of processors used. Unfortunately, because of problems such as load balancing, synchronization and communication overheads, these systems could not achieve an ideal linear speed-up.

One of the approaches to high performance computing that is growing in credibility is the use of reconfigurable hardware to form custom hardware accelerators for numerical computations. This can be achieved through the use of Field Programmable Gate Arrays (FPGAs), which are large arrays of programmable logic gates and memory. FPGAs can be reconfigured in real time to provide large parallel arrays of simple processors that can co-operate with the host computer to solve a problem much faster and more efficiently. In recent years, Graphics Processing Units (GPUs) computing has also become a popular trend in high performance computing due to their massively parallel simplified CPU-units. However, for data-intensive numerical problems, such as FDM and FEM, the performance of GPUs can degrade because the memory accesses required by such numerical algorithms have a long latency which cannot always be hidden by pipelining.

FPGAs have evolved rapidly in the last few years, and have now reached sufficient speed and logic density to implement highly complex systems. FPGAs are being applied in many areas to accelerate algorithms that can make use of massive parallelism, such as bioinformatics [1], real-time image processing [2], data mining [3], communication networks [4], etc. The rapid improvement in hardware capabilities in the last few years has steadily widened the range of prospective application areas. One promising application area is the use of FPGA-based reconfigurable hardware to form custom hardware accelerators within standard computers for numerical computations. In this research, the term “reconfigurable computing” is used to refer to any use of an FPGA co-processor, not restricted only to run-time reconfigurable applications.

As a poor man’s supercomputer, FPGA co-processors are more expensive than general purpose GPUs, but they are still more easily affordable than expensive parallel computers. FPGAs within desk-top systems open a new window to low cost hardware acceleration. It is therefore desirable to explore how well the FDM and FEM can be mapped onto an FPGA-based reconfigurable computing platform.

1.2 Contribution of this thesis

This thesis presents a study of the use of reconfigurable hardware using FPGAs to accelerate implementations of the FDM and the FEM. An emphasis within this thesis is to find formulations that can perform well on low-cost platforms. In practice, this means seeking algorithms that have very low communications and synchronization overheads, as low-cost platforms tend to be characterized by low bandwidth for communications between FPGA boards and the host. The major contributions made by this work are as follows:

- A novel approach was taken to accelerate the finite difference method to solve a three-dimensional Laplace equation on a single FPGA. Both fixed-point arithmetic and floating-point arithmetic were investigated, and the performance of hardware based on 32-bit customised fixed-point arithmetic and 32-bit floating-point arithmetic was compared.
- A novel parallel hardware architecture for a preconditioned Conjugate Gradient solver was presented to solve the finite element equations using an element-by-element storage scheme.
- A domain decomposition technique has also been employed. Fourier decomposition was applied to the FDM to eliminate communication and synchronization delays between the sub-domains, whilst ensuring that the pattern of memory accesses is easy and efficient to implement. An element-by-element (EBE) approach was chosen for the FEM to efficiently handle sparse problems without requiring complicated data structures that inhibit the efficiency of hardware implementation.

The use of the element-by-element storage scheme also reduces the RAM requirement.

- Compared to an equivalent software solution on a 2.4GHz Pentium4 PC, a speed-up of 38 was achieved for solution of a 3-D Laplace equation using a 256×256×256 finite difference method using 32-bit floating-point arithmetic on the reconfigurable computing board with a Virtex2V6000 FPGA, whereas, a speed-up of 105 was achieved for a 64×64×64 FDM problem by using customized fixed-point arithmetic. For the finite element method, the 32bit floating-point hardware-software coprocessor for the 3D tetrahedral finite element problem with 48,000 elements using the preconditioned conjugate gradient method achieved a speed-up of 40 for a single FPGA board (based on a Virtex4VLX160 FPGA) compared to a software solution.
- Predictions of scalability are presented as to how well the hardware architecture would map onto larger FPGAs and larger number of FPGAs. The effects of data precision and additional resources are also considered.
- An analysis of speed-up has been carried out. It demonstrates that the performance of the hardware implementations is determined mainly by the available FPGA resources, with communication bandwidth and synchronization overheads between FPGAs and the host machine imposing relatively modest limitations.

1.3 Thesis organisation

Chapter 2 introduces the basic concepts of Partial Differential Equations and their solution. Direct methods and iterative methods are formulated, and their feasibility is considered.

Chapter 3 presents a brief review of reconfigurable computing in parallel processing.

Chapter 4 introduces the two most widely used general solution techniques of PDEs: the finite difference method (FDM) and the finite element method (FEM). The FDM is used to solve a 3-dimensional Laplace equation. Because the FDM can be extremely computationally expensive, especially when the number of grid points becomes large, Fourier decomposition is used to split the 3-D problem into a series of 2-D sub-problems, which can each be farmed out to a different FPGA (or fed sequentially through a single FPGA). The FEM, also widely used as an approximation for the solutions of PDEs, is also discussed in chapter 4.

Chapter 5 describes several hardware designs that were implemented onto a reconfigurable computing board. The first is a very compact design of an FDM implementation, which uses a customized 32-bit fixed-point arithmetic to fit the parallel computational units and working memory for an entire self-contained domain onto a single FPGA. The second is a more complex implementation, which extends the work to use floating-point arithmetic in order to avoid the poor numerical properties of fixed-point arithmetic. However, the impact of the larger logic requirements of floating-point arithmetic operators cannot be ignored: the number of parallel pipelines must be reduced due to the limitation of hardware resources. Furthermore, an element-by-element preconditioned Conjugate Gradient iterative solver for the solution of a 3D FE analysis has been implemented using 32-bit floating-point arithmetic on a single FPGA.

In chapter 6, the hardware implementations are compared with software implementations in terms of speed-up and numerical precision. Furthermore, the performance of several hardware implementations is compared based on logic requirements, clock rates, and error propagation etc. The floating-point hardware implementation of the finite difference method gives a factor of 24 speed-up compared to the software version, whereas the floating-point hardware implementation of the finite element method gives a factor of 40 speed-up. More sophisticated iteration schemes are examined in hardware and the data dependences are discussed. The Red-Black successive over-relaxation (SOR) method is judged to be a particularly attractive approach due its benign pattern of data dependencies and simple data path.

Chapter 7 describes how the hardware designs can be modified to enhance the performance (in terms of speed and efficiency) and discusses the bottlenecks of parallel computing. The whole of the reconfigurable computing system is considered: the limitations on the speed of communication between board and host, the memory resources and also the embedded microprocessors and multipliers on future FPGAs. A projection of how a typical system can be implemented and how well the system performs is presented. Based on the hardware implementations on Virtex II and Virtex 4 FPGAs, an estimate on how well the design can be scalable to take advantage of the properties of future FPGAs is also presented and discussed. Some of the main parameters that impact the performance of the hardware designs are discussed in Chapter 5. Also projections are made as to how much hardware will be needed and what level of speed-up could be expected.

Chapter 8 discusses the conclusions of the study, and gives some recommendations for possible future work.

Chapter 2

BACKGROUND

The history of research on partial differential equations (PDEs) goes back to the 18th century. One of the most important phenomena in the application of PDEs in science and engineering since the Second World War has been the impact of high speed digital computation [5]. Numerical analysis can be considered as a branch of analytical applied mathematics. There is a variety of numerical techniques for solving PDEs, such as the finite difference method [6], finite element method [7], finite volume method [8], boundary element method [9], meshfree method [10], and the spectral method [11]. The finite element method and finite volume method are widely used in engineering to model problems with complicated geometries; the finite difference method is often regarded as the simplest method [12]; the meshfree method is used to facilitate accurate and stable numerical solutions for PDEs without using a mesh.

This chapter provides a comprehensive guide to two numerical approaches to solution of partial differential equations, the finite difference method and the finite element method.

2.1 Introduction of Partial Differential Equations

Partial differential equations (PDEs) are used to formulate problems involving functions of several independent variables; the equations are expressed as a relationship between a function of two or more independent variables and the partial derivatives of this function with respect to these independent variables. The order of the highest derivative defines the order of the equation. PDEs are widely used in most fields of engineering and science, where many real physical processes are governed by partial differential equations [13]. Moreover, in recent years, there has been a dramatic increase in the use of PDEs in areas such as biology, chemistry, computer science and in economics.

PDEs fall roughly into these three classes, which are [13]

- ♦ Elliptic PDEs
- ♦ Parabolic PDEs
- ♦ Hyperbolic PDEs

If all of the partial derivatives appear in linear form and none of the coefficients depends on the dependent variable, then it is called a linear partial differential equation [13]. Otherwise, the PDE is non-linear if the coefficients depend on the dependent variable or the derivatives appear in a non-linear form. For example, consider the following two equations:

$$\frac{\partial f}{\partial t} = \alpha \frac{\partial^2 f}{\partial x^2} \qquad Eq.\ (1)$$

where x and t are the independent variables, f is the unknown function, and α is the coefficient, and

$$f\frac{\partial f}{\partial x}+\frac{\partial f}{\partial y}=0 \qquad \textit{Eq. } (2)$$

where x and y are the independent variables, and f is the unknown function.

Eq. (1) is the one-dimensional diffusion equation, which is a linear PDE, whereas, Eq. (2) is nonlinear because the coefficient of $\frac{\partial f}{\partial x}$ is the function f.

The 3 dimensional Laplace equation (3) for a function $\phi(x,y,z)$, which is a classical example of an elliptic linear PDE, describes the electrostatic potential in the absence of unpaired electric charge, or describes steady-state temperature distribution in the absence of heat sources and sinks in the domain under study in heat and mass transfer theory [14].

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}+\frac{\partial^2\phi}{\partial z^2}=0 \text{ or } \nabla^2\phi=0 \qquad \textit{Eq. } (3)$$

where x, y and z are the independent variables, ϕ is the unknown function, and ∇^2 is the Laplacian operator.

The equation is supplemented by initial and/or boundary conditions in order for a solution to be found. Laplace’s equation is a second-order homogeneous partial differential equation. (An equation is classified as homogeneous if the unknown function or its derivatives appear in each term). The Poisson equation (4) (which represents a steady state seepage problem with source term, an electrical field problem with source term, or a steady state heat transfer problem with heat sources) is the non-homogeneous form of the Laplace equation:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}=F(x,y,z) \text{ or } \nabla^2 u = F(x,y,z) \qquad \textit{Eq. (4)}$$

where the non-homogeneous term $F(x,y,z)$ is application dependent.

In electrostatics, the three-dimensional Poisson's equation (5) defines the relationship between the electrostatic potential and the electric charge density [14]:

$$\frac{\partial^2 \phi}{\partial x^2}+\frac{\partial^2 \phi}{\partial y^2}+\frac{\partial^2 \phi}{\partial z^2}=-\frac{\rho}{\varepsilon_0} \text{ or } \nabla^2 \phi = -\frac{\rho}{\varepsilon_0} \qquad \textit{Eq.}\ (5)$$

where ε_0 is the vacuum permittivity, ρ is the charge density and $F(x,y,z)$ is presented as a constant value $-\frac{\rho}{\varepsilon_0}$.

The appearance of the non-homogeneous term $F(x,y,z)$ can greatly complicate the exact solution of the Poisson equation, however, it does not change the general features of the PDEs, nor does it usually change or complicate the numerical method of solution [13]. Consequently, the solution of the linear homogeneous Laplace's equation, which is a common elliptic PDE, is considered in the whole thesis. All of the following discussions can be applied directly to the numerical solution of the Poisson equation, because the non-homogeneous term is simply added to the numerical approximation of the Laplace equation at each node or computational location.

Generally, various methods can be used to reduce the governing PDEs to a set of ordinary differential equations (ODEs). Unfortunately, only a limited number of special types of elliptic equations can be solved analytically. The most dramatic progress in PDEs has been achieved in the last century with the introduction of numerical approximation methods that allow the use of computers to solve PDEs in most situations for general geometries and under arbitrary external conditions, even though there are still a large number of hurdles to be overcome in practice.

The analytical solution of a two-dimensional elliptic equation is produced by calculating a function with the space co-ordinates x and y, which satisfies the partial differential equation at every point of area S which is bounded by a plane closed curve C, and satisfies certain conditions at every point on the boundary curve C as shown in Figure 1. Unfortunately, only a limited number of special types of elliptic equations can be solved analytically. In other cases, numerical approximation methods are necessary.

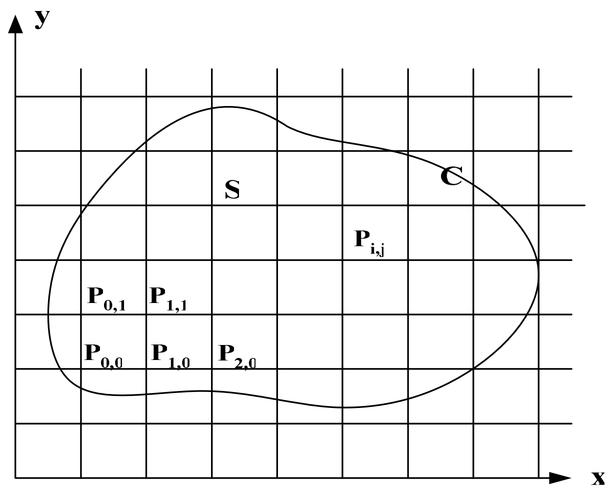


Figure 1 : 2-D Solution Domain for FDM.

In the following sections, the two widely used numerical approximation methods, the finite difference method (FDM) and the finite element method (FEM), will be introduced in the context of the Laplace's equation, which is a classical elliptic PDE that can be solved using relaxation methods [6].

2.1.1 Boundary conditions

There are three types of boundary conditions [13]:

1. ***Dirichlet boundary condition: the value of the function is specified.***
2. ***Neumann boundary condition: the value of the derivative normal to the boundary is specified.***
3. ***Mixed boundary condition: A combination of the function and its normal derivative is specified on the boundary.***

Figure 2 illustrates the closed solution domain $\Omega(x_1,x_2)$ and its boundary Γ. Equilibrium problems are steady-state problems in closed domains $\Omega(x_1,x_2)$ in which the solution $f(x_1,x_2)$ is governed by an elliptic PDE subject to boundary conditions specified at each point on the boundary Γ of the domain.

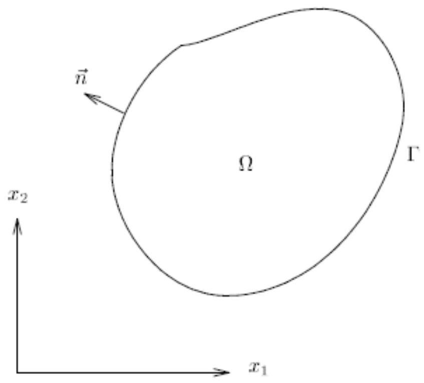


Figure 2 : Solution domain for an equilibrium problem.

2.2 Finite difference method

The Finite Difference Method (FDM) is one of the numerical approximation methods that are frequently used to solve partial differential equations [13]. The continuous physical domain is discretized into a discrete finite difference grid in order to approximate the individual exact partial derivatives in the PDE by algebraic finite difference approximations, then the approximations are substituted into the PDE to form a set of algebraic finite difference equations and, finally, the resulting algebraic equations are solved [13].

For technical purposes, FDMs can give solutions accurately, so they are as satisfactory as one calculated from analytical solutions [6]. Figure 3 shows a solution domain which is covered by a two-dimensional finite difference grid. The finite difference solution to the PDE is obtained at the intersections of these grid lines. Assuming that f is an unknown function of the independent variables x and y, the x-y plane is subdivided into sets of rectangles of sides Δx and Δy. The subscript i is used to denote the physical grid lines corresponding to constant values of x, where $x_i = i \cdot \Delta x$, and the subscript j is used to denote the physical grid lines corresponding to constant values of y, where $y_j = j \cdot \Delta y$. Additionally, a three-dimensional physical domain can be obtained by a three dimensional grid of planes perpendicular to the coordinate axes in a similar manner, where the subscripts i, j and k denote the physical grid planes perpendicular to the x, y and z axes. The grid point (i, j, k) represents location (x_i, y_j, z_k) in the solution domain.

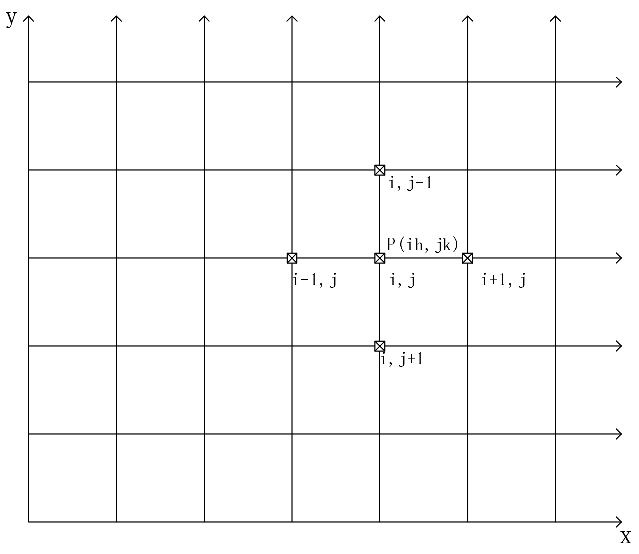


Figure 3 : Solution domain of 2D Laplace Equation and finite difference grid [6].

By using a second-order central difference approximation, the two-dimensional Laplace equation (3) becomes

$$\frac{f_{i+1,j}+f_{i-1,j}-2f_{i,j}}{\Delta x^2}+\frac{f_{i,j+1}+f_{i,j-1}-2f_{i,j}}{\Delta y^2}=0 \qquad Eq.\ (6)$$

Solving Eq. (6) for $f_{i,j}$ yields

$$f_{i,j}=\frac{f_{i+1,j}+f_{i-1,j}+\beta^2 f_{i,j+1}+\beta^2 f_{i,j-1}}{2(1+\beta^2)} \qquad Eq.\ (7)$$

where β is the grid aspect ratio $\beta=\frac{\Delta x}{\Delta y}$.

When the grid aspect ratio β is unity, i.e. $\Delta x=\Delta y$, Eq. (7) simplifies to

$$f_{i,j} = \frac{f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1}}{4} \qquad \textit{Eq. } (8)$$

or

$$f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1} - 4f_{i,j} = 0 \qquad \textit{Eq. } (9)$$

This can be solved by either of two approaches. The first approach assembles the contribution of each point into a global matrix, which can be written as follows:

$$\begin{bmatrix}
-4 & 1 & 0 & 0 & 0 & 1 & & & & & & \\
1 & -4 & 1 & 0 & 0 & 0 & 1 & & & & 0 & \\
0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & & & & \\
0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & & & \\
0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & & \\
1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & \\
 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & \ddots \\
 & & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & \ddots \\
 & & & \ddots & & & & \ddots & \ddots & \ddots & \ddots & \ddots \\
 & & & & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\
 & 0 & & & & 1 & 0 & 0 & 0 & 1 & -4 & 1 \\
 & & & & & & 1 & 0 & 0 & 0 & 1 & -4
\end{bmatrix}
\begin{bmatrix} f_{0,0} \\ f_{0,1} \\ f_{0,2} \\ f_{0,3} \\ f_{0,4} \\ f_{1,0} \\ f_{1,1} \\ f_{1,2} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}
=
\begin{bmatrix} b_{0,0} \\ b_{0,1} \\ b_{0,2} \\ b_{0,3} \\ b_{0,4} \\ b_{1,0} \\ b_{1,1} \\ b_{1,2} \\ \vdots \\ \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

Figure 4 : Pattern of global matrix of 3-D finite difference mesh

and the global matrix is then inverted by direct methods, such as Gauss Elimination. The second approach is to repeatedly apply Eq. (8) across all points in an iterative fashion until convergence is reached. Convergence is guaranteed as the global matrix in Figure 4 is diagonally dominant for the finite difference method. The direct methods and iterative methods for solving the system equations are presented in section 2.4 and 2.5.

Three-dimensional problems can be solved by including the finite difference approximations of the exact partial derivatives in the third direction. It is more complicated than two-dimensional problems as the size of the system of PDEs increases dramatically and the computation becomes expensive.

2.3 Finite element method

The finite element method is considered as the most general and well understood PDEs solution available. ***"The finite element method replaces the original function with a function that has some degree of smoothness over the global domain but is piecewise polynomial on simple cells, such as small triangles or rectangles."***[12]

The essential idea of the finite element method is to approach the continuous functions of the exact solution of the PDE using piecewise approximations, generally polynomials [15]. A complex system is constructed with points called nodes which make a grid called a mesh. This mesh is programmed to contain the material and structural properties which define how the structure will react to predefined loading conditions in the case of structural analysis. Nodes are assigned at a predefined density throughout the material depending on the anticipated stress levels of a particular area.

Thus, a basic flow chart of FEM is shown in Figure 5. First, the spatial domain for the analysis is sub-divided by a geometric discretization based on a variety of geometrical data and material properties using a number of different strategies. Generally, the solution domain is descretized into triangular elements or quadrilateral elements, which are the two most common forms of two-dimensional elements. Then, the element matrices and forces are formed; then the system equations are assembled and solved. Finally, the results are post-processed so that the results are presented in a suitable form for human interaction.

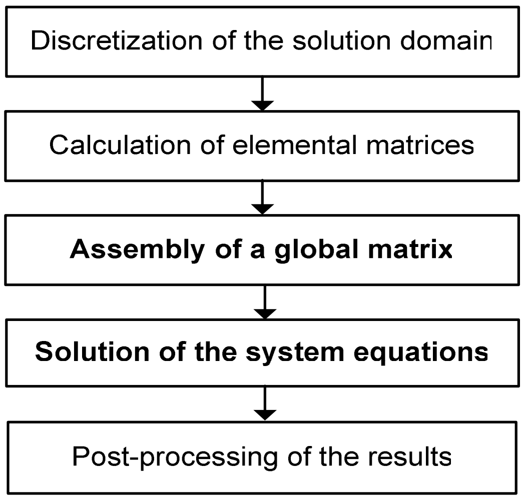

Discretization of the solution domain
Calculation of elemental matrices
Assembly of a global matrix
Solution of the system equations
Post-processing of the results

Figure 5 : Flow chart for the finite element algorithm.

The finite element method is used to model and simulate complex physical systems. The continuous functions are discretized into piecewise approximations, so the whole system is broken to many, but finite parts. However, the finite element method can be extremely computationally expensive and the available memory can be exhausted, especially when the number of grid points becomes large. The resulting system of equations may be solved either by direct methods or iterative methods such as Jacobi, Gauss Seidel, Conjugate Gradients or other advanced iterative methods such as the Preconditioned Conjugate Gradient (PCG) method, Incomplete Cholesky Conjugate Gradient method and GMRES. The direct method can provide the accurate solution with minimal round-off errors, but it is computationally expensive in terms of both processing and memory requirements, especially for large matrices and three dimensional problems because the original zero entries will be filled in during the elimination process. The global matrix for linear structural problems is a symmetric positive definite matrix. In general, it is also large and sparse. Consequently, the iterative methods are more efficient and more suitable for parallel computation but with lower accuracy (though a higher accuracy can be obtained at the expense of computational time) and the risk of a slow convergence rate.

Instead of assembling the global matrix, element stiffness matrices can be used directly for iterative solution techniques. An element-by-element approximation for finite element equation systems was presented in [16], and applied in the context of conventional parallel computing in [17]. This approach is very memory efficient (despite the fact that more memory is required than storing just the non-zero elements as in the CSR structure introduced in section 3.6), computationally convenient and retains the accuracy of the global coefficient matrix.

2.4 Direct Methods

Direct methods for solving the system equations theoretically deliver an exact solution in arbitrary-precision arithmetic by a (predictable) finite sequence of operations based on algebraic elimination. Gauss elimination, Gauss Jordan elimination and LU factorization are some of the examples of direct methods.

Consider the system of linear algebraic equations,

$$Ax = b \qquad \textit{Eq.}\ (10)$$

where matrix A is the coefficient $n \times n$ matrix obtained from the system of equations.

The Gauss elimination procedure is summarized as follows [18]:

1. Define the $n \times n$ coefficient matrix A, and the $n \times 1$ column vectors x and b,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \qquad \textit{Eq.}\ (11)$$

2. Perform elementary row operations to reduce the matrix into the upper triangular form

$$\begin{bmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{bmatrix} \qquad \textit{Eq.}\ (12)$$

3. Solve the equation of the n th row for x_n, then substitute back into the equation of the $(n-1)$ th row to obtain a solution for x_{n-1}, etc., according to the formula

$$x_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{n} a'_{ij} x_j \right) \qquad \textit{Eq. } (13)$$

The number of operations required by Gauss elimination method is $N = (n^3/3 - n/3) + n^2$. The Gauss-Jordan method, the matrix inverse method, the LU factorization method and the Thomas algorithm are variations or modifications of the Gauss elimination method. The Gauss-Jordan method requires more operations than the Gauss elimination method, which is $N = (n^3/2 - n/2) + n^2$. The matrix inverse method is simple but not all matrices have an inverse (there is no inverse matrix if the matrix's determinant is zero, i.e. singular, and no unique solution for the corresponding system of equations). The LU method requires $N = 4n^3/3 - n/3$ multiplicative operations, which is much less than Gauss elimination, especially for large systems.

When either the number of equations is small (100 or less), or most of the coefficients in the equations are non-zero, or the system domain is not diagonal, or the system of equations is ill conditioned[1] direct elimination methods would normally be used. Otherwise, an alternative solution method for the system of equations is an iterative method. This is desirable when the number of equations is large, especially when the system matrix is sparse [13].

[1] The condition number of a matrix is the ratio of the magnitudes of its maximum and minimum eigenvalues. A matrix is ill conditioned if its condition number is very large.

2.5 Iteration Methods

Beside the direct approach, the iterative approach is another common approach used to solve the system of PDEs. Direct methods are systematic procedures; whereas, iterative methods are asymptotical procedures with an iterative approach. Generally, direct methods are better for full or banded matrices, whereas, iterative methods are better for large and sparse matrices, especially for those arising from 3-dimensional PDEs. By assuming an initial guess solution vector $x^{(0)}$, iterative methods attempt to solve a system of equations by finding successive approximations and this procedure is repeated until the solution converges to some prescribed tolerance. If the matrix is diagonally dominant (i.e. the magnitude of the diagonal entry in every row of the matrix is larger than or equal to the sum of the magnitudes of all the other entries in this row), or extremely sparse, iterative methods are generally more efficient ways to solve the system of equations than direct methods. Stationary iterative methods and non-stationary iterative methods are the two main classes of iterative methods to solve a system of linear equations. Stationary iterative methods are called stationary because the same operations are performed on the current iteration vectors for every iteration (i.e. the coefficients are iteration-independent). Non-stationary iterative methods have iteration-dependent coefficients. In this sub-section, several stationary iterative approaches that are easy to solve and analyse will be presented.

2.5.1 Jacobi Method

The Jacobi method is the simplest algorithm for solving a system of linear equations. Due to the simultaneous iteration of all values, the Jacobi method is also called the method of simultaneous iteration, where all values of x^{k+1} depend only on the values of x^{k}. The process is then iterated until it converges.

Consider Eq. (10), written in index notation:

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \qquad (i = 1, 2, ..., n) \qquad \textit{Eq. } (14)$$

The solution vector x_i becomes

$$x_i = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j - \sum_{j=i+1}^{n} a_{i,j} x_j \right) \qquad (i = 1, 2, ..., n) \qquad \textit{Eq. } (15)$$

An initial solution vector $x^{(0)}$ is chosen. The superscript in parentheses denotes the iteration number, where zero denotes the initial solution vector.

Substituting the initial vector $x^{(0)}$ into Eq.(15), the first improved solution vector $x^{(1)}$ is then

$$x_i^{(1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(0)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(0)} \right) \qquad (i = 1, 2, ..., n) \qquad \textit{Eq. } (16)$$

After the k th iteration step, the solution vector $x^{(k+1)}$ is

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k)} \right) \qquad (i = 1, 2, ..., n) \qquad \textit{Eq. } (17)$$

This procedure is iterated until it converges to some specified criterion. Eq.(17) can be re-written in an equivalent way as

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^{(k)}\right) \qquad (i = 1, 2, \ldots, n) \qquad Eq.\ (18)$$

The Jacobi algorithm is simple and easy to implement on a parallel computing system, because the order of processing the equations is immaterial. However, from the point of view of numerical analysis, the Jacobi method has a poor convergence property in comparison to other iterative methods. Consequently, the Gauss Seidel and successive over-relaxation (SOR) methods are introduced in the following sub-sections.

2.5.2 Gauss Seidel Method

Compared to the independence among all values of $x^{(k+1)}$ in the Jacobi method, the Gauss Seidel method uses the most recently computed values of all x_i in all computations immediately. Thus, the solution vector $x_i^{(k+1)}$ is

$$x_i^{(k+1)} = x_i^{(k)} + \frac{1}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)}\right) \qquad (i = 1, 2, \ldots, n) \qquad Eq.\ (19)$$

Because the most recent values of x_i are used in the calculations, the Gauss Seidel method generally converges faster than Jacobi method [13].

2.5.3 Successive Over-Relaxation (SOR) Method

The successive over-relaxation (SOR) method is a numerical method used to speed up the convergence of the Gauss-Seidel method. Here, ω is a relaxation factor. The successive over-relaxation method is equivalent to the Gauss-Seidel method when $\omega = 1$. The Gauss Seidel procedure to compute the new value x_i^{GS}; then the successive over-relaxation update applies a scaled version of the Gauss-Seidel update, where ω is the scaling factor:

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{a_{i,i}}\left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)} \right) \qquad (i = 1, 2, ..., n) \qquad \textit{Eq.}\ (20)$$

Eq. (20) can be written in terms of the solution value x_i^{GS} from Gauss Seidel iteration method to yield

$$x_i^{(k+1)} = x_i^{(k)} + \omega\left(x_i^{GS(k+1)} - x_i^{(k)} \right) \qquad (i = 1, 2, ..., n) \qquad \textit{Eq.}\ (21)$$

Based on Ostrowski's Theorem [19]:

If A is symmetric and positive definite, then for any $\omega \in (0,2)$ *and any starting vector* $x^{(0)}$*, the successive over-relaxation (SOR) iterates converge to the solution* $Ax = b$.

When ω<1.0, the system of equations is under-relaxed. When ω=1.0, Eq. (21) becomes the Gauss Seidel method. When ω>2.0, the iterative method may diverge. When 1.0<ω<2.0, the system of equations is over-relaxed. The maximum rate of convergence is achieved for some optimum value of the over-relaxation factor ω, which lies between 1.0 and 2.0 [13]. The optimum value of ω depends on the size of the system of equations and the nature of the equations. However, there is no good general method for determining the optimal ω rather than searching by numerical experimentation for a optimal ω.

2.5.4 Red-black Successive Over-Relaxation

With the optimal choice of ω, the successive over-relaxation (SOR) iterative method is the recommended method, which converges much faster than the Jacobi and Gauss Seidel methods [13]. However, in both the Gauss Seidel and successive over-relaxation (SOR) methods, the update of the $(k+1)$th element depends on the update of the kth elements, as in Eq. (19) and Eq. (21). There is data dependency between elements and their neighbours. This causes a problem if one attempts to perform the updates in parallel, as the computation of Eq. (19) and Eq. (20) must wait until the required elements have been computed. In this sub-section, Red-black successive over-relaxation, a parallel scheme for the traditional successive over-relaxation method, is introduced.

Imagine that the two dimensional finite difference grids are coloured with a red and black checkerboard as in Figure 6. With this red-black group identification strategy, it is immediately apparent that the solution at the red square (R) depends only on its four immediate black neighbours. Similarly, the solution at the black square (B) depends only on its four red neighbours. The iteration scheme proceeds by alternating between update of the red squares and the black squares. So on an odd-numbered pass of the matrix, only the red squares are updated, using the previously computed values of the black squares. On an even-numbered pass, the black squares are updated using the newly computed values of the red squares. The use of the red-black scheme removes the requirement for each element to have immediate access to an updated value of some of its neighbours.

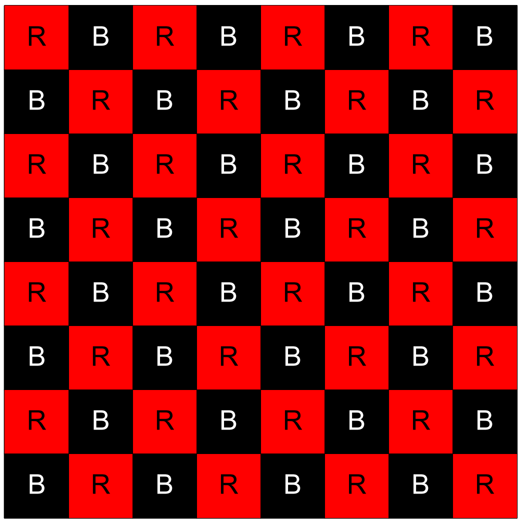


Figure 6 : Two-dimensional Red-Black Grid.

2.5.5 Convergence

The iterative methods attempt to solve the system of PDEs by finding successive approximations to the solution from an initial approximation. Convergence of an iteration method is achieved when the maximum relative error of the whole system is smaller than the tolerance ε required, i.e. $\left|\frac{x_i^{(k+1)} - x_i^{exact}}{x_i^{exact}}\right|_{\max} \leq \varepsilon$. Since the exact solution is unknown in most situations, the relative error at any step in the iterative process is based on the change in the values being calculated from one step to the next. Thus, convergence is assumed to be achieved when $\left|\frac{x_i^{k+1} - x_i^{k}}{x_i^{k}}\right|_{\max} \leq \varepsilon$. “The iterative methods require diagonal dominance to guarantee convergence” [13]. Some non-diagonally dominant problems can be rearranged by transforming to an equivalent diagonally dominant problem in a straightforward way, such as row interchanges. Some non-diagonally dominant system may converge for certain initial solution vectors, but convergence is not assured. Diagonal dominance requires that

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \qquad (i = 1, 2, ..., n) \qquad \textit{Eq.}\ (22)$$

with the inequality satisfied for at least one equation.

Based on the discussion in section 2.2, the system of finite difference equations arising from the five-point second-order central difference approximation of the Laplace equation is always diagonally dominant [13]. Therefore, convergence is assured when the iteration methods are applied on the finite difference approach to the solutions of PDEs.

2.6 Conjugate Gradient Method

The conjugate gradient (CG) method, named from the fact that it generates a sequence of conjugate vectors, is a non-stationary method for numerical solution. The method proceeds by generating vector sequences of iterates, residuals corresponding to the iterates, and searching the directions used in updating the iterates and residuals [20]. The residuals of the iterates are the gradients of a quadratic functional. Figure 7 demonstrate the performance of Conjugate Gradient (CG) for two variables.

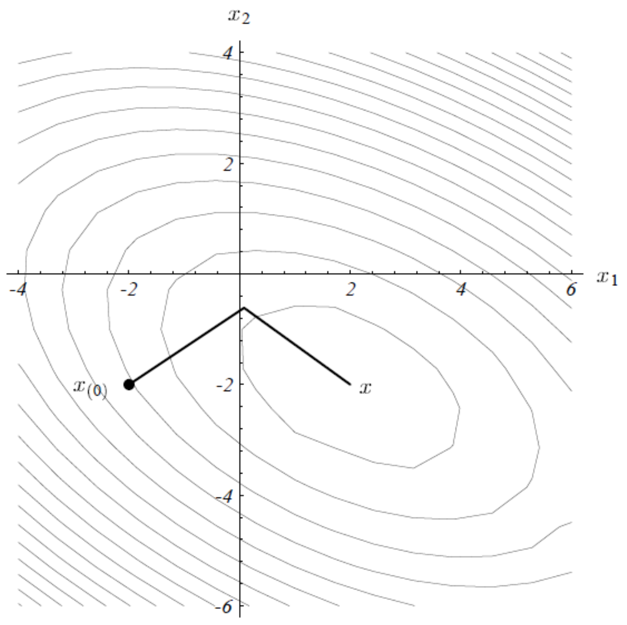


Figure 7 : The method of Conjugate Gradients [21].

The method of Conjugate Gradient is:

Consider Eq. (10),

$$d_{(0)} = r_{(0)} = b - Ax_{(0)} \qquad \textit{Eq.}\ (23)$$

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T A d_{(i)}} \qquad \textit{Eq.}\ (24)$$

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)} \qquad \textit{Eq.}\ (25)$$

$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A d_{(i)} \qquad \textit{Eq.}\ (26)$$

$$\beta_{(i+1)} = \frac{r_{(i+1)}^T r_{(i+1)}}{r_{(i)}^T r_{(i)}} \qquad \textit{Eq.}\ (27)$$

$$d_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} d_{(i)} \qquad \textit{Eq.}\ (28)$$

This method can be used effectively when the coefficient matrix A is :

- ♦ Symmetric (i.e. $A = A^T$)
- ♦ Positive definite, defined equivalently as:
    - All eigenvalues are positive
    - $x^T Ax > 0$ for all nonzero vectors x
    - A Cholesky factorization, $A = LL^T$ exists

Compared to relaxation iterative methods, the Conjugate Gradient method converges much faster when the global matrix A is symmetric and positive definite. For each iteration, the conjugate gradient method needs more operations and the global matrix needs to be assembled for finite element method, whereas, the relaxation methods are more straightforward as fewer operations are required per iteration and updates can be calculated directly without need to assemble the global matrix.

To speed-up convergence, preconditioning techniques are used to improve the spectral properties of the coefficient matrix A. If the coefficient matrix is ill-conditioned, it is useful to use a preconditioner to increase the convergence rate of the conjugate gradient method.

Other non-stationary iterative methods, like generalized minimal residual method and the biconjugate gradient method, are not considered in this study as their greater complexity makes efficient hardware implementation difficult.

2.7 Summary

The numerical solution of elliptic PDEs by the finite difference method and the finite element method is discussed in this chapter. The two general approaches to the solution of linear system of equations are presented. Direct methods obtain the exact solution in a finite number of operations, but they are not suitable for very large sparse matrices, especially 3-dimensional problems. Therefore, iterative methods will be considered in this research. In the following chapters, the history of the numerical analysis using parallel computers will be briefly reviewed. Furthermore, the parallel properties inside these methods will be discussed to obtain full utilization of the features of parallel computers.

Chapter 3

Review and Analysis of Parallel Implementations of Numerical Solutions

3.1 Introduction

The introduction of parallel microprocessor systems is a milestone in the history of scientific computing [22]. Based on Moore's Law, the number of transistors on microprocessors doubles roughly every two years; its corollary is that CPU performance should also double approximately every two years. In 1971, the first commercially available microprocessor Intel 4004 was launched, which employed 10 μm (i.e. 10000 nm) semiconductor process technology. Nowadays, Intel is looking at 11 nm as the next technology node after the 32 nm Clarkdale/Arrandale Westmere processor core launched in 2010.

However, due to power and technology issues, the increasing performance of microprocessors has lost steam in recent years [23]. Recently, many semiconductor manufacturers have turned from single-core to multi-core designs in order to increase the performance of their processors.

Multi-core processors and multi-CPU workstations lead the trend to develop dedicated parallel systems as opposed to the traditional single microprocessor system. The move to parallel computation creates interesting new challenges for software programmers, and operating system and compiler developers to adapt their sequential way of thinking to a parallel world. However, as with any disruptive technology, this also opens the door to other ways of organizing computation. A number of the major supercomputing vendors such as Cray and SRC, traditional high-end computing servers from IBM, SGI, Sun and more recent companies such as Linux Networx have begun to re-cast vast farms of commodity blade servers into FPGA-based hybrid systems [23]. Startups Xtreme Data and DRC have developed interface cards to bring high-performance computing to the masses, with methodologies to add an FPGA directly onto a commodity PC motherboard [23]. Coupled with increasing pressure to decrease costs and time-to-market, reconfigurable hardware can provide a flexible and efficient platform for satisfying the performance, area and power requirements [24].

In recent years, the use of a GPU (Graphic Processing Unit) to do general purpose (non-graphical) scientific and engineering computing has become a popular research topic, and has been applied to areas such as database operations [25], N-body simulation [26], stochastic differential equations [27] etc. The Tesla 20-series GPU is the latest CUDA architecture with features optimized for double precision floating point hardware support, with application areas including ray tracing, 3D cloud computing, video encoding, etc. [28]. Unlike the traditional complicated CPU, the GPU has a large number of simplified CPU-units but no cache. GPU performance can degrade massively if the memory access pattern required by the numerical algorithms is not a good match to the architecture of the GPU hardware.

Some work has been done to compare the performance of GPUs with FPGAs. In [29] and [30], for 2D convolution algorithms for video processing, the performance of the GPU was found to be not good enough due to the requirement of a high number of memory accesses. In [31], a comparative study of application behaviour on the performance and code complexity between GPUs and FPGAs was shown. Also in [32], the performance of applications, including Monte-Carlo simulation, a weighted sum algorithm and FFT, was analyzed.

This chapter discusses the promise and problems of reconfigurable computing systems. An overview of the chip and system architecture of reconfigurable computing systems is presented, as well as the application of these systems. The challenges and opportunities of future systems are discussed.

3.2 Parallel Computing

Why parallel? There are many computationally expensive problems in science and engineering; we want to solve them in a reasonable amount of time. So there are always pressures to alleviate the extremely time consuming nature of simulations. By using the latest and fastest processors, a shorter computation time should, in principle, be achieved. However, due to problems with thermal management and reliability, the clock speed of new generations of processors is no longer rising at a significant rate. Additionally, Gordon Moore, the inventor of Moore's Law, said that Moore's Law is dead because that transistors would eventually reach the limits of miniaturization at atomic levels [33]. Therefore, the introduction of parallel processing was a milestone in the history of computing as it provides a way to increase performance without increasing clock speed. Now parallel processing is used everywhere in real world computational applications, such as atmospheric science, mechanical engineering, chemistry, genetics, etc.

Parallel computing, literally, is to process multiple tasks simultaneously on multiple processors. Simply, a given task is divided into multiple sub-tasks, which are then solved concurrently. The most popular way to evaluate the performance of a parallel machine is to compare the execution time of the best possible serial algorithm for the problem with the execution time of the parallel algorithm. Speed-up describes the speed advantage of the parallel algorithm, as in Eq. (29).

$$Speed\_up(n) = \frac{\text{the execution time of the fastest sequential algorithm}}{\text{the execution time of the parallel algorithm with } p \text{ processors}} \qquad Eq.\ (29)$$

where n represents problem size.

Factors such as synchronization and communication overheads prevent parallel systems from achieving linear speed-ups, so in practical systems the achievable speed up will not scale linearly with *p*.

3.3 FPGAs & Reconfigurable Computing Systems

The first-ever Field Programmable Gate Array (FPGA) was invented by Ross Freeman in 1985 [34]. FPGAs are arrays of reconfigurable logic blocks connected by reconfigurable wiring. These form the underlying flexible fabric that can be used to build any required function. In the past decade, the capabilities of FPGAs have been greatly improved; modern FPGAs also contain highly optimized dedicated functional blocks, e.g. hardware multipliers, memory blocks, and even embedded microprocessors. From their origins as simple glue-logic to the modern day basis of a huge range of reprogrammable systems, FPGAs have now reached sufficient speed and logic density to implement highly complex systems. The latest FPGA devices have multi-million gate logic fabrics capable of achieving frequencies up to 600MHz, large on-chip memory and fast I/O resources [35].

An approach to high performance computing that is growing in credibility is the use of FPGA-based reconfigurable hardware to form a custom hardware accelerator for numerical computations [36-40]. FPGAs are now widely applied in many areas that can make use of massive parallelism. For the right type of application, a reconfigurable computer can rival expensive parallel computers that are normally used to accelerate computationally expensive algorithms. SRAM-based FPGAs have become the workhorse [41] of many computationally intensive applications. This is a result of the rapid improvement in FPGA hardware of the last few years (Table 1). Due to technology advances of FPGAs, it has become possible to make an increasing level of hardware and software co-operative system available. These co-operative systems are used into a wide range of applications, not only high performance computing, but also everyday technology like mobile communication [42]..

Table 1 : Xilinx Devices Comparison [35].

Date (Year)	Device (Series)	Capacity (Gates)	Max. Clock (MHz)
1994	XCV3000	20K	30
1999	Virtex 1	1M	120
2000	Virtex Ⅱ	10M	200
2002	Virtex Ⅱ Pro	20M	300
2005	Virtex 4	35M	500
2006	Virtex 5	55M	550
2009	Virtex 6	150M	600

Floating-point operations on FPGAs are now approaching a level that is competitive compared to floating-point operations on standard microprocessors. Reconfigurable computing has become an attractive option for numerical computations because of its great performance and flexibility. The most obvious benefit of using FPGAs is performance. The capability of FPGAs has grown dramatically making 64 bit floating point operations feasible and thus their use in large scale scientific and high performance computing intriguing. Thus, FPGA-based reconfigurable hardware/software co-processors are suitable for use for algorithm acceleration, not only with low cost, but also with a great deal of functional flexibility [43].

Reconfigurable co-processors use FPGAs to accelerate algorithm execution by implementing compute-intensive calculations in the reconfigurable substrate. The strength of a reconfigurable processor is the ability to customize hardware for the specific requirements of the system. Reconfigurable hardware takes care of the regular kernels of computations that are responsible for a large fraction of execution time and energy, while the main processor executes all the remaining original algorithm tasks and manages reconfigurable specific tasks like reconfiguration, reading/writing memory, and I/O control. Put simply, algorithm execution is partitioned between reconfigurable hardware and the main processor.

Reconfigurable computing has many advantages over CPUs and ASICs (Application-Specific Integrated Circuits). First of all, it is possible to get greater functionality with simpler hardware, as all of the logic functions can be stored in the memory but not required to be present in FPGA at all times. Secondly, the logic functionality can be customized to perform exactly the operation desired, with higher computation density than CPUs. Last but not least, the logic design is flexible, as the functionality can be reconfigured based on the type of application desired by using the same resources.

FPGA clock speeds tend to be an order of magnitude slower than the microprocessor [44]. As a general rule therefore, the FPGAs should be performing a very large number of arithmetic operations on each clock cycle in order to give a significant speed-up. This is done by creating many deep pipelines of arithmetic operations, and operating many such pipelines in parallel. The term *pipeline* refers to a set of data processing stages connected and executed in series. A deeper pipelined architecture has more stages in the pipeline and fewer logic gates in each stage, and therefore, more such computational pipelines can be operated in parallel to increase the throughput of the operators. Within fully pipelined designs, the computation latency can be fully overlapped so that the results can be generated on every clock cycle. However, very high levels of parallelism may lead to situations where the solution is limited by memory bandwidth; if there are too many parallel pipelines, then they simply become starved of data. Without enough data feeding the hardware, the hardware can not perform computations continually. Another concern with FPGA-based approaches is the time taken to transfer the data onto and off of the reconfigurable co-processor. These data transfer times may nullify the speed gains achieved by the parallel processing within FPGAs. One more drawback of reconfigurable computing design is that the design tool productivity is low compared to other approaches, so it takes a long time to come up with a design for a particular application.

3.4 Parallel Implementation

In this section, some of the previous attempts to parallelize the FDM and the FEM on different multiprocessor systems are discussed. These attempts are presented in chronological order, so that they can also be evaluated in terms of the computational resources available at that time.

3.4.1 Parallel FDM Implementations

The finite difference method is one of the most powerful and widely used approaches to the solution of Partial Differential Equations, due to its good accuracy and flexibility [13]. However, the finite difference method can be enormously computationally expensive, especially when the number of grid points becomes large. Much effort has been made to find an efficient solution to FDM solutions for PDEs.

Some earlier efforts on the parallel implementation of the algorithms for discrete elliptic equations were reviewed [45]. These include the solution of Poisson's equations implemented on an Illiac IV in 1972 [46], and on the TI-ASC [47] and the solution of Laplace's equation in 1974 [48].

Long [49] describes a solution of the Boltzmann equation to model a one-dimensional shock wave structure, a boundary layer, and general 3-D flow fields by using the Bhatnagar-Gross-Krook (BGK) model combined with a finite difference scheme. The optimized algorithm sustained 61 GFLOPs on a 1024-node CM-5.

In 1994, a parallel algorithm Simple Parallel Prefix (SPP) was proposed for the compact finite difference scheme by Sun [50]. Compared with the traditional finite difference method, the compact finite difference scheme has the features of higher accuracy with smaller difference stencils and leads to more accurate approximations due to the smaller coefficients of the truncation error. The implementation of the simple parallel prefix algorithm on a 16k processing elements MasPar MP-1 SIMD parallel computer achieved a speed-up of 1000 compared to the best sequential algorithm. In addition to the good performance on the SIMD machines, the algorithm also performed better than the sequential algorithm on a Cray 2.

Several techniques used to optimize a finite difference algorithm on the CM5, a distributed memory parallel processing system by Thinking Machines Corporation, were presented in [51]. The processing nodes in the CM5 interact through the control and data networks and all activities are coordinated by a control processor. A fat-tree topology data network provides point-to-point communication between processing nodes at a rate of at least 5 Mbytes per second. The optimized algorithm runs almost 2 times faster than the global CMFortran code.

A domain decomposition algorithm based on an implicit finite difference discretization of the 2-D Maxwell's equations was developed to solve the scattering problem on a multiprocessor vector supercomputer Cray C98 with 8 CPUs in [52]. Compared to an execution time obtained by sequential solution, a speed-up of 7.8 was obtained for an eight-subdomain solution. The domain decomposition finite difference algorithm could achieve a speed-up close to the physical number of available CPUs for suitable balance factors.

In [53], a parallel 3-D viscoelastic finite difference code was implemented which allowed the work to be distributed across several PCs connected via standard Ethernet or to be run on massively parallel supercomputers, like Crays. Because of poor performance of communication through Ethernet, the speed-up saturated at a factor of 13 when the number of Processing Elements (PEs) becomes larger on a PC cluster. However, the speed-up achieved on the Cray T3E was superlinear: a run with 343 processing elements was 370 times faster than a serial execution.

3.4.2 Parallel FEM Implementations

Finite element analysis (FEA) was first introduced by R. Courant (1943) [54], who utilized the Ritz method of numerical analysis and minimization of variational calculus to obtain approximate solutions to vibration systems. Turneret al in 1956 [55] established a broader definition of numerical analysis. They presented the element stiffness matrix for a triangular element for structural analysis, together with the direct stiffness method for assembling the elements. The term “Finite Element” was first coined by Clough in a paper describing applications in plane elasticity. In the early 1960s, engineers used the method for approximate solutions of problems in stress analysis, fluid flow, heat transfer, and other areas. In the late 1960s and early 1970s, the FEM was applied to a wide variety of engineering problems. Since the rapid decline in the cost of computers and the phenomenal increase in computing power, FEA has produced solutions to an incredible precision. Nowadays, supercomputers are able to produce accurate results for all kinds of problems. The method has been generalized into a branch of applied mathematics for numerical modelling of physical systems in a wide variety of engineering disciplines, e.g., fluid dynamics and electromagnetism.

3.4.2.1 Finite Element Machine in NASA

In late 1970s to the early 1980s, the Finite Element Machine [56], which contained 32 TMS9900 processors and 32 Input/Output boards with a TMS99/4 controller as shown in Figure 8, was completed and successfully tested at the NASA Langley Research Center. The aim of this project was to build and evaluate the performance of a parallel computer for structural analysis. As shown in Figure 9, an array of microprocessors was connected together by 12 local links plus a global bus. An attached unit for floating-point operations was associated with each processor. Due to the local memory for each processor, the processors can be programmed to perform calculations independently. There is a small minicomputer as a front-end controller to code and compile the programs. It is a special purpose computer designed for the efficient solution of finite element code. The speed-up for a plane stress problem was a factor of 4.

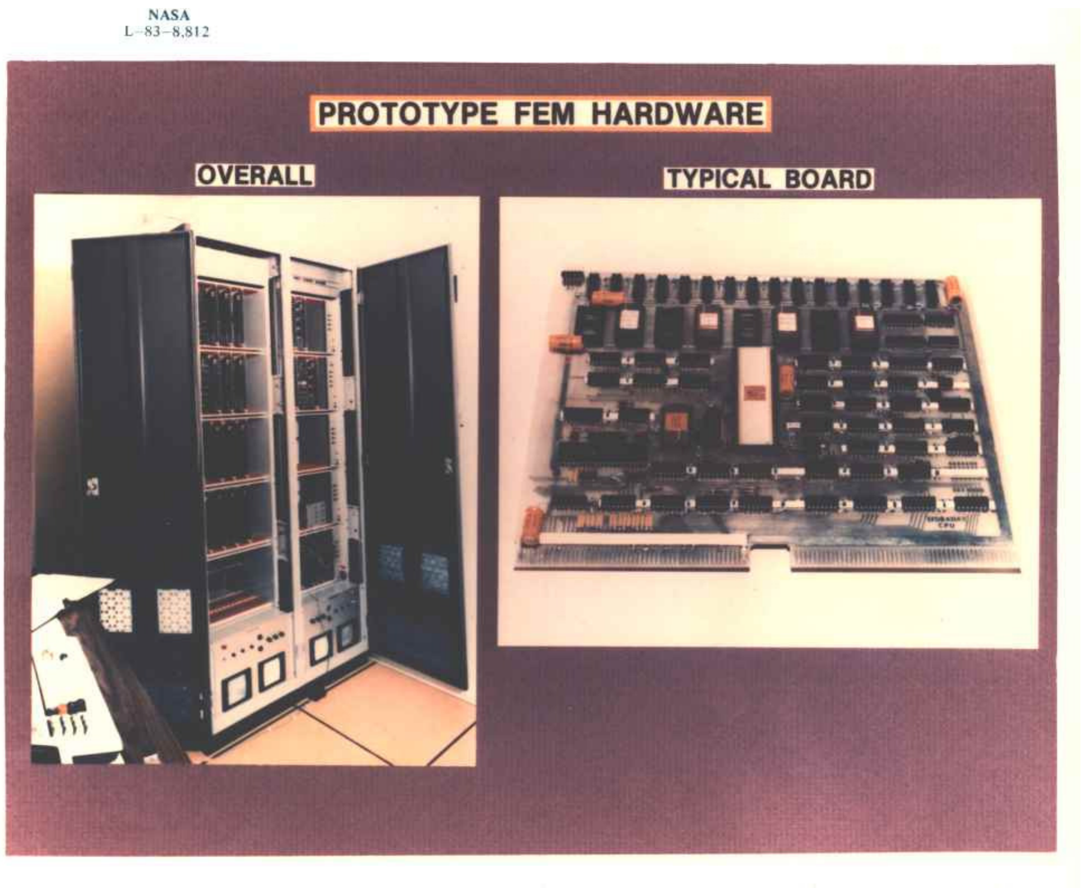


Figure 8 : Prototype NASA Finite Element Machine Hardware [57]

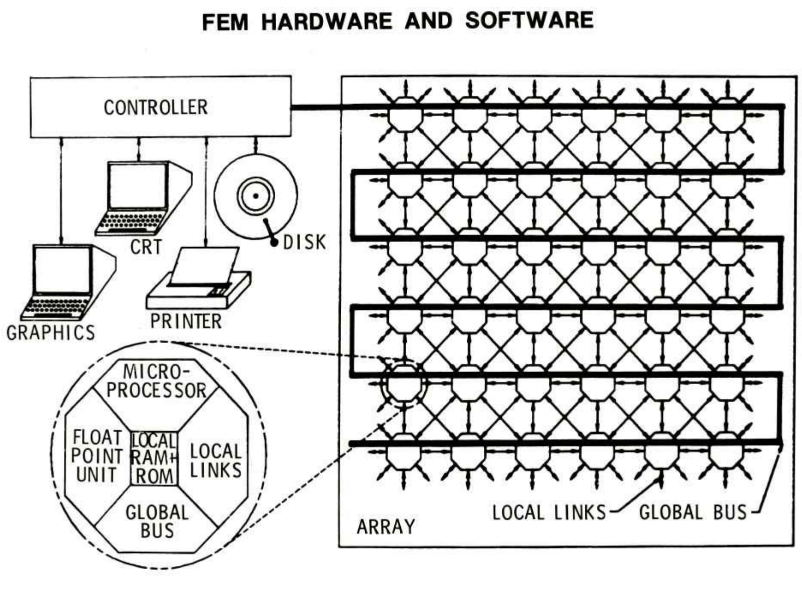


Figure 9 : FEM block diagram [57].

3.4.2.2 The FEM on the Parallel Machine Cenju

Cenju (Cenju-1), which employed the MC68020/WTL1167 as a processing element, was first launched in 1988. An implementation of the FEM on the parallel simulation machine Cenju was introduced in [58] and [59], investigating nonlinear dynamic finite element analysis, which is one of the most time consuming problems in FEM. There are two main stages, the calculation of the stiffness matrix and the solution of a set of linear equations. They present methods of parallelizing the assembly process by using extra buffers for each element of the stiffness matrix in order to reduce the synchronization overhead. A speed up of 48 times was achieved by eliminating the serial bottleneck on a 64 processor system. Furthermore, by parallelizing the solutions of linear equations, a speed-up of 2.9 was attained by parallelizing the Lower triangular matrix and Upper triangular matrix (LU) factorization on a 7 processor system. Also, a speed-up of 36 was gained by parallelizing the conjugate gradient method on a 64 processor system.

3.4.2.3 Interactive and Large Scale Finite Element Analysis in the University of Manchester

The University of Manchester developed techniques to drastically reduce the time taken for FEA [60]. Instead of using the traditional three stage process FEA which involves: (1) pre-processing, (2) equation solution and (3) post-processing, these three tasks are carried out concurrently as a single process by using a number of algorithmic techniques as well as parallel and distributed computing in their new software architecture. Furthermore, the ‘element-by-element’ approach was used, where no overall system equation need be assembled.

Their programs were run on the SGI Origin 3000 machine “Green” with 512 MIPS R12000 processors and 512 GBytes total memory. For the direct numerical solution of the Navier Stokes equations in fluid mechanics, a speed-up of 256 was achieved on 256 processors, whilst sustaining an impressive 30% of the CSAR (Computer Services for Academic Research) machines’ peak performance [61]. An elastoplasticity problem was demonstrated to achieve 116,500 MFLOPs using up to 500 processors. Due to the novel algorithms and software architecture applied, which reduced the solution times by up to five orders of magnitude, finite element problems that are 2-3 orders of magnitude larger compared with a commercial off-the-shelf package could be solved.

3.5 FPGA-based High Performance Computer

Although the growth of the performance of general-purpose computers and parallel supercomputers has been significant in the past decade, the real fraction of peak performance achievable for most numerical computing applications has generally been very poor. Because a large portion of transistors in modern commodity CPUs are utilized to provide flexible data flow, the prevalent computer architecture model is not very well suited for timing-consuming numerical computing applications. The use of supercomputers can get around this problem, but not everyone can afford the high costs of operating and maintaining a supercomputer. Therefore, there is a compelling requirement on finding a more powerful, faster but cheaper computer system for extremely timing-consuming numerical computing applications, especially for 3-D geometrical domains.

Recently, due to the critical requirements for higher speed of program execution, as well as the fact that cluster computing systems based on conventional processors are running out of room to grow, the use of reconfigurable hardware to form customised hardware accelerators for a variety of fields, such as data mining, medical imaging, numerical computations, and financial analytics is growing rapidly. By applying FPGAs coprocessors, the power consumption and the total cost of ownership can be greatly reduced. As a result, FPGAs have been applied in a number of different application areas in the past few years, including [62]:

- Custom hardware.
- Prototyping and testing integrated circuits.
- Reconfigurable hardware accelerator.

- High performance reconfigurable computing.

FPGAs are cheaper than ASICs for small production runs and able to be reprogrammed for functional upgrades in the future. FPGAs are also used to prototype and test integrated circuits before costly dies are made. The first two areas have been the most common use of FPGAs since FPGAs were first launched in 1985, consistent with the original aims of FPGAs hardware and due to their flexibility and short design cycle. Modern FPGA devices allow designers to implement complete systems with minimal requirement for off-chip resources, which has led to research on implementation of FPGA-based hardware/software co-processors within general PCs, which can be used for algorithm acceleration. These will be discussed in detail in section 3.6. In this section, several high performance reconfigurable computers will be introduced.

3.5.1 Finite difference analysis in a configurable computing machine

A 2-D heat transfer simulation system using a Splash-2 configurable computing machine is described in [63]. As shown in Figure 10, Splash-2 consists of 16 splash array boards, an interface board and a SUN SPARC-2 workstation host. Each array board contains 16 processing elements which consist of one Xilinx XC4010 FPGA and one fast static memory.

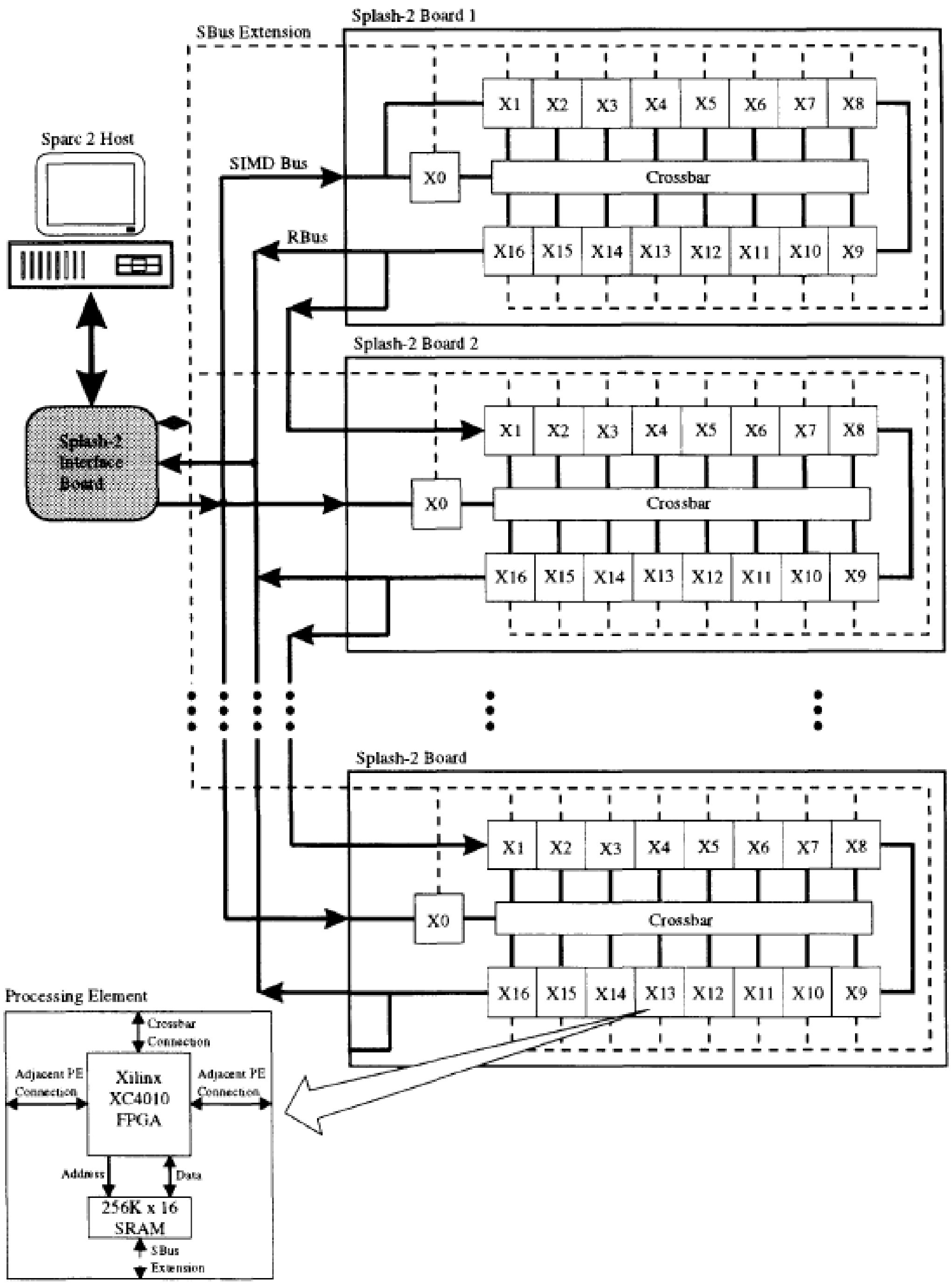


Figure 10 : The Splash-2 system [63].

By discretizing the physical domain of the problem, the computation was partitioned into many individual computations whose independence allowed for the possibility of calculating numerous nodes in parallel. The performance reached up to 3.5 GFLOPs using 16 Splash-2 boards at 20 MHz, with a speed-up of nearly 20 000 compared to the performance of the same simulation on a Sun SPARC-2 workstation. For this particular application, speed up was nearly linearly versus the number of processing elements provided.

3.5.2 Langley's FPGA-based reconfigurable hypercomputer 2001

Several pathfinder scientific applications involving floating-point arithmetic were solved by NASA Langley Reconfigurable Hypercomputers [64]. A development environment and a graphical programming language VIVA was developed and extended by NASA Langley collaborating with Star Bridge.

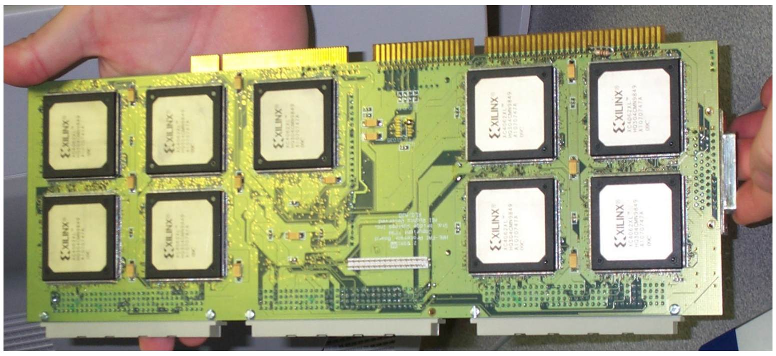


Figure 11 : HAL-15 Reconfigurable Computing Board with 10 Xilinx XC4062 chips (One FPGA located on the reverse side)[64]

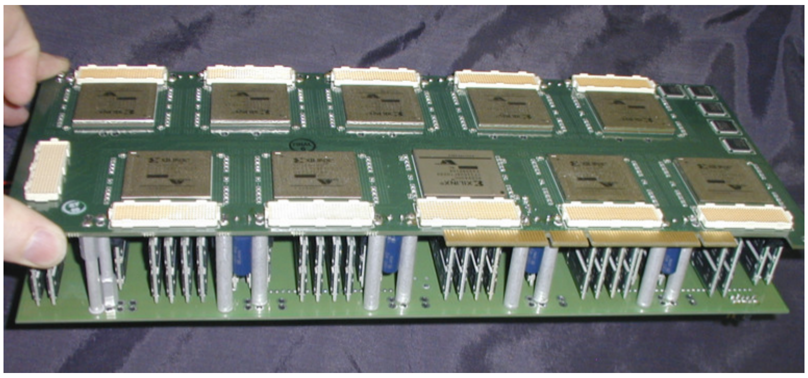


Figure 12 : HAL-15 Reconfigurable Computing Board with 10 Xilinx XC2V6000 FPGA chips[65]

The NASA HAL-15 Hypercomputers contain a circuit board with ten FPGAs with one FPGA located on the reverse side, shown in Figure 11. Each Xilinx XC4062 FPGA contained 62,000 hardware gates, and the performance could reach 0.4 GFLOPS. Thus, the performance of one HAL-15 Hypercomputer is 4 GFLOPS with 10 FPGAs fitted on board. With the rapid growth in FPGA capability, performance has grown from 4 to 470 GFLOPS by using the "newer" XC2V6000 FPGAs, as shown in Figure 12.

3.5.3 High performance linear algebra operations on Cray XD1 (2004)

Several deeply pipelined and highly parallelized linear algebra operations were implemented on a Cray XD1 in [66]. The hardware architecture of the Cray XD1 is shown in Figure 13. The compute blade, which consists of two AMD Opteron processors and one Xilinx Virtex-II Pro FPGA, is the basic architectural unit. Each FPGA can access 16 MB SRAM and 8 GB DRAM through the RapidArray Processors. 6 compute blades fit into one chassis and a typical installation of XD1 contains 12 such chassis, connected by RapidArray external switches.

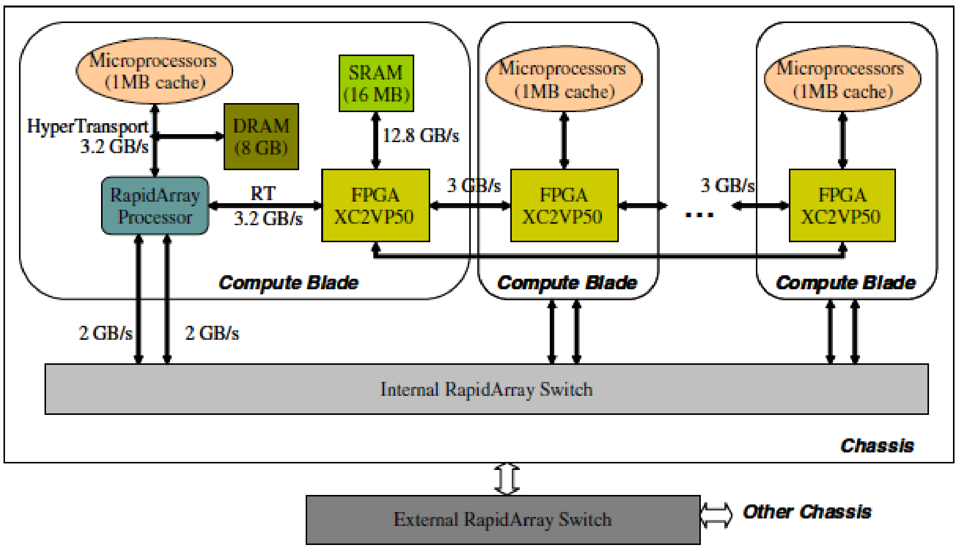


Figure 13 : Hardware Architecture of Cray XD1 [66].

Due to the use of their own floating-point units, the design for 64 bit floating-point matrix multiply achieved a sustained performance of 2.06 GFLOPs on a single FPGA using an SRAM memory bandwidth of 2.1 GB/s and a DRAM memory bandwidth of 24.3 MB/s. Thus, 148.3 GFLOPs can be gained on a 12-chassis installation of the XD1.

3.5.4 FPGA-based supercomputer Maxwell (2007)

The FPGA supercomputer "Maxwell" [67], designed as a general-purpose supercomputer for high-performance reconfigurable computing, was launched in 2007 at the University of Edinburgh in Scotland. Maxwell uses FPGAs as an alternative to conventional microprocessors. The system comprises a 32-way IBM BladeCentre chassis hosting 64 Xilinx Virtex-4 FPGAs, where the FPGA network consists of point-to-point links between the MGT (Multi-Gigabit Transceiver) connectors of adjacent FPGAs, as illustrated in Figure 14. Each blade server holds one Intel Xeon CPU and two Xilinx V4 FPGAs, XC4VLX160 on

Nallatech H101 plug-in PCI cards and XC4VFX100 on Alpha Data ADM-XRC_4FX plug-in PCI cards.

The performance of MCopt (Monte Carlo option pricing), which is a mathematical finance model to calculate the value of an option with multiple sources of uncertainty or with complicated features, gains a speed-up of 300 compared to the results obtained by using the 2.8 GHz Intel Xeon processors in the IBM blades.

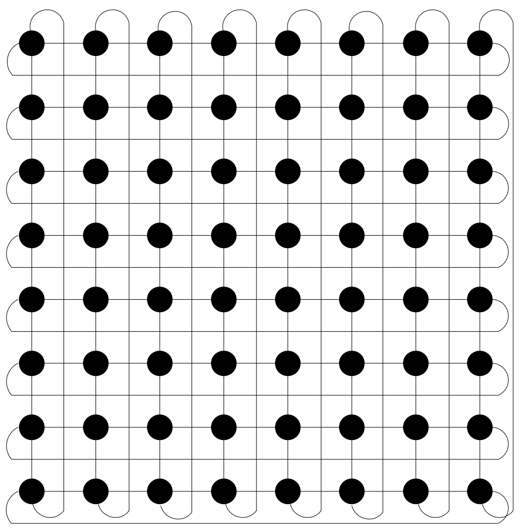


Figure 14 : FPGA connectivity in Maxwell [67].

3.6 FPGA-based Reconfigurable Co-processor Implementation

The rapid improvement in semiconductor technology in the last decade has led to a steady widening of the range of application areas where FPGAs can be applied to achieve massively parallel computation. As a result of the increasing computing power of current FPGAs, reconfigurable hardware has been increasingly used to form custom hardware accelerators within standard computers to achieve numerical computation ([68], [38], [39], [40], [69], [70] and [71]). An FPGA-based stand-alone seismic data processing platform was described in [72], and from the early 1990s, there were several efforts on FPGA-based solutions to accelerate the finite-difference time-domain problem [73], [74]. These applications can exploit the parallelism and pipelineability within the solution algorithms in a much more thorough way than can be done with standard uniprocessor or parallel computers using general-purpose microprocessors. Reconfigurable computing systems bridge the gap between the accessibility of general-purpose microprocessors and the performance of special-purpose supercomputers.

Many researchers have investigated the use of parallel software and hardware approached to FEM acceleration, for example [75], [76], [77], [78], [79], [80] and [81]. In [76], commodity DRAM was used due to its large capacity and bandwidth, achieving a processing rate of 570 MFLOPs. However this approach may not be suitable for many real world applications because of the long latency of DRAM memory. In [75] an implementation based around a sparse matrix-vector multiplier achieved 1.76 GFLOPs. Researchers have also devoted much effort to finding efficient data structures to store the matrix so that the number of memory redirections during sparse matrix-vector multiplication will be minimized, as in [82]. This includes the classic Compressed Storage Row (CSR) format that was introduced to store the non-zero elements of the sparse matrix contiguously in memory, but the additional data structures, which add additional memory access operations, memory band-width pressure and cache interference, will affect the performance and parallelism that can be achieved [83].

These solutions obtained impressive speed-up compared to contemporary PCs. Nevertheless, there are some obvious bottlenecks which affect the performance achieved in their FPGA-based system designs. In these investigations, software algorithms are directly mapped onto the FPGA-based systems without modifications or with only limited modifications such as instruction rescheduling. Normally, the existing numerical algorithms are well suited for commodity CPUs; however they may not be ideal for hardware designs. Therefore, well-modified numerical methods, which can exploit the parallelism and pipelineability within the algorithms, are desirable. Domain decomposition can play an important role, and this will be introduced in later section.

Previously, due to the limitation of logic capacity, hardware designs based on reconfigurable accelerators always used fixed-point arithmetic. However, fixed-point arithmetic has poor numerical properties, i.e. smaller dynamic range and worse accuracy compared to floating-point arithmetic, and would not normally be regarded as acceptable, especially for an unpredictable problem. Recently, with the rapid improvements of speed and programmable logic within FPGAs, its high potential in numerical analysis using floating-point arithmetic has been widely noticed. As shown in [36] and [37], the floating-point arithmetic operations can be competitive compared to floating-point operations on general purpose PCs. In this research, most of the implementations were applied with 32-bit floating-point arithmetic.

3.7 Summary

Numerical solutions to Partial Differential Equations usually require an extremely large amount of computation. The finite difference method is one of the most powerful and widely used approaches to the solution to PDEs, due to its good accuracy and flexibility. Nevertheless, it can be extremely computationally expensive, especially when the number of grid points becomes large. Similarly, the Finite element method usually requires the solution of a large system of linear equations repeatedly. Much effort has therefore been spent on approaches to parallelization of the kernel operations inside the methods. For a successful parallel computing design, the programming models should be independent of the number processors.

An overview of the evolution of parallel computing for numerical solutions as well as a brief description of the current technologies was given in this chapter. A number of parallel implementations of the FDM and FEM were surveyed. They all suffered the same problems: synchronization and communication overheads between processors, as well as poor load balancing between processors, made the speed-up less than linear.

One approach to the parallel computation of solutions is to use reconfigurable hardware to form custom hardware accelerators. FPGA-based reconfigurable computing systems could offer a more efficient way to accelerate the numerical solutions to obtain a reasonable speed-up at an affordable price. In this chapter, several approaches to numerically intensive computation have been described.

In general, 3-dimensional system design for the numerical evaluation of scientific and engineering problems governed by PDEs numerically is computationally demanding and data intensive. Solving such problems may easily take traditional CPUs a couple of days, even months. Good speed-up can be achieved by using the fastest supercomputers. However, the cost in terms of power and energy will be increased dramatically. An alternative is to build specific computer systems using ASICs, but this approach requires a long development period, and is inflexible and costly. GPUs, provide a new approach and are based around a large number of simplified CPU-units. Unfortunately, GPUs are not suitable for problems that are data-intensive due to the long latency of memory. Based on the discussion above, coupling an FPGA-based platform with a general processor becomes the most feasible way to build a high performance "poor man's computer system" with acceptable flexibility at a reasonable cost. Thus, the FPGA-based reconfigurable computing system is used in this research to accelerate computationally intensive and data demanding numerical solutions of 3-dimensional PDE problems.

In the following chapters, the numerical algorithms and the domain decomposition will be formulated and implemented in software to verify the feasibility of this research, and several reconfigurable computing approaches based on the implementation of FPGAs will be presented and evaluated.

Chapter 4

FORMULATION OF THE NUMERICAL SOLUTION APPROACHES

This chapter provides a comprehensive introduction to two numerical approaches to computing the solution of partial differential equations, the finite difference method and the finite element method. The principal stages of the algorithms and the domain decomposition are formulated, and a variety of iteration schemes are evaluated and discussed.

Domain decomposition methods can be applied to numerical problems to break down one large problem into a number of smaller interconnected problems. This chapter will introduce the domain decomposition approaches used for the hardware and software implementation presented in this thesis, namely Fourier decomposition for the FDM and element-by-element analysis for the FEM. The 3-D problems use Fourier decomposition in order to obtain a series of *independent* 2-D sub-problems, so the 3-D problems become suitable for parallel computing. The element-by-element scheme reduces the memory and communication requirements for the solution of large linear algebra equations, because we can calculate the sub-domain matrix-vector multiplications in parallel without assembling the global matrix of FEM. These approaches were chosen in order to maximize parallelism and to minimize the communication required between sub-domains.

4.1 Software Implementation of Finite Difference Method

This section formulates the domain decomposition for splitting the 3-dimensional problem into a series of independent 2-dimensional sub-problems using Fourier decomposition for a cubic 3D domain and the solution of Laplace equation using the finite difference method, and describes the software implementation of a FDM simulator.

4.1.1 Domain discretizations

Firstly, we use the solution of the Laplace equation for a cubic 3D domain as an example. The equation will be solved numerically on a grid consisting of nx, ny and nz grid points in the x, y and z directions respectively. For the Fourier decomposition method to be applicable, it is required that nz be a power of 2 because of the use of the Fast Fourier Transform. The domain therefore has the appearance shown in Figure 15. For simplicity, it is assumed that the domain is x=[0,1], y=[0,1] and z=[0,1].

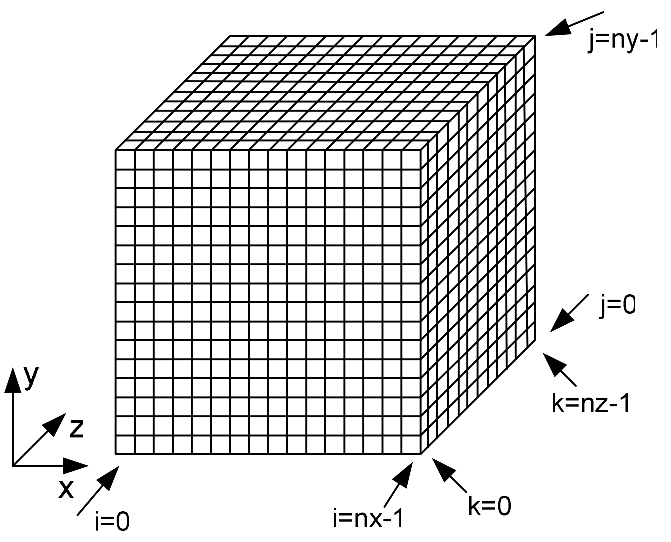


Figure 15 : The 3-dimensional domain to be solved

The governing 3-dimensional Laplace equation is:

$$\nabla^2 u = 0 \quad \text{or} \quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \qquad \textit{Eq. } (30)$$

The Dirichlet boundary conditions are:

$$\begin{array}{lll} \text{i)} & u = f_1(y,z) \ \text{ for } \ x = 0, y = [0,1] \ \text{ and } \ z = [0,1] & \text{(a)} \\ \text{ii)} & u = f_2(y,z) \ \text{ for } \ x = 1, y = [0,1] \ \text{ and } \ z = [0,1] & \text{(b)} \\ \text{iii)} & u = f_3(x,z) \ \text{ for } \ x = [0,1], y = 0 \ \text{ and } \ z = [0,1] & \text{(c)} \\ \text{iv)} & u = f_4(x,z) \ \text{ for } \ x = [0,1], y = 1 \ \text{ and } \ z = [0,1] & \text{(d)} \\ \text{v)} & u = 0 \ \text{ for } \ x = [0,1], y = [0,1] \ \text{ and } \ z = 0 & \text{(e)} \\ \text{vi)} & u = 0 \ \text{ for } \ x = [0,1], y = [0,1] \ \text{ and } \ z = 1 & \text{(f)} \end{array} \qquad \textit{Eq. } (31)$$

Note that the use of a zero boundary conditions for z=0 and z=1 does not entail a loss of generality, since a Laplace equation and, in general, a Poisson equation with a non-homogeneous boundary condition:

vii) $u = f_5(x, y)$ for $x = [0,1], y = [0,1]$ and $z = 0$ (g)

viii) $u = f_6(x, y)$ for $x = [0,1], y = [0,1]$ and $z = 1$ (h)

can be transformed to a Poisson equation satisfying (a-f) using the transformation $u \to \overline{u} + (1-z) f_5(x, y, t) + z f_6(x, y, t)$ during the pre-processing stage, and the inverse transform in the post-processing.

Even though only the Laplace equation is treated in this chapter, the same procedure can be applied to the Poisson equation.

4.1.2 Fourier Decomposition

Here a solution method of the partial differential equation, which uses the Fast Fourier Transform, is introduced. In reality, a 3-D problem of size $N \times N \times N$ is too big to be solved efficiently in hardware. It would need to be broken down into a series of smaller problems using domain decomposition methods. However, standard domain decomposition approaches, based on matrix partition methods, may be inefficient as each sub-domain needs to access the boundaries of other sub-domains, which have not been solved yet. Nevertheless, this problem can be solved by using Fourier decomposition methods. In this case, the 3D problem can be transformed into N 2D problems. These 2D problems represent the Fourier coefficients of the solution in the z direction. The important feature of this is that for a linear problem, these 2D problems are completely decoupled from each other, and can be solved in isolation from one another. As a result, the 2D problems (which are small enough to be efficiently solved in hardware) can be solved in parallel to give very fast solution. Alternatively, if the problem is linear in the y direction, we can use the Fourier decomposition method again to further decompose each 2D problem down into N 1D problems, which are then solved in parallel. The parallelism is suitable to be implemented in hardware. We use the Fast Fourier Transform (discrete) [84] to obtain the Dirchlet boundary conditions and Neumann boundary conditions of the PDE.

Consider Eq. (30) and let

$$u(x,y,z)=\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi z) \qquad Eq.\ (32)$$

The function $u(x,y,z)$ used in the Fourier decomposition is periodic in the z direction, but this is not a significant restriction. For a real system, the domain is finite; whether the function is periodic or not outside the domain in z direction is immaterial, because what happens outside the domain is not a problem to the method of analysis. The required behaviour within the domain can be obtained by an appropriate selection of the Dirchlet or Neumann boundary conditions.

Checking the boundary conditions:

i) at y=1, z=(0,1) and x=(0,1) we have

$u(x,1,z)=\sum_{m=0}^{\infty}u_m(x,1)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,1)\sin(2m\pi z)$. By choosing the value of the functions $u_m(x,1)$ and $v_m(x,1)$ so that

$$u(x,1,z)=f(x,z)=\sum_{m=0}^{\infty}u_m(x,1)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,1)\sin(2m\pi z).$$

The boundary value of the functions can be determined by Fourier transform for the continuous problem. For the discrete problem, the Fast (discrete) Fourier transform can be used instead.

ii) at y=0, we have

$u(x,0,z)=\sum_{m=0}^{\infty}u_m(x,0)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,0)\sin(2m\pi z)=0$. This would imply $u_m(x,0)=v_m(x,0)=0$ using the orthogonality property of $\cos(2m\pi z)$ and $\sin(2m\pi z)$.

iii) at z=0, we have

$$u(x,y,0)=\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi 0)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi 0)=0$$

giving $\sum_{m=0}^{\infty}u_m(x,y)=0$

iv) at z=1, we have

$$u(x,y,1)=\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi\cdot 1)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi\cdot 1)=0$$

also gives $\sum_{m=0}^{\infty}u_m(x,y)=0$

v) at x=0, we have

$u(0,y,z)=\sum_{m=0}^{\infty}u_m(0,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(0,y)\sin(2m\pi z)=0$. This would imply

$u_m(0,y)=v_m(0,y)=0$ again using the orthogonal property of $\cos(2m\pi z)$ and $\sin(2m\pi z)$.

vi) at x=1, we have

$u(1,y,z)=\sum_{m=0}^{\infty}u_m(1,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(1,y)\sin(2m\pi z)=0$. This would imply $u_m(1,y)=v_m(1,y)=0$ using the orthogonality property of $\cos(2m\pi z)$ and $\sin(2m\pi z)$.

Using the Fourier transform, we are going to transform the partial differential equation (PDE) in the form of *Eq.*(30) into PDEs with respect to x and y only by putting *Eq.* (32) into *Eq.*(30), we have

$$\begin{aligned}&\frac{\partial^2\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi z)}{\partial x^2}\\&+\frac{\partial^2\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi z)}{\partial y^2}\\&+\frac{\partial^2\sum_{m=0}^{\infty}u_m(x,y)\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,y)\sin(2m\pi z)}{\partial z^2}=0\end{aligned} \qquad Eq.\ (33)$$

$$\begin{aligned}&\sum_{m=1}^{\infty}u_m(x,y)4m^2\pi^2\cos(2m\pi z)+\sum_{m=1}^{\infty}v_m(x,y)4m^2\pi^2\sin(2m\pi z)\\&=\sum_{m=0}^{\infty}\frac{\partial^2u_m(x,y)}{\partial y^2}\cos(2m\pi z)+\sum_{m=1}^{\infty}\frac{\partial^2v_m(x,y)}{\partial y^2}\sin(2m\pi z)\\&+\sum_{m=0}^{\infty}\frac{\partial^2u_m(x,y)}{\partial x^2}\cos(2m\pi z)+\sum_{m=1}^{\infty}\frac{\partial^2v_m(x,y)}{\partial x^2}\sin(2m\pi z)\end{aligned} \qquad Eq.\ (34)$$

Using the orthogonality properties of the cosine and sine function, which are

$$\int_0^1 \cos(2m\pi z)\cos(2k\pi z)\,dz$$
$$= \tfrac{1}{2}\int_0^1 \left[\cos\left(2(m-k)\pi z\right) + \cos\left(2(m+k)\pi z\right)\right] dz \quad = \begin{cases} 1 \; for \; k = m = 0 \\ \tfrac{1}{2}\delta_{km} \; otherwise \end{cases}$$ *Eq.* (35)

$$\int_0^1 \sin(2m\pi z)\sin(2k\pi z)\,dz$$
$$= \tfrac{1}{2}\int_0^1 \left[\cos\left(2(m-k)\pi z\right) - \cos\left(2(m+k)\pi z\right)\right] dz \quad = \begin{cases} 0 \; for \; k = m = 0 \\ \tfrac{1}{2}\delta_{km} \; otherwise \end{cases}$$ *Eq.* (36)

$$\int_0^1 \sin(2m\pi z)\cos(2k\pi z)\,dz$$
$$= \tfrac{1}{2}\int_0^1 \left[\sin\left(2(m-k)\pi z\right) + \sin\left(2(m+k)\pi z\right)\right] dz \quad = 0$$ *Eq.* (37)

Eq. (34) becomes

$$\frac{\partial^2 u_m(x,y)}{\partial x^2} + \frac{\partial^2 u_m(x,y)}{\partial y^2} - 4m^2\pi^2 u_m(x,y) = 0$$ *Eq.* (38)

and

$$\frac{\partial^2 v_m(x,y)}{\partial x^2} + \frac{\partial^2 v_m(x,y)}{\partial y^2} - 4m^2\pi^2 v_m(x,y) = 0$$ *Eq.* (39)

subject to the boundary conditions

i) at x=0 $u_m(0,y) = v_m(0,y) = 0$

ii) at x=1 $u_m = u_m(1,y)$ and $v_m = v_m(1,y)$ *Eq.* (40)

iii) at y=0 $u_m(x,0) = v_m(x,0) = 0$

iv) at y=1 $u_m(x,1) = v_m(x,1) = 0$

Thus the original 3D problem has been composed into a series of *nz* 2D slices (corresponding to different values of m in Eq. (38) and Eq. (39)), which are decoupled and can be solved separately without any requirement for exchange of data or synchronization between the slices.

The same procedure can also be applied to a time dependent problem, such as the parabolic diffusion equation, which represents the time dependent consolidation problem, time dependent electrical field problem, and heat diffusion problem, which has the form:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = C\frac{\partial u}{\partial t} \qquad \textit{Eq. (41)}$$

The method is even applicable to some nonlinear problems, such as the non-linear form:

$$\frac{\partial}{\partial x}\left[k_x(x,y)\frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[k_y(x,y)\frac{\partial u}{\partial y}\right] + k_z(x,y)\frac{\partial^2 u}{\partial z^2} = 0 \qquad \textit{Eq. (42)}$$

$$\frac{\partial}{\partial x}\left[k_x(x,y)\frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[k_y(x,y)\frac{\partial u}{\partial y}\right] + k_z(x,y)\frac{\partial^2 u}{\partial z^2} = g(x,y,z) \qquad \textit{Eq. (43)}$$

and

$$\frac{\partial}{\partial x}\left[k_x(x,y,t)\frac{\partial u}{\partial x}\right] + \frac{\partial}{\partial y}\left[k_y(x,y,t)\frac{\partial u}{\partial y}\right] + k_z(x,y,t)\frac{\partial^2 u}{\partial z^2} = C(x,y,t)\frac{\partial u}{\partial t} \qquad \textit{Eq. (44)}$$

The method is applicable as long as the domain is uniform in the direction of the Fourier Transform i.e. the *z*-direction in the current configuration and the material non-linearity is not included in that direction. The principle of superposition must apply therefore only linear system in *z* direction will be allowed for Fourier decomposition to work. In civil engineering, Fourier decomposition is widely applicable for “long” structures, e.g. roads and dams where the cross section is constant for a long distance. Fourier decomposition is also widely used in the finite strip method and the finite layer method.

4.1.3 The Finite Difference Method

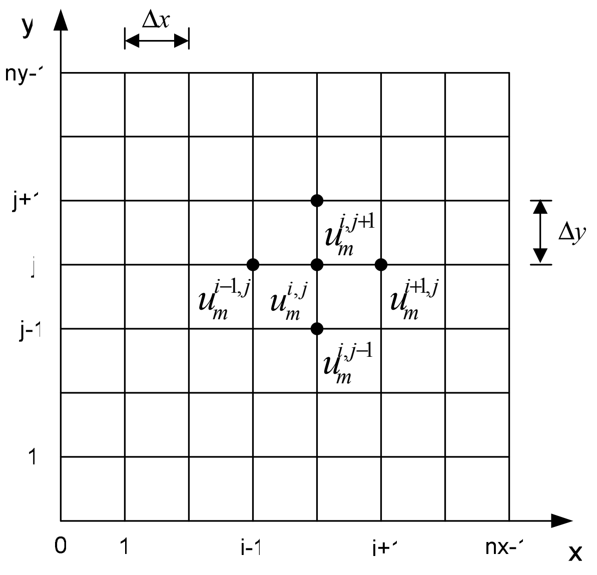


Figure 16 : 2D finite difference grid

The 2D solution domain is shown in Figure 16, covered by a two-dimensional grid of lines. The intersections of these grid lines are the grid points at which the finite difference solution to the PDE after Fourier decomposition is to be obtained. As shown in Figure 16, these equally spaced grid lines are perpendicular to the x and y axes and the uniform distances are Δx and Δy, respectively. The subscript i is used to denote the physical grid lines corresponding to constant values of x, where $x_i = i \cdot \Delta x$, and the subscript j is used to denote the physical grid lines corresponding to constant values of y, where $y_j = j \cdot \Delta y$. Therefore, grid point (i, j) is corresponding to location (x_i, y_j) in the solution domain. Also, the subscript notation is used to denote the dependent variable at the grid point which uses the same subscript, i.e. $u_m(x_i, y_j) = u_m^{i,j}$.

If Eq.(38) is to be solved using finite difference method then, it can be written as

$$\frac{\partial^2 u_m(x,y)}{\partial x^2} + \frac{\partial^2 u_m(x,y)}{\partial y^2} - \lambda^2 u_m(x,y) = 0 \qquad Eq.\ (45)$$

where $\lambda^2 = 4m^2\pi^2$, and Eq. (45) can be approximated as

$$\frac{u_m^{i+1,j} + u_m^{i-1,j} - 2u_m^{i,j}}{\Delta x^2} + \frac{u_m^{i,j+1} + u_m^{i,j-1} - 2u_m^{i,j}}{\Delta y^2} - \lambda^2 u_m^{i,j} = 0 \qquad Eq.\ (46)$$

Choosing $\Delta x = \Delta y$, we have

$$u_m^{i,j} = \frac{u_m^{i-1,j} + u_m^{i+1,j} + u_m^{i,j-1} + u_m^{i,j+1}}{4 + \lambda^2 \Delta x^2} \qquad Eq.\ (47)$$

Eq. (47) must be modified at the boundaries. For example, at $u^{0,1}$, Eq. (47) gives a reference to a point outside the domain at location $(-1,0)$. If $u^{0,1}$ lies on a Dirichlet boundary, no update on the value of u is performed, so Eq. (47) is not evaluated, and there is no problem. If $u^{0,1}$ lies on a Neumann boundary condition, the reflecting boundary condition is enforced by using a "virtual point" at $(-1,1)$, whose behaviour exactly mirrors the behaviour of the corresponding point at $(+1,1)$. Thus the value $u^{-1,1}$ is substituted by the value of $u^{1,1}$, and then the equation solved as normal. A similar procedure is applied whenever $i=0, i=nx-1, j=0$ or $j=ny-1$.

And the same procedure applied on Eq. (39), then we have

$$v_m^{i,j} = \frac{v_m^{i-1,j} + v_m^{i+1,j} + v_m^{i,j-1} + v_m^{i,j+1}}{4+\lambda^2 \Delta x^2} \qquad Eq.\ (48)$$

The value of $u_m(x,1)$ and $v_m(x,1)$ can be found from

$$f(x,z) = \sum_{m=0}^{\infty} u_m(x,1)\cos(2m\pi z) + \sum_{m=1}^{\infty} v_m(x,1)\sin(2m\pi z) \qquad Eq.\ (49)$$

4.1.4 Fourier Decomposition in the y direction and Exact Solution

We can use the Fourier transform again for the y direction to transform the partial differential equation (PDE) in the form of Eq. (38) and (39) into an ordinary differential equation (ODE) by assuming

$$u_m(x,y) = \sum_{n=0}^{\infty} u_{mn}(x)\cos(2n\pi y) + \sum_{n=1}^{\infty} a_{mn}(x)\sin(2n\pi y) \qquad Eq.\ (50)$$

And

$$v_m(x,y) = \sum_{n=0}^{\infty} v_{mn}(x)\cos(2n\pi y) + \sum_{n=1}^{\infty} b_{mn}(x)\sin(2n\pi y) \qquad Eq.\ (51)$$

Putting Eq. (50) into Eq. (38), we have (only u_m is shown as v_m is the same but with different coefficients):

$$\frac{\partial^2\left(\sum_{n=0}^{\infty}u_{mn}(x)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(x)\sin(2n\pi y)\right)}{\partial x^2} + \frac{\partial^2\left(\sum_{n=0}^{\infty}u_{mn}(x)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(x)\sin(2n\pi y)\right)}{\partial y^2} - 4m^2\pi^2\left(\sum_{n=0}^{\infty}u_{mn}(x)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(x)\sin(2n\pi y)\right)=0 \qquad \text{Eq. (52)}$$

$$\sum_{n=0}^{\infty}\frac{d^2u_{mn}(x)}{dx^2}\cos(2n\pi y)+\sum_{n=1}^{\infty}\frac{d^2a_{mn}(x)}{dy^2}\sin(2n\pi y) = 4n^2\pi^2\left(\sum_{n=0}^{\infty}u_{mn}(x)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(x)\sin(2n\pi y)\right) + 4m^2\pi^2\left(\sum_{n=0}^{\infty}u_{mn}(x)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(x)\sin(2n\pi y)\right) \qquad \text{Eq. (53)}$$

Using the orthogonal properties of the cosine and sine function, Eq. (53) becomes

$$\frac{d^2u_{mn}(x)}{dx^2}-4\left(m^2+n^2\right)\pi^2u_{mn}(x)=0 \qquad \text{Eq. (54)}$$

The exact solution of Eq. (54) is

$$u_{mn}(x)=C_1\sinh(2\sqrt{m^2+n^2}\pi x)+C_2\cosh(2\sqrt{m^2+n^2}\pi x) \qquad \text{Eq. (55)}$$

subject to the boundary conditions

v) at x=0 $u_m = v_m = 0$

vi) at x=1 $u_m(1,y)=\sum_{n=0}^{\infty}u_{mn}(1)\cos(2n\pi y)+\sum_{n=1}^{\infty}a_{mn}(1)\sin(2n\pi y)$

Eq. (56)

and the coefficients can be found using the orthogonality property of the cosine and sine terms.

$$u_{mn}(0) = C_1 \sinh(0) + C_2 \cosh(0) = 0 \qquad \text{Eq. (57)}$$

$$C_2 = 0 \qquad \text{Eq. (58)}$$

Putting Eq. (58) into Eq. (55), we have

$$u_{mn}(x) = C_1 \sinh(2\sqrt{m^2+n^2}\pi x) \qquad \text{Eq. (59)}$$

Therefore

$$u_{mn}(x) = u_{mn}(1)\frac{\sinh 2\sqrt{m^2+n^2}\pi x}{\sinh 2\sqrt{m^2+n^2}\pi} \qquad \text{Eq. (60)}$$

If the exact solution of Eq. (54) is not available, using the finite difference method, Eq. (54) can be written as

$$\frac{d^2u_{mn}(x)}{dx^2} - \lambda^2 u_{mn}(x) = 0 \qquad \text{Eq. (61)}$$

where $\lambda^2 = 4\pi^2(m^2+n^2)$. Eq. (61) can be approximated as

$$\frac{u_{mn}^{1} + u_{mn}^{-1} - 2u_{mn}^{0}}{\Delta x^2} - \lambda^2 u_{mn}^{0} = 0 \qquad \text{Eq. (62)}$$

or

$$u_{mn}^{0} = \frac{u_{mn}^{1} + u_{mn}^{-1}}{2 + \lambda^2 \Delta x^2} \qquad \text{Eq. (63)}$$

For finite difference $\frac{u_{mn}^{1}+u_{mn}^{-1}}{u_{mn}^{0}}=2+\lambda^2\Delta x^2$

For exact solution $\frac{u_{mn}^{1}+u_{mn}^{-1}}{u_{mn}^{0}}=\frac{A\sinh(\lambda x+\lambda\Delta x)+A\sinh(\lambda x-\lambda\Delta x)}{A\sinh(\lambda x)}=\frac{2\sinh(\lambda x)\cosh(\lambda\Delta x)}{\sinh(\lambda x)}$

$=2\cosh(\lambda\Delta x)\ =2+\lambda^2\Delta x^2+\frac{\lambda^4\Delta x^4}{12}+\ldots$ Eq. (64)

4.1.5 Iteration Schemes

Based on the discussions in section 2.5, relaxation methods are considered here, because the matrices arising from the system of finite difference equations are generally very large and sparse, especially for 3-dimensional system domain.

The finite difference method based on Jacobi iteration is:

$$u_{i,j}^{k+1}=\frac{u_{i,j+1}^{k}+u_{i,j-1}^{k}+u_{i-1,j}^{k}+u_{i+1,j}^{k}}{4+\lambda^2\Delta x^2}\qquad\text{Eq. (65)}$$

where the superscript $k(k=0,1,2,\ldots)$ denotes the iteration number, λ is a coefficient from Fourier decomposition with $\lambda^2=4m^2\pi^2$. An initial approximation $(k=0)$ must be made for $u_{i,j}$ to start the finite difference process.

The Gauss Seidel method, which is applied to the finite difference solution to the Laplace equation, is as follows:

$$u_{i,j}^{GS(k+1)} = \frac{u_{i,j+1}^{k} + u_{i,j-1}^{k+1} + u_{i-1,j}^{k+1} + u_{i+1,j}^{k}}{4 + \lambda^2 \Delta y^2} \qquad \text{Eq. (66)}$$

The successive over-relaxation (SOR) method becomes like:

$$u_{i,j}^{k+1} = u_{i,j}^{k} + \omega(u_{i,j}^{GS(k+1)} - u_{i,j}^{k}) \qquad \text{Eq. (67)}$$

where ω is a relaxation factor, which lies between 0.0 and 2.0.

4.1.6 Software Implementation

This sub-section describes and analyses the software simulation of the FDM for 1 dimension, 2 dimensions, and 3 dimensions. A software simulator for the FDM was written in C Language in order to understand the processes and operations of the FDM better, to allow the functional verification and validation of the hardware implementation, and the most important is to provide a baseline against which speed is measured. Fast Fourier Transform (FFT), written in C based on modified Numerical Recipes Software [100], is used for Fourier decomposition process. The software also acts as an interface to the reconfigurable computing platform.

The procedure used by the simulation is illustrated in Figure 17. It first reads what kind of FDM needs to be simulated and which boundary conditions it has. It then decides which method will be used, and then compares the results in different methods to get ready for the hardware simulation.

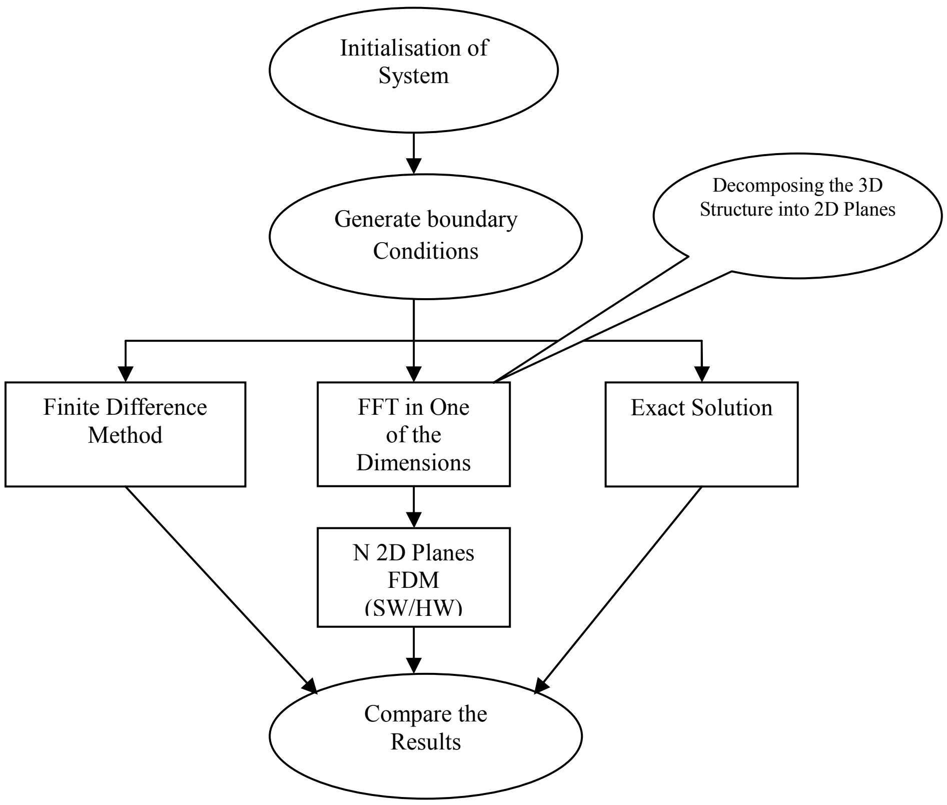


Figure 17 : 3D FDM simulation flow graph

Solution of the Partial Differential Equation using the FFT, requires that the domain to be uniform in one of the directions and in this direction no variation of material property is allowed. Usually, we assume that this direction is the z -direction.

4.2 The Finite Element Method

The mathematical basis of the finite element method is introduced in this section, from the 1-D linear solid element to 3-D tetrahedral element. The basic flow chart of FEM is shown in Figure 5. The first step of FEM is to break up a continuous physical domain into smaller sub-domains, for example, triangular for 2-D domains, while tetrahedral for 3-D domains. Then, the mesh generation is an important step which will determine the approximation accuracy, however, the common mesh generation methods integrate complex data structures which can not be implemented effectively on hardware devices. So, the main effort is spent on accelerating the solution of large linear system equations.

1-D linear solid element and 2-D rectangular plane strain element are introduced to explain basic concepts about finite element analysis. Nowadays, 3-dimensional finite element modelling is widely applied, which gives a more realistic solution. However, the 3-D finite element method requires a large amount of computation power and memory in order to solve the large but sparse matrix-vector system of equations while losing the ability to run on all but the fastest computers effectively. Therefore, an iterative element-by-element scheme is used for the solution of 3-D finite element analysis.

4.2.1 1D Linear Solid Element

A 1D linear solid system, with *n* elements connected through nodes 0-*n*, is shown in Figure 18. This is a dynamic solid element in the form of a bar with Young's modulus E, material density ρ, cross-sectional area A and element length of l. The node number 0 is free, element 1 has nodes 1 and 2 while element 2 has nodes 2 and 3, and so on. The node number n is fixed.

The Governing differential equation for dynamic behaviour is:

$$E\frac{\partial^2 u}{\partial x^2} - \rho\frac{\partial^2 u}{\partial t^2} = 0 \qquad \text{Eq. (68)}$$

which can be written as

$$c^2\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial t^2} = 0 \qquad \text{Eq. (69)}$$

where $c = \sqrt{\frac{E}{\rho}}$ is called the wave velocity.

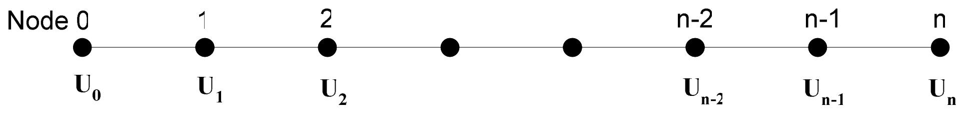


Figure 18 : 1D finite element mesh

$$u(x) = N_1(x)u_1 + N_2(x)u_2 \qquad \text{Eq. (70)}$$

where N_1 and N_2 are called the shape functions.

A simple set of shape functions for the 1D linear element is:

$$N_1(x) = 1 - \frac{x}{l} \text{ and } N_2(x) = \frac{x}{l} \qquad \text{Eq. (71)}$$

when $x = 0$, $N_1 = 1$ and $N_2 = 0$, as for $x = l$, $N_1 = 0$ and $N_2 = 1$.

The total mass of the element u_i is $M_e = \rho A l$ and this is divided into two equal parts and, using the lumped mass scheme, assigned to each side node, then we have

$$M_e = \frac{\rho A l}{2}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \text{Eq. (72)}$$

Direct mass lumping drives a diagonal mass matrix, which can be stored simply as a vector and offers computational advantages under certain simulations, such as the inverse of a diagonal matrix is also diagonal.

Since there two nodes for a single element with one degree of freedom at each node, the lumped mass matrix and stiffness matrix are as follows:

$$\underline{M}_e \underline{\ddot{u}}_e = \frac{\rho A l}{2}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} \ddot{u}_1 \\ \ddot{u}_2 \end{pmatrix} \qquad \text{Eq. (73)}$$

$$\underline{K}_e \underline{u}_e = \frac{EA}{l}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \qquad \text{Eq. (74)}$$

For the element eigenvalue problem

$$\underline{K}_e \underline{u}_e = \omega^2 \underline{M}_e \underline{u}_e \qquad \text{Eq. (75)}$$

The eigenvectors and eigenvalues are

$$\underline{u}_e^1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad \omega_1^e = 0 \qquad \text{Eq. (76)}$$

and

$$\underline{u}_e^2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \omega_2^e = \frac{2}{l}\sqrt{\frac{E}{\rho}} \qquad \text{Eq. (77)}$$

The critical time step derives from the equation with the largest eigenvalue of the element eigenvalue problem Eq. (75), where $\omega_{\max}^e = \max(\omega_1^e, \omega_2^e) = \frac{2}{l}\sqrt{\frac{E}{\rho}}$. Therefore the critical time step is [85] :

$$\Delta t_{crit} = \frac{2}{\omega_{\max}} = l\sqrt{\frac{\rho}{E}} \qquad \text{Eq. (78)}$$

where the time step must be smaller than the critical time step to obtain stability for the explicit scheme used.

From Eq.(73) and Eq.(74) , the assembled global mass matrix and stiffness matrix are in the form of:

$$\underline{M}\underline{\ddot{u}} = \frac{\rho A l}{2}\begin{bmatrix} 1 & 0 & 0 & 0 & .... \\ 0 & 1+1 & 0 & 0 & .... \\ 0 & 0 & 1+1 & 0 & .... \\ 0 & 0 & 0 & 1+1 & .... \\ .... & .... & .... & .... & .... \end{bmatrix}\begin{pmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \\ \ddot{u}_4 \\ .... \end{pmatrix} = \frac{\rho A l}{2}\begin{bmatrix} 1 & 0 & 0 & 0 & .... \\ 0 & 2 & 0 & 0 & .... \\ 0 & 0 & 2 & 0 & .... \\ 0 & 0 & 0 & 2 & .... \\ .... & .... & .... & .... & .... \end{bmatrix}\begin{pmatrix} \ddot{u}_1 \\ \ddot{u}_2 \\ \ddot{u}_3 \\ \ddot{u}_4 \\ .... \end{pmatrix} \qquad \text{Eq. (79)}$$

and

$$\underline{K}\underline{u} = \frac{EA}{l}\begin{bmatrix} 1 & -1 & 0 & 0 & .... \\ -1 & 1+1 & -1 & 0 & .... \\ 0 & -1 & 1+1 & -1 & .... \\ 0 & 0 & -1 & 1+1 & .... \\ .... & .... & .... & .... & .... \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ .... \end{pmatrix} = \frac{EA}{l}\begin{bmatrix} 1 & -1 & 0 & 0 & .... \\ -1 & 2 & -1 & 0 & .... \\ 0 & -1 & 2 & -1 & .... \\ 0 & 0 & -1 & 2 & .... \\ .... & .... & .... & .... & .... \end{bmatrix}\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ .... \end{pmatrix} \qquad \text{Eq. (80)}$$

The global dynamic equation is:

$$\underline{M}\ddot{\underline{u}} + \underline{K}\underline{u} = 0 \qquad \text{Eq. (81)}$$

Using central difference scheme, the global dynamic equation becomes

$$\underline{M}\frac{\underline{u}^{t+\Delta t} + \underline{u}^{t-\Delta t} - 2\underline{u}^{t}}{\Delta t^2} + \underline{K}\underline{u}^{t} = 0 \qquad \text{Eq. (82)}$$

Then solve out $\underline{u}^{t+\Delta t}$:

$$\underline{u}^{t+\Delta t} = -\underline{u}^{t-\Delta t} + 2\underline{u}^{t} - \frac{\Delta t^2 \underline{K}\underline{u}^{t}}{\underline{M}} \qquad \text{Eq. (83)}$$

or applying Neumman boundary condition at x=0 and Direchlet boundary condition at x=n*l*

$$\begin{aligned} u_i^{t+\Delta t} &= -u_i^{t-\Delta t} + 2u_i^t + const0_i \times (2u_i^t - u_{i+1}^t - u_{i+1}^t) && \text{(i=0)} \\ u_i^{t+\Delta t} &= -u_i^{t-\Delta t} + 2u_i^t + const0_i \times (2u_i^t - u_{i-1}^t - 0) && \text{(i=n)} \qquad \text{Eq. (84)} \\ u_i^{t+\Delta t} &= -u_i^{t-\Delta t} + 2u_i^t + const0_i \times (2u_i^t - u_{i-1}^t - u_{i+1}^t) && \text{(i=1 to n-1)} \end{aligned}$$

where $const0_i = -\frac{2E\Delta t^2}{\rho l^2 \times Mi,i}$.

4.2.2 2D Rectangular Plane Strain Element

The rectangular element, one of the simplest 2D elements for 2D finite element analysis, has 4 nodes at each corner. Consider a rectangular element, Figure 19, with nodes 1, 2, 3 and 4 with local coordinates (x_i,y_i) which are (-a,-b), (a,-b), (a,b) and (-a,b) respectively. The displacement at the nodes are (u_1,v_1), (u_2,v_2), (u_3,v_3) and (u_4,v_4) respectively.

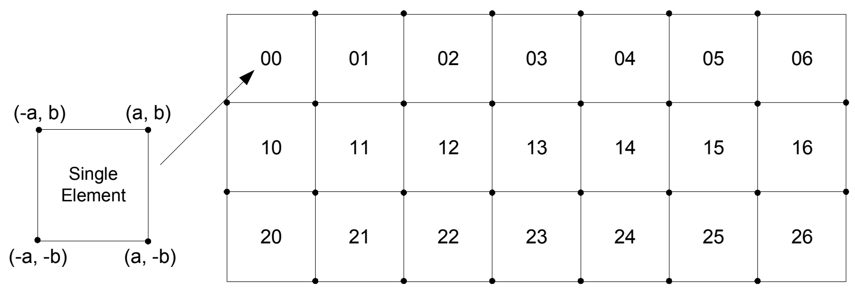


Figure 19 : 2-dimensional rectangular plane strain elements

The next step is the key finite element approximation – assuming that the displacement at any point within the element depends only on the displacement of the nodes therefore

$$u(x,y)=\sum_{i=1}^{4}N_i(x,y)u_i \qquad \text{Eq. (85)}$$

$$v(x,y)=\sum_{i=1}^{4}N_i(x,y)v_i \qquad \text{Eq. (86)}$$

where N_1, N_2, N_3, N_4 are called the shape functions, a simple set of shape functions for the 4-noded element is:

$$N_1(x,y)=\frac{1}{4}\left(1-\frac{x}{a}\right)\left(1-\frac{y}{b}\right) \qquad \text{Eq. (87)}$$

$$N_2(x,y)=\frac{1}{4}\left(1+\frac{x}{a}\right)\left(1-\frac{y}{b}\right) \qquad \text{Eq. (88)}$$

$$N_3(x,y)=\frac{1}{4}\left(1+\frac{x}{a}\right)\left(1+\frac{y}{b}\right) \qquad \text{Eq. (89)}$$

$$N_4(x,y)=\frac{1}{4}\left(1-\frac{x}{a}\right)\left(1+\frac{y}{b}\right) \qquad \text{Eq. (90)}$$

Or

$$N_i(x,y)=\frac{1}{4}\left(1+\frac{x}{x_i}\right)\left(1+\frac{y}{y_i}\right) \qquad \text{Eq. (91)}$$

The mass matrix is given by

$$\underline{M}_e\underline{\ddot{u}}_e=\rho t\begin{bmatrix}\int_{-b}^{b}\int_{-a}^{a}N_iN_jdxdy & 0\\ 0 & \int_{-b}^{b}\int_{-a}^{a}N_iN_jdxdy\end{bmatrix}\begin{pmatrix}\ddot{u}_j\\ \ddot{v}_j\end{pmatrix} \qquad \text{Eq. (92)}$$

Using nodal integration in order to obtain a diagonal matrix, we have

$$\underline{M}_e\underline{\ddot{u}}_e=\rho tab\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{pmatrix}\ddot{u}_j\\ \ddot{v}_j\end{pmatrix} \qquad \text{Eq. (93)}$$

Or

$$\underline{M}_e\underline{\ddot{u}}_e=\rho tab\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix}\begin{pmatrix}\ddot{u}_1\\ \ddot{v}_1\\ \ddot{u}_2\\ \ddot{v}_2\\ \ddot{u}_3\\ \ddot{v}_3\\ \ddot{u}_4\\ \ddot{v}_4\end{pmatrix} \qquad \text{Eq. (94)}$$

The stiffness matrix is in the form of

$$\underline{K}_e\underline{u}_e = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}$$

$$\int_{-b}^{b}\int_{-a}^{a}\begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & \frac{\partial N_i}{\partial y} \\ 0 & \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} \end{bmatrix}\begin{bmatrix} 1 & \frac{\nu}{1-\nu} & 0 \\ \frac{\nu}{1-\nu} & 1 & 0 \\ 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix}\begin{bmatrix} \frac{\partial N_j}{\partial x} & 0 \\ 0 & \frac{\partial N_j}{\partial y} \\ \frac{\partial N_j}{\partial y} & \frac{\partial N_j}{\partial x} \end{bmatrix} dxdy \begin{pmatrix} u_j \\ v_j \end{pmatrix} \qquad \text{Eq. (95)}$$

The global dynamic equation is:

$$\underline{M}\,\underline{\ddot{u}} + \underline{K}\,\underline{u} = 0 \qquad \text{Eq. (96)}$$

using central difference scheme, the global dynamic equation becomes

$$\underline{M}\frac{\underline{u}^{t+\Delta t} + \underline{u}^{t-\Delta t} - 2\underline{u}^{t}}{\Delta t^2} + \underline{K}\,\underline{u}^{t} = 0 \qquad \text{Eq. (97)}$$

Then solve out $\underline{u}^{t+\Delta t}$:

$$\underline{u}^{t+\Delta t} = -\underline{u}^{t-\Delta t} + 2\underline{u}^{t} - \frac{\Delta t^2 \underline{K}\,\underline{u}^{t}}{\underline{M}} \qquad \text{Eq. (98)}$$

Or

$$u_i^{t+\Delta t} = -u_i^{t-\Delta t} + 2u_i^{t} + const0_i \times Ku_i \qquad \text{Eq. (99)}$$

where $const0_i = -\frac{\Delta t^2}{\rho tab \times M_{i,i}}$

4.2.3 3D Tetrahedral Element

The 3D finite element method can handle greater detail and more complex characterizations of construction materials, thus a powerful and realistic analysis of almost any structure can be provided. An element by element approach is a promising one to use on FPGA based hardware accelerators due to the simplicity of data structures required, and also the minimization of communications overheads. In this sub-section, a tetrahedral element, which is widely used in 3D finite element analysis, is introduced.

4.2.3.1 3D Mesh Generation

In finite element analysis, the domain of the problem is discretized into a finite number of sub-regions or sub-domains firstly. Tetrahedrons are widely employed in 3-D finite element analysis to achieve the geometric discretization of the problem domain. Figure 20 shows a simple example of the discretization of the three dimensional solution domain into a mesh of 4-noded tetrahedral elements. Firstly, a problem is subdivided into a number of scaled hexahedrons. Then the hexahedron will be subdivided into six tetrahedrons.

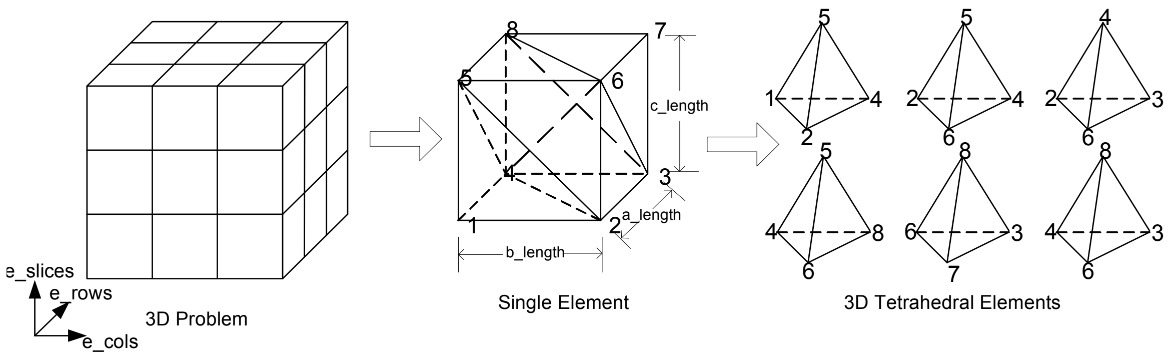


Figure 20 : Subdivision of the domain into Three-Dimensional Tetrahedral Elements

Then one of the six tetrahedrons in Figure 20 is assumed with global coordinates (x_i, y_j, z_k) and the displacement at the nodes are (u_1, v_1, w_1) (u_2, v_2, w_2) (u_3, v_3, w_3) (u_4, v_4, w_4) respectively, as in Figure 21.

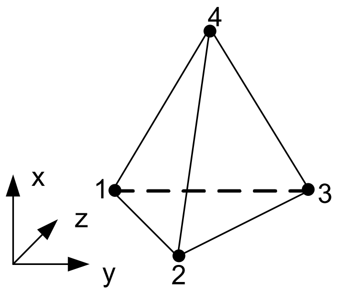


Figure 21 : 3D Tetrahedral Element

Next step is the key finite element approximation – Assume that the displacement at any point in the element depends only on the displacement of the nodes therefore

$$u(x,y,z) = \sum_{i=1}^{4} N_i(x,y,z)u_i \qquad \text{Eq. (100)}$$

$$v(x,y,z) = \sum_{i=1}^{4} N_i(x,y,z)v_i \qquad \text{Eq. (101)}$$

$$w(x,y,z) = \sum_{i=1}^{4} N_i(x,y,z)w_i \qquad \text{Eq. (102)}$$

where N_1 to N_4 are called the shape functions.

A simple set of shape functions are:

$$N_1(x,y,z) = \frac{1}{6V}\begin{vmatrix} 1 & x & y & z \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} = a_1 + b_1 x + c_1 y + d_1 z = \frac{1}{6V}\left(\begin{vmatrix} x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{vmatrix} + x\begin{vmatrix} y_2 & z_2 & 1 \\ y_3 & z_3 & 1 \\ y_4 & z_4 & 1 \end{vmatrix} + y\begin{vmatrix} z_2 & 1 & x_2 \\ z_3 & 1 & x_3 \\ z_4 & 1 & x_4 \end{vmatrix} + z\begin{vmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} \right) \qquad \text{Eq. (103)}$$

$$N_2(x,y,z) = \frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x & y & z \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} = a_2 + b_2 x + c_2 y + d_2 z \qquad \text{Eq. (104)}$$

$$N_3(x,y,z) = \frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x & y & z \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} = a_3 + b_3 x + c_3 y + d_3 z \qquad \text{Eq. (105)}$$

$$N_4(x,y,z) = \frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x & y & z \end{vmatrix} = a_4 + b_4 x + c_4 y + d_4 z \qquad \text{Eq. (106)}$$

or

$$N_i(x,y,z) = a_i + b_i x + c_i y + d_i z \qquad \text{Eq. (107)}$$

with $6V = \begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix}$.

The mass matrix is given by

$$\underline{M}_e \underline{\ddot{u}}_e = \rho \begin{bmatrix} \int_{-c}^{c}\int_{-b}^{b}\int_{-a}^{a} N_i N_j dxdydz & 0 & 0 \\ 0 & \int_{-c}^{c}\int_{-b}^{b}\int_{-a}^{a} N_i N_j dxdydz & 0 \\ 0 & 0 & \int_{-c}^{c}\int_{-b}^{b}\int_{-a}^{a} N_i N_j dxdydz \end{bmatrix} \begin{pmatrix} \ddot{u}_j \\ \ddot{v}_j \\ \ddot{w}_j \end{pmatrix} \qquad \text{Eq. (108)}$$

Using nodal integration, we have

$$\underline{M}_e \underline{\ddot{u}}_e = \frac{\rho V}{4} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} \ddot{u}_j \\ \ddot{v}_j \\ \ddot{w}_j \end{pmatrix} \qquad \text{Eq. (109)}$$

The derivation of mass matrix and stiffness matrix can be found in [101] and [102]. The stiffness matrix is in the form of

$$\underline{K}_e \underline{u}_e = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)} \int_V \begin{bmatrix} \frac{\partial N_i}{\partial x} & 0 & 0 & 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} \\ 0 & \frac{\partial N_i}{\partial y} & 0 & \frac{\partial N_i}{\partial z} & 0 & \frac{\partial N_i}{\partial x} \\ 0 & 0 & \frac{\partial N_i}{\partial z} & \frac{\partial N_i}{\partial y} & \frac{\partial N_i}{\partial x} & 0 \end{bmatrix} \begin{bmatrix} 1 & \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & 1 & \frac{\nu}{1-\nu} & 0 & 0 & 0 \\ \frac{\nu}{1-\nu} & \frac{\nu}{1-\nu} & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2(1-\nu)} \end{bmatrix} \begin{bmatrix} \frac{\partial N_j}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_j}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_j}{\partial z} \\ 0 & \frac{\partial N_j}{\partial z} & \frac{\partial N_j}{\partial y} \\ \frac{\partial N_j}{\partial z} & 0 & \frac{\partial N_j}{\partial x} \\ \frac{\partial N_j}{\partial y} & \frac{\partial N_j}{\partial x} & 0 \end{bmatrix} dV \begin{pmatrix} u_j \\ v_j \\ w_j \end{pmatrix} \quad \text{Eq. (110)}$$

4.2.3.2 Preconditioned Conjugate Gradient method

As mentioned in subsection 3.3.2, iterative methods are very efficient and suitable for the numerical solution of three dimensional finite element analysis. Moreover, the conjugate gradient method is an effective algorithm for the numerical solution of symmetric positive definite systems $Ax = b$ [86]. The method proceeds by generating vector sequences of iterates, residuals corresponding to the iterates, and search directions used in updating the iterates and residuals. If the coefficient matrix **A** is ill-conditioned, i.e. it has a large condition number, it is often useful to use a preconditioning matrix **M**, where $M^{-1} \approx A^{-1}$ and solve the system $M^{-1}Ax \approx M^{-1}b$ instead as the rate of convergence for iterative linear solvers usually increases as the condition number of a matrix decreases. The main purpose of preconditioning is to reduce the condition number of the coefficient matrix. The simplest preconditioning matrix **M** is Jacobi preconditioning, which is a diagonal matrix comprising the diagonal elements of **A** only.

$$M = \begin{cases} A_{ii} \cdots i = j \\ 0 \cdots otherwise \end{cases}, \qquad \text{Eq. (111)}$$

The algorithm of the preconditioned Conjugate Gradient method is shown below[87].

1. Make an initial estimate x_0

2. $r_0 = b - Ax_0$ Eq. (112)

$h_0 = M^{-1} r_0$

$p_0 = h_0$

3. For k=0, 1, 2 …

$$\omega_k = Ap_k \qquad \text{Eq. (113)}$$

$$\alpha_k = \frac{h_k^T r_k}{p_k^T \omega_k} \qquad \text{Eq. (114)}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k \omega_k \qquad \text{Eq. (115)}$$

(Until $\left\| r^k \right\|_2 \le \varepsilon_1 \left\| b \right\|_2$, where $\|\cdot\|$ denotes the L2-norm, i.e. the Euclidean norm, or $\left\| x^{k+1} - x^k \right\|_2 \le \varepsilon_2 \left\| x^k \right\|_2$)

$$h_{k+1} = M^{-1} r_{k+1}$$

$$\beta_k = \frac{h_{k+1}^T r_{k+1}}{h_k^T r_k}$$

$$p_{k+1} = h_{k+1} + \beta_k p_k$$

4.2.3.3 Software Implementation

A software simulator for finite element analysis has been developed for this study. It has the same facility as FDM simulator to communicate with the reconfigurable computing platform.

Generally, the first thing the simulator does is to initialise the software environment, i.e. the selection of element type and the dimension of the solving problem. After that, the material property parameters, such as E, ν, ρ, α, l, can be supplied. Then the stiffness matrices and mass matrices are assembled by assigning the connectivity of elements. With the boundary conditions and loads, the processing can start to solve the system equations. Due to the domain decomposition scheme used to achieve greater parallelism, a run time comparison between the sequential program and the parallel implementation on reconfigurable computing boards will be carried out.

4.2.3.4 Domain Decomposition

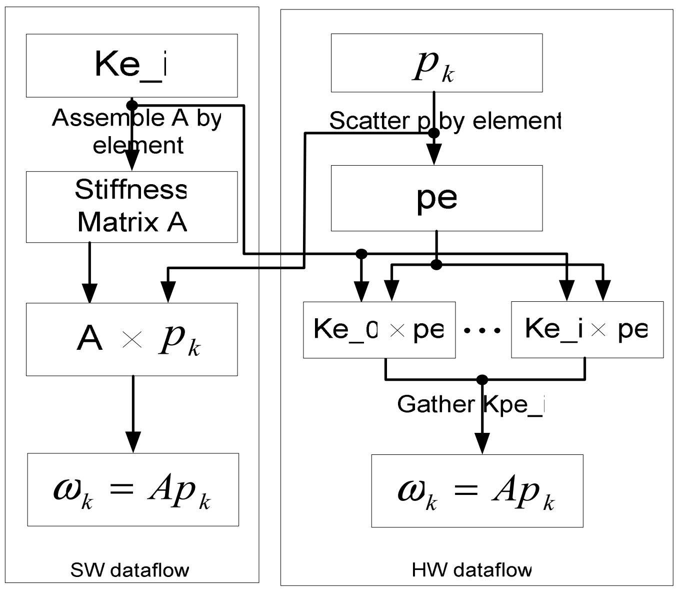


Figure 22 : Matrix Multiplication Parallelization

Because the resulting system is characterized by a global stiffness matrix which is usually large and sparse, but needs to be generated only once, the software host is used to scatter the whole problem into a series of sub-problems that can be fed into parallel computing systems. The element-by-element (EBE) solution scheme was studied since 1983, as given in [16] [17] [88] [89], where the element contributions are computed independently.

Based on the three-dimensional tetrahedral finite elements in Figure 20, the global stiffness matrix *A* is built up by assembling the element stiffness matrices K_e. Thus, the matrix-vector multiplication Eq. (113) can be modified as shown in Figure 22. The data flow on the left-hand side is for the normal software solution, in which the global stiffness matrix A is assembled first, and then the matrix multiplication is performed. The element-by-element approach used for parallel computing is shown on the right-hand side of Figure 22. Instead of assembling the global stiffness matrix A, each local stiffness matrix K_e is directly multiplied its element vector p_{e_i}, which is scattered from the conjugate gradient vector p_k. Then the vector Kp_{e_i} is gathered together in order to obtain the same final result as $\omega_k = Ap_k$.

The basic element-by-element (EBE) solution strategy is described below:

```
1.    Preparation
      For elements $e = 0,1,2,\ldots$, do
            obtain element matrix and vector $b_e$
            apply appropriate boundary conditions
      End do
2.    Initial $x_0$
         Calculate $r_0 = b - Ax_0$
         Preconditioning $h_0 = M^{-1}r_0$
               $p_0 = h_0$
3.    For $k = 0,1,2,\ldots$ iterate
         Scatter $p$
         Element-by-element matrix-vector multiplication $\omega_{ke} = k_e p_e$
         Assemble $\omega_k$ from element-by-element $\omega_{ke}$
```

$$\alpha_k = \frac{h_k^T r_k}{p_k^T \omega_k}$$

$$x_{k+1} = x_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k \omega_k$$

(Until $\left\| r^k \right\|_2 \leq \varepsilon_1 \left\| b \right\|_2$, where $\|\cdot\|$ denotes 2-norm (Euclidean norm or $\left\| x^{k+1} - x^k \right\|_2 \leq \varepsilon_2 \left\| x^k \right\|_2$)

$$h_{k+1} = M^{-1} r_{k+1}$$

$$\beta_k = \frac{h_{k+1}^T r_{k+1}}{h_k^T r_k}$$

$$p_{k+1} = h_{k+1} + \beta_k p_k$$

.

By using this domain decomposition technique, irregular mesh geometries of different shaped domains can be handled in order to make the method completely general and reduce the communication overheads in the parallel computing system.

4.3 Evaluation of the Software Implementation

4.3.1 The Finite Difference method

A software simulator based on the methods of section 4.1 was written in C language and compiled by Microsoft Visual C++ NET. 2003 in release mode. The simulator is capable of solving 3D problems under a variety of Neumann and Dirichlet boundary conditions. The simulator is used as a baseline for comparison with the hardware methods that will be developed in chapter 5. The simulator uses IEEE 754 standard double floating point arithmetic and was run on a 2.4 GHz Pentium 4 PC with 1GByte RAM.

This section presents an evaluation of the different approaches detailed in section 4.1, considering both accuracy and CPU time requirements. The evaluation was carried out on a 3D unit cubic domain using a set of boundary conditions that was chosen such that an exact analytical solution of the equation can be derived. This exact analytical solution was used as the baseline for accuracy analysis on the various numerical approaches. The Dirichlet boundary conditions used were as follows:

$$u(x,y,1) = \sin(2m\pi x)\sin(2m\pi y) \qquad \text{Eq. (116)}$$

$$u(x,y,z) = 0 \quad \text{for} \quad x=[0,1],\ y=[0,1] \ \text{and} \ z=[0,1) \qquad \text{Eq. (117)}$$

All other boundaries used a Neumann boundary condition (i.e. the derivative of u normal to the boundary was zero). As a result of these boundary conditions, all values of u within the cubic domain will lie in the range 0 to 1. The number of grid lines used for the discretization varied from 4 to 256 in each dimension.

The operation of the simulator is shown in Figure 17, the solution approaches used are as follows:

E3	Exact solution for 3D Laplace equation
F_FDM2	Solution via 1D FFT in x direction and 2D FDM
FDM3	Solution via 3D FDM
FF_FDM1	Solution via FFT in x direction, FFT in y direction, and then 1D FDM
F_E2	Solution via FFT in x direction, then 2D exact solution
FF_E1	Solution via FFT in x direction, FFT in y direction, and 1D exact solution

Table 2 compares the precision of each method to the exact solution and Table 3 shows the CPU time consumed for each method.

Table 2 : Maximum Relative Error between Domain Decomposition Methods and Exact Solution for N×N×N cubes.

Cube Size	F_FDM2	FDM3	FF_FDM1	FF_E1	F_E2
4×4×4	0.4615	0.5820	0.0000	0.3848	0.0000
8×8×8	0.06471	0.07812	0.0000	0.05633	0.0000
16×16×16	0.01737	0.02088	0.0000	0.01423	0.0000
32×32×32	0.004431	0.005318	0.0000	0.003566	0.0000
64×64×64	0.000974	0.001169	0.0000	0.000195	0.0000
128×128×128	0.000244	0.000293	0.0000	0.000195	0.0000
256×256×256	0.000063	0.000076	0.0000	0.000051	0.0000

The relative error of the finite difference approximation results from round-off error and discretization error. The round-off error comes from the loss of precision due to computer rounding of decimal quantities, while the discretization error depends on the finite approximation Δx and Δy. As the number of sub-divisions for the sides of the cube increases, a more accurate approximation is obtained and the discretization error becomes smaller.

It can be easily seen that although F_FDM2 is not the best of the different methods, it does perform well when the problem size becomes larger, thus demonstrating that the 2D finite difference method is satisfactory for large systems.

The F_E2 approach, where an FFT is applied in the x direction followed by a 2D exact solution, the result is identical to the exact solution of the 3D problem. This demonstrates that the use of a discrete FFT in solving such problems does not cause a loss of accuracy. The difference between the F_E2 and E3 approaches is just that one solves the problem in the space domain, and the other uses the spatial "frequency" domain using the discrete Fourier Transform, which is better as we can analyze the 3 dimension problem as a series of 2 dimension problems. Of course, the exact analytical solution is only possible because of the contrived choice of boundary condition: in the general cause of more usual boundary conditions the 2D exact solution is not available, so the method F_E2 is not generally applicable.

The FF_FDM1 approach, using the FFT twice and then a 1D-FDM, also produces a good result, and gives the same answer as the exact solution. Based on the above analysis, Fourier decomposition maintains the numerical property of PDEs, and the 3-dimensional problem is divided into n 2-dimensional sub-problems which are independent to each other. Therefore, the parallel solution to PDEs using the finite difference method in this case can be easily fitted onto parallel computing systems.

The FF_FDM1 approach imposes limitations on 2 of the dimensions (the grid resolution must be an exact power of two, and the material properties should be invariant in those directions). The F_FDM2 is therefore chosen for implementation in hardware as it provides a good compromise between generality and decomposition of the big problem into parallel smaller problems.

Table 3 : Timing Cost of Different Methods for N×N×N cubes (Jacobi iteration)

Timing (CPU seconds)	E3	F_FDM2	FDM3
4×4×4	6.2×10^{-5}	1.4×10^{-4}	6.4×10^{-4}
8×8×8	2.1×10^{-4}	4.2×10^{-4}	1.8×10^{-3}
16×16×16	1.3×10^{-3}	3.3×10^{-3}	4.4×10^{-2}
32×32×32	7.8×10^{-3}	3.8×10^{-2}	1.1×10^{0}
64×64×64	7.8×10^{-2}	4.7×10^{-1}	2.9×10^{1}
128×128×128	4.4×10^{-1}	5.9×10^{0}	7.0×10^{2}
256×256×256	3.5×10^{0}	7.2×10^{1}	1.7×10^{4}

From Table 3, it can be seen that exact solution has the lowest CPU time consumption. However, most PDEs do not have an exact analytical solution so this method is not generally applicable. It can be seen from Table 3 that the performance of the finite difference method with Fourier decomposition (F_FDM2) becomes much better than the standard 3D finite difference approach (FDM3) when the number of grid points becomes large. Additionally, the

F_FDM2 approach will be much more suitable for parallelisation on an FPGA than the FDM3 approach. So in hardware the advantage of F_FDM2 will be even larger.

Table 4 : Timing cost for iterative methods of F_FDM2 (Release mode)

Timing (CPU seconds)	Jacobi	Gauss-Seidel	SOR (ω=1.5)
4×4×4	1.1×10^{-4}	1.5×10^{-4}	1.7×10^{-4}
8×8×8	3.7×10^{-4}	3.5×10^{-4}	3.1×10^{-4}
16×16×16	2.9×10^{-3}	2.9×10^{-3}	1.1×10^{-3}
32×32×32	2.8×10^{-2}	2.7×10^{-2}	9.3×10^{-3}
64×64×64	3.4×10^{-1}	3.1×10^{-1}	1.1×10^{-1}
128×128×128	4.1×10^{0}	3.5×10^{0}	1.3×10^{0}
256×256×256	5.2×10^{1}	4.1×10^{1}	1.4×10^{1}

Table 4 compares the timing cost, which was obtained from the software simulator compiled in release mode by Microsoft Visual .NET 2003, for the F_FDM2 approach using three different relaxation methods, Jacobi iteration, Gauss-Seidel iteration and successive over-relaxation (SOR) iteration.

Based on the above analysis, F_FDM2 is the algorithm that is chosen for implementation in hardware. The 3-dimensional domain is decomposed into n 2-D sub-domains, which are computationally independent, and the 2D sub-domains are computed in hardware.

4.3.2 Finite Element Method

A software simulator was written for finite element solution of a 3D Laplace equation using the method shown in section 4.2.3. This software simulator was used to provide the baseline against which the hardware implementation would be evaluated. The simulator used IEEE 754 standard double floating point arithmetic and was run on a 2.4 GHz Pentium4 PC with 1GByte RAM. In this section, results are presented showing how the various iteration methods perform on problem formulations with varying numbers of elements.

Table 5 : The number of iterations using different iteration methods for the FEM.

Size(elems)	SW(Jacobi)	SW(CG)	SW(PCG_global)	SW(PCG_EBE)
6	219	4	4	4
48	980	19	18	18
384	4004	56	39	39
750	6264	70	49	49
3,072	16018	102	78	78
6,000	24994	121	95	95
24,576	100000	175	150	150
48,000	NA	211	188	188

Table 5 shows the number of iterations required for the Jacobi method, conjugate gradient method and preconditioned conjugate gradient methods to achieve convergence. The final results of both the hardware (HW) version and the software (SW) version were identical, i.e. error=0. The Jacobi method has a very poor convergence characteristic in comparison with the preconditioned conjugate gradient method. It can be seen from Table 5 that the element-by-element (EBE) method (which is the method that has been chosen for hardware implementation) entails no penalty compared to the more standard global formulation method. Table 6 shows the timing cost for different size problems implemented in software using different iterative methods. The Jacobi method converges slowly, and the method failed to converge when the matrix size becomes larger. Due to the use of the preconditioning matrix, preconditioned conjugate gradient method achieves a better performance than the Conjugate Gradient (CG) method, as the preconditioned Conjugate Gradient method needs fewer iterations to converge.

Table 6 : Timing (in seconds) Comparison in Different Methods

Size(elems)	SW(Jacobi)	SW(CG)	SW(PCG_global)	SW(PCG_EBE)
6	0.002575	0.000109	0.000084	0.000321
48	0.074731	0.001985	0.001871	0.003447
384	2.33013	0.036038	0.023366	0.045372
750	8.99663	0.079769	0.058451	0.111514
3,072	74.1568	0.481076	0.362953	0.826589
6,000	224.388	1.16750	0.864280	1.84944
24,576	4055.25	6.53732	5.64370	12.6824
48,000	NA	15.8564	19.1038	31.7281

4.4 Summary

In this chapter, the software solvers for the finite difference method and the finite element method have been presented. Also the performance of the different methods has been compared. The problem formulations chosen for solution on the parallel computing systems due to their good balance between computation and communication have been discussed. Domain decomposition was introduced to maximize the independence of the processing elements and to minimize the communication overhead.

In the following chapters, the design and evaluation of several reconfigurable computing approaches based on the algorithms discussed in this chapter will be presented.

Chapter 5

FPGA-BASED HARDWARE IMPLEMENTATIONS

5.1 Introduction

The previous chapter outlined the specification for the system design. This chapter deals with the implementation of the designs on hardware, and presents the memory hierarchy and data caching structures needed to satisfy the computational performance requirements.

Software numerical algorithms have been migrated onto FPGA-based co-processors in a relatively straightforward manner: while leaving almost all software subroutines still running on the commodity CPU of the host machine, only the most time-consuming kernel portion of the programs are replaced by subroutines calling for the help of FPGAs.

Reconfigurable computing based on FPGAs has become regarded as acceptable for problems in computational mechanics that require floating-point arithmetic in order to achieve numerical stability and acceptable precision. In 1994, reference [90] showed the feasibility of implementing IEEE Standard 754 single precision floating-point arithmetic units on FPGAs for the first time. Since then, with the rapid growth of FPGAs in density and speed, as well as the introduction of on-chip ALU units that are optimized for DSP operations, highly complex systems using floating-point arithmetic can be implemented within modern FPGAs. However, compared to fixed-point arithmetic, floating-point arithmetic operations require a lot more logic resource and have a lower speed. The reason why fixed-point arithmetic is generally regarded as undesirable is because floating-point arithmetic can represent a wider dynamic range[2] and obtain more accurate results for PDE solution. Some recent research efforts have attempted to offset the undesirable features of fixed point arithmetic in order to achieve a balance between accuracy and efficiency. Examples include multiple word-length optimisation [91], which was extended to differentiable nonlinear systems in [92], and the Dual FiXed-point (DFX) approach [93], which combined conventional fixed-point and floating-point representations, and was used to simplify and speed up IIR filter implementation. The performance of the power, area and speed in some hardware designs can be efficiently increased by the use of fixed point arithmetic. For the numerical methods considered in this research, both the finite difference method and the finite element method are widely used to obtain numerical approximations for a wide variety of PDEs. For an unknown numerical solution, in order to maintain a high accuracy, the use of floating-point arithmetic is necessary. This chapter will describe approaches that use both fixed-point arithmetic and floating-point arithmetic within the reconfigurable hardware accelerators, as different architectures are formulated in order to explore the maximum extent of parallelism that can be achieved within the FPGAs.

[2] Dynamic range is defined as the ratio between the maximum absolute value representable and the minimum positive (i.e. non-zero) absolute value representable

5.2 System Environment

5.2.1 Software Interface

This section will describe the software interface for the RC2000 (ADM-XRC-II) board [94], a PCI-based reconfigurable coprocessor developed by Alpha-data. The initial pre-processing is carried out in the software implementation which was explained in chapter 3. The boundary conditions are also generated for a particular problem. Then, this data is fed into either the Software Simulator or the Hardware Simulator. Finally, the precision of results from each of the two simulators and the timing costs for the different solutions are compared. This procedure, which is used in all the following designs, is illustrated in Figure 23.

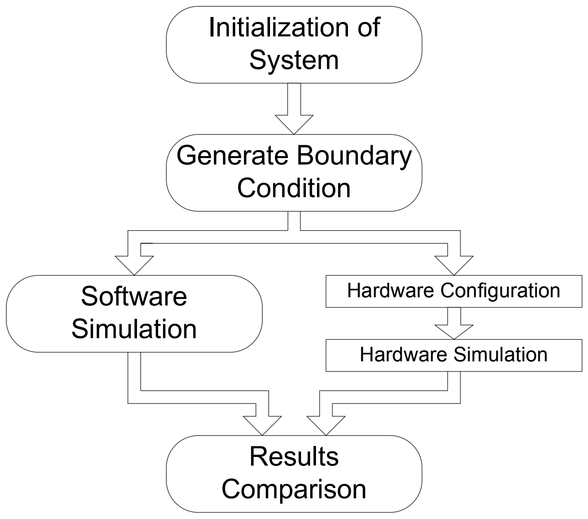


Figure 23 : Software-Hardware Design Flow.

As shown in Figure 23, the steps in rounded ovals run in software, i.e. on the PC's microprocessor, as formulated in chapter 4. On the other hand, the steps in rectangles run in hardware, i.e. on the FPGA-based reconfigurable computing board. Before the Hardware Simulator runs, the appropriate FPGA configuration file (the bit file) is needed to configure the FPGA on the RC2000 boards, the clock rate at which the FPGA should be operated is set, and the initialisation data and boundary conditions are transferred into the boards' SRAM banks.

5.2.2 Hardware Platform

The hardware implementations are loaded into two Celoxica RC2000 PCI bus plug-in cards equipped with one single Xilinx Virtex 2V6000 FPGA and one single Xilinx Virtex 4VLX160 FPGA [95] respectively. Figure 24 shows the detail of the 2V6000 implementation platform [96], it is equipped with one FPGA and 24 Mbytes static RAM arranged in 6 banks that can be read or written simultaneously. The board plugs into a host microprocessor system and exchanges data with the host memory across the PCI bus.

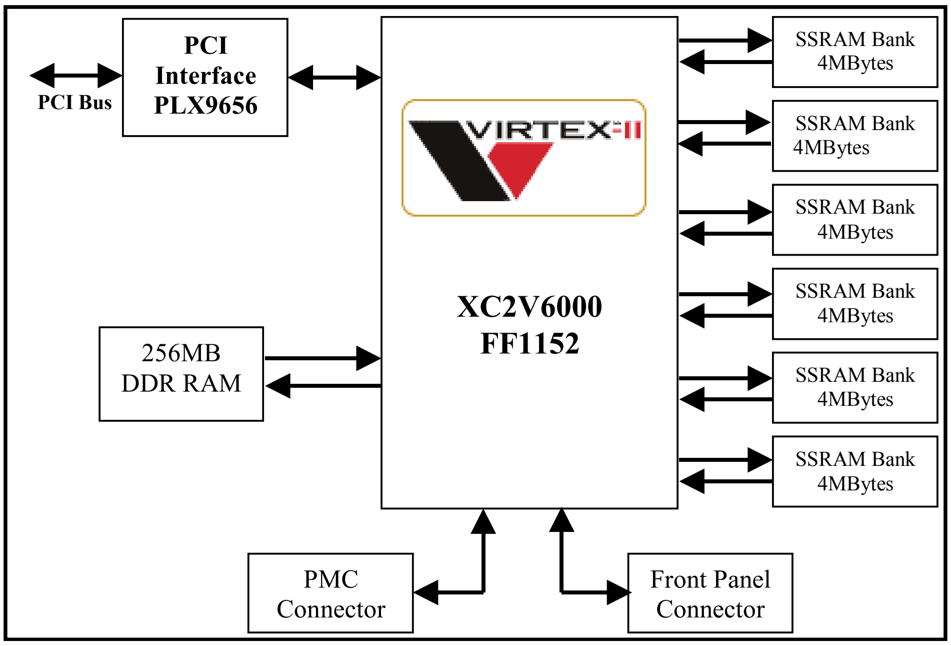


Figure 24 : Block diagram of the RC2000[97].

The card with the Virtex 2V6000 FPGA is plugged into a 2.4 GHz Pentium 4 PC with 1GByte RAM, and the card with the Virtex 4VLX160 FPGA is plugged into a 2.01 GHz Athlon 64 Processor PC with 1GByte RAM.

The RC2000 is a 64 bit PCI card utilising a PLX-9656 PCI controller. It is capable of carrying either one or two mezzanine boards; in our case it hosts a single ADM-XRC-II board from Alpha-Data [94]. The mezzanine board carries the XC2V6000-4, 24Mbytes of SSRAM and 256Mbytes of DDR memory, along with PMC and front panel connectors for interacting with external hardware. The SSRAM is arranged in six 32-bit wide banks. However the FPGA sits between it and the host, so a portion of the FPGA is always instantiated to act as a memory control system, arbitrating between host access and FPGA access to this shared resource. The control system implemented allows the host both DMA transfer and virtual address access to the SSRAM and the six banks are independently arbitrated to allow greater design flexibility.

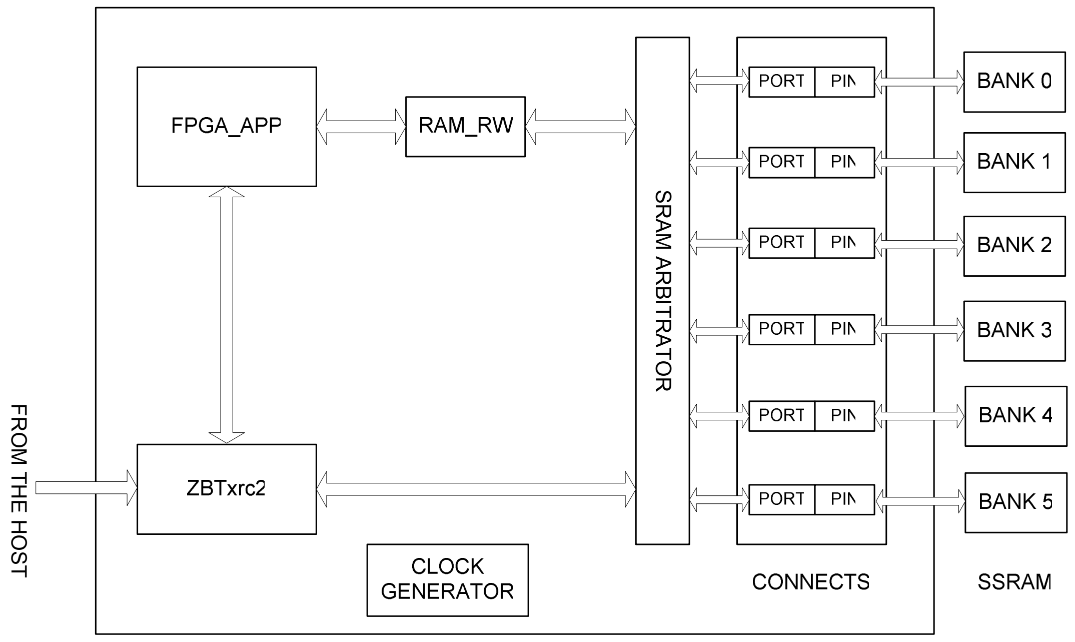


Figure 25 : Block diagram of RC2000 Interface hardware components [98].

Figure 25 shows a diagram of the hardware interface, which includes 6 main hardware components [98]:

1. *Clock Generator* uses digital clock managers to derive the system clock (clk) and phase aligned frequency multiplied version thereof (clk2X and clk4X) from an externally applied master clock that is generated on the board. It also generates 2 appropriately de-skewed clock signals for the SRAMs.
2. *Connects* provides the correct timings for the ZBT SRAM read and write operations.
3. *ZBTxrc2* handles data transfer between the host and the FPGA.
4. *SRAM_arbitrator* controls the arbitration of all six SRAM banks between the host and the FPGA.
5. *RAM_RW* generates the 'ready' signal indicating the data read from SRAM is valid on the data port.
6. *FPGA_APP* is the framework which contains the user's top level entity.

Figure 26 shows the interface of the hardware design in Synplify RTL view.

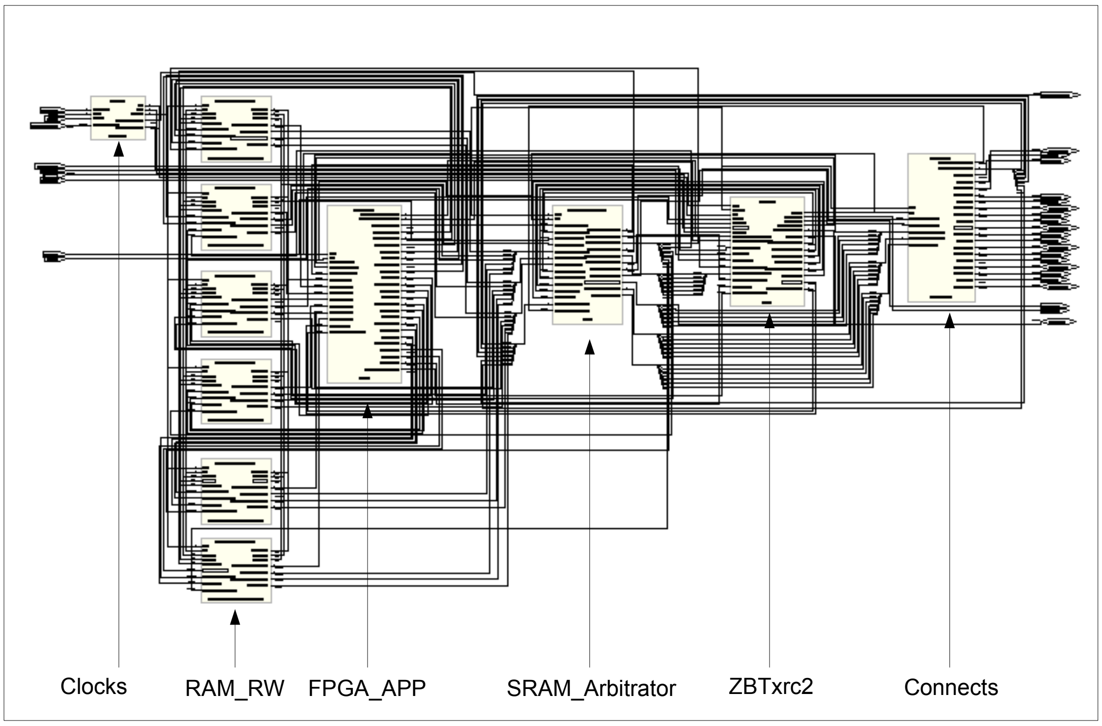


Figure 26 : RTL view of hardware design

5.2.3 Data Format

5.2.3.1 Customised Fixed-Point Data Format

A binary fixed-point number is usually written as I.F, where I represents the integer part, '.' is the radix point, and F represents the fractional part. Each integer bit represents a power of two, and each fractional bit represents an inverse power of two. A fixed-point data type represents numbers within a finite range; thus positive and negative overflows must be taken care of if there is any result of an operation that lies outside the appropriate range. In order to make the design fit onto the available FPGAs, a customised 32-bit data format was chosen.

The use of fixed-point arithmetic reduces the complexity of the computational pipelines within FPGAs, thereby allowing a greater level of parallelism (and thus performance) to be achieved. The Laplace equation has the property that the steady state solution values are bounded above and below by the largest and smallest Dirichlet boundary condition respectively, with the result that the required dynamic range for the numerical values is easy to predict a priori. This means that fixed-point is relatively safe. Trials were carried out to determine that the fixed point range selected for use in the implementation is greater than that required to be equivalent to a single precision floating-point representation, without overflow.

The customised 32-bit fixed-point data format is shown below in Figure 27:

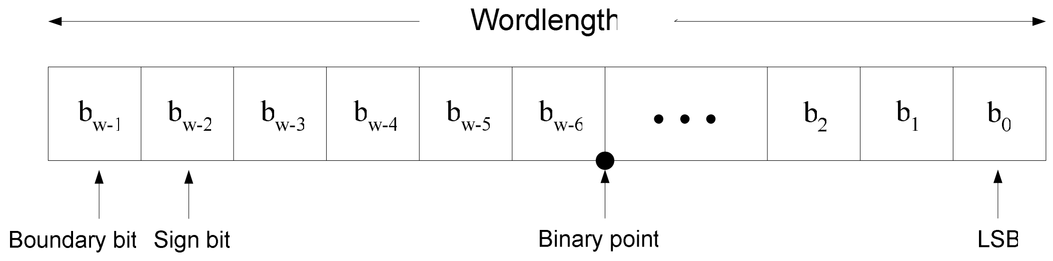


Figure 27 : Customised fixed-point data format.

where

- b_i is the i^{th} binary digit.
- w is the word length in bits.
- b_{w-1} represents the boundary flag for the FDM.
- b_{w-2} is the sign bit.
- b_{w-3} is the most significant bit.
- b_0 is the least significant bit.

- The binary point is 26 places to the left of the LSB, as a fractional part of 2^{-26} gives sufficient precision. The integer part (excluding the sign bit) is 4 bits wide, which means that a maximum number of ±16 can be represented.

A converter is used to convert the software IEEE 754 format into the customized fixed-point data format, this is implemented within the software.

5.2.3.2 Floating-Point Representation

The IEEE 754 [99] floating point format, which is the standard usually used on computers for 32-bit single precision variables and 64-bit double precision variables, is discussed here. Normally scientific work requires floating-point precision. Figure 28 shows the format of an IEEE 754 floating-point value. Binary floating-point numbers are stored in a sign magnitude form where the most significant bit is the sign bit (s), exponent is the biased exponent (e), and fraction is the significant without the most significant bit (f). For single precision binary floating-point, the number is stored in 32 bits, w=32, we=8 (exponent), and wf =24 (mantissa). Similarly, double precision is stored in 64 bits, w=64, we=11 (exponent), and wf =52 (mantissa).

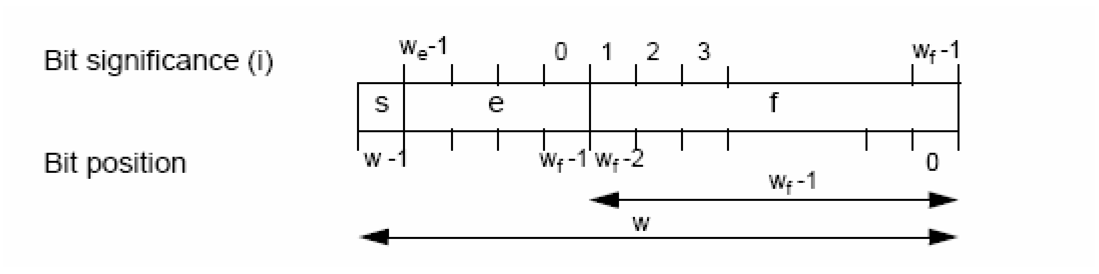


Figure 28 : Bit Fields within the Floating-Point Representation

There have been several efforts to develop parameterizable floating-point cores for FPGAs [100-102]. The design software *Xilinx CORE Generator* was used in our designs. This contains a parameterized library of pre-designed useful design components called *cores*. For the floating-point arithmetic operators, the core can be customized to allow for any required value of w, w_f and w_e. There are also trade offs that can be made in terms of the latency and throughput of the operators [95].

One point that is worth noting is that a floating-point adder is a much more complicated circuit than a fixed-point adder. This is because the input operands for a floating-point adder must be shifted until they have the same exponent prior to addition, and the resulting output must then be shifted to be correctly normalized. This consumes a large amount of the FPGA logic resource, and can also reduce the maximum speed that the circuit can achieve. By contrast, fixed-point arithmetic requires no special circuitry to provide input alignment or output normalization, but fixed-point arithmetic is useful only for a relatively restricted class of problems where the dynamic range of the data is small and the problem has benign error propagation properties. Thus, floating-point arithmetic is considerably more widely used in scientific calculations.

5.3 Hardware Implementations

This section describes the hardware designs implemented in the FPGA-based reconfigurable computing platform within a general purpose PC. The first group of implementations applied several different iteration methods (detailed in section 4.1) using 32-bit customised fixed-point arithmetic. Use of customised fixed-point arithmetic allowed a very high degree of parallelism to be achieved. This is followed by the implementation of a 32-bit floating-point FDM solution. Due to limitations of area and speed, the memory hierarchy is rearranged to achieve maximum parallelism on the board with small communication overheads. Then the 1D and 2D rectangular finite element solutions, of section 4.2.1 and section 4.2.2, are implemented using Xilinx System Generator in Simulink, which only supports fixed-point arithmetic at this time. The final implementations show an element-by-element parallelisation of the 3D tetrahedral element FEM solutions, explained in section 4.2.3, using floating point arithmetic.

The hardware designs were implemented using VHDL (Very high speed integrated circuit Hardware Description Language) and 5 different IP (intellectual property) cores, which are block RAMs, fixed-point adders, fixed-point multipliers, floating-point adders and floating-point multipliers. Upon the completion of the design entry stage, the Synplify Pro 8.5.1 synthesis tool is employed to generate the logic as an EDIF netlist. Then this black-box EDIF netlist is used as an input into Xilinx ISE 8.1, a tool that maps the logic requirement to the physical resources of the FPGA. The Xilinx FPGA design flow is shown in Figure 29.

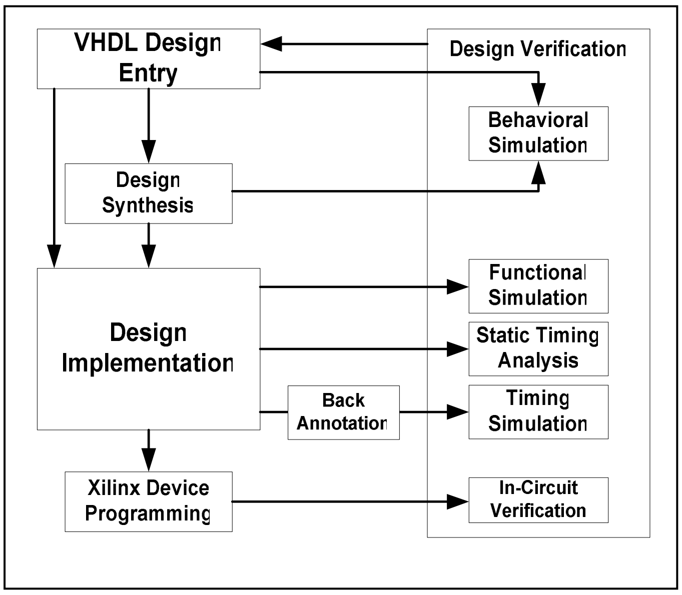


Figure 29 : Xilinx FPGA design flow [44].

5.3.1 FPGA implementation of the Finite Difference Method

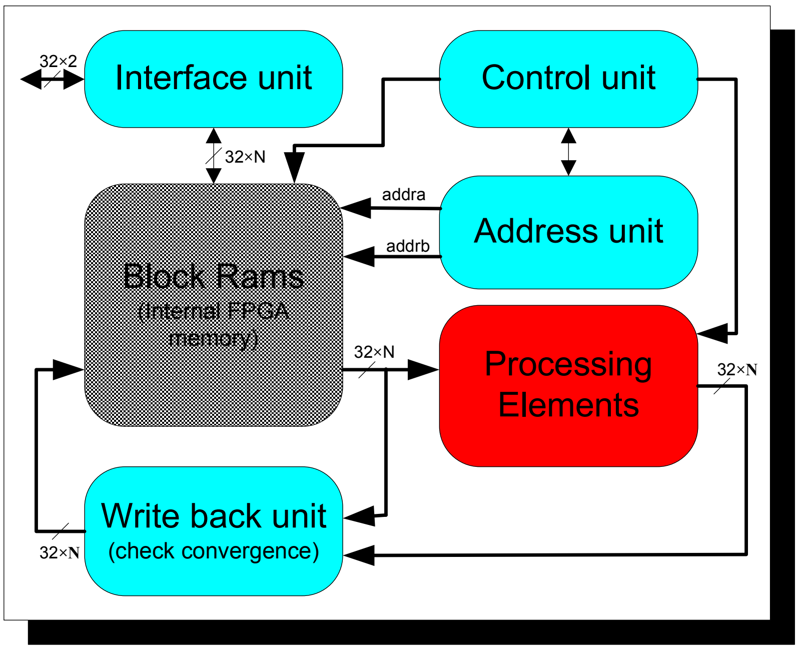


Figure 30 : FPGA implementation block diagram of FDM

Figure 30 shows a conceptual overview of the hardware implementation. There are five main units:

1. An *interface unit* to read and write data to and from the external memory (i.e. 6 SRAM banks on our computing boards).
2. A *control unit* to synchronize all the units and check the boundary conditions.
3. An *address unit*, which generates all the address signals (read and write).
4. A *write back unit* to write the results of the arithmetic units back to the Block RAMs and check for convergence of the processing.
5. *Processing elements*, which calculate out the results of the FDM.

The block RAM of the FPGA is used to hold the data of 2D slices, which are required for the finite difference method. Several memory architectures were implemented and are discussed in the following sections. One of the most challenging parts of the hardware design is to arrange the scheduling of internal memory accesses and the exchange of data between internal memory and external memory.

Using the formulation developed in section 4.1.2, a 3 dimensional problem has been decomposed into a series of nx 2D slices (corresponding to different values of m in Eq. (32)), which are decoupled and can be solved separately without any exchange data or synchronization required between the slices. By assuming $\Delta x = \Delta y$, the formula of the finite difference method reduces to:

$$u_m^{j,k} = \frac{u_m^{j-1,k} + u_m^{j+1,k} + u_m^{j,k-1} + u_m^{j,k+1}}{4 + \lambda^2 \Delta y^2} \qquad Eq.\ (118)$$

In the following subsections, Jacobi, Gauss-Seidel, successive over-relaxation and Red-black successive over-relaxation iteration methods are respectively implemented.

5.3.1.1 Fixed-Point Hardware Implementation

The fixed-point FDM hardware design was implemented on a Celoxica RC2000 board containing a single Xilinx Virtex 2V6000 FPGA with 24 Mbyte SRAM. The customised 32-bit fixed-point arithmetic was used here in order to reduce the complexity of the computational pipelines within the FPGA, so that a greater level of parallelism and performance could be achieved.

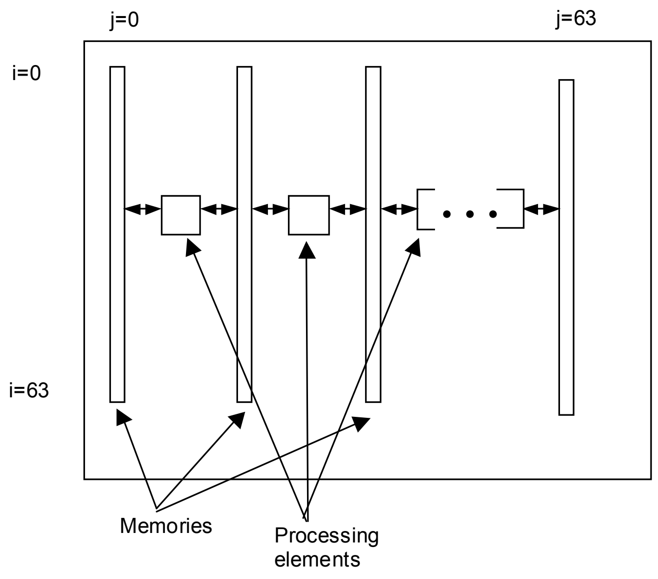


Figure 31 : Architecture of the Jacobi solver within the FPGA

Figure 31 shows the 32-bit customised fixed-point hardware design. The size of one 2D slice is nx=64 and ny=64. In order to have maximum parallelism achieved in hardware, there are 64 columns of memory, which are used to hold the columns of values and boundary flags. These are stored as 32-bit fixed-point. The memory columns are implemented in dual port block RAM, with the read addresses and write addresses capable of being incremented in each clock cycle. Processing elements are situated between the columns of memory with one element responsible for the calculations of each column. With reference to Eq. (47), the processing element used to perform the Jacobi update is shown in Figure 32.

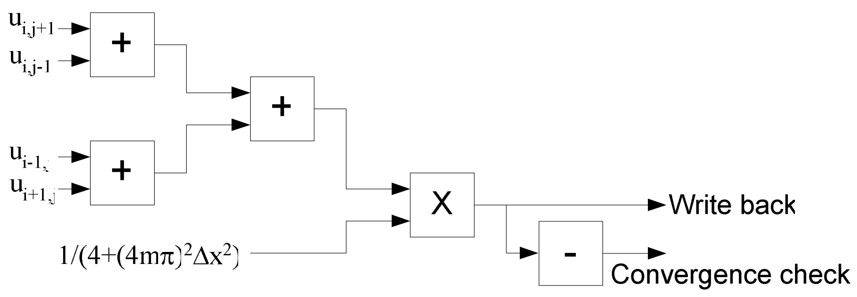


Figure 32 : Processing element for row update by using Jacobi iteration

It is partially from this parallelism of processing elements that hardware speed-ups are achieved. An iteration of the hardware matrix involves moving only down the number of elements in a column of N numbers whereas the software system iterates both across each row and down each column of N^2 numbers.

Gauss-Seidel iteration, which is able to converge faster due to the immediate use of the newly computed values, has also been implemented. The essential idea behind the architecture is similar to that of Figure 31 and Figure 32. Even faster convergence is is achieved by using successive over-relaxation iteration. The processing element is modified as in Figure 33.

However, the data path becomes more complicated because of the data dependency between rows of the memory, which means that new values cannot be fed into the pipelined data path on every clock cycle. Thus, full pipelining cannot be achieved by using either the

Gauss-Seidel iteration scheme or successive over-relaxation (SOR) iteration scheme, and a latency must be incurred between the processing of each row.

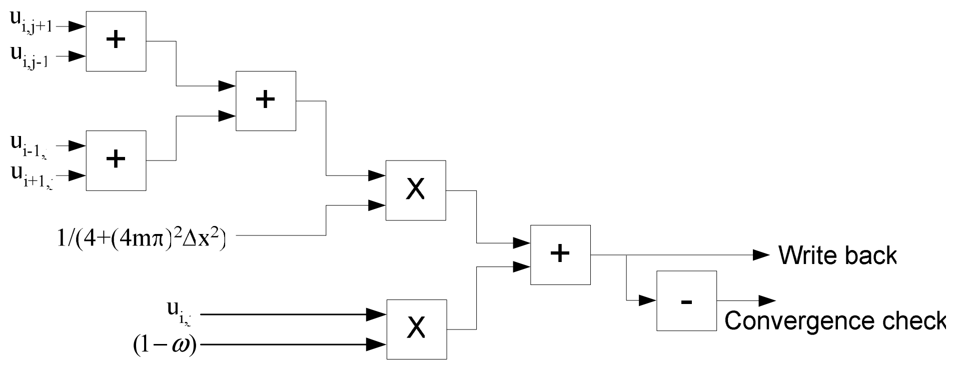


Figure 33 : Processing element for row update by using successive over-relaxation iteration

Table 7 shows a typical set of results that illustrates the speed of each method. The results were taken for a 32 × 32 × 32 3D Laplace equation. The table shows the number of clock cycles required to compute one 2D slice using an over-relaxation parameter of $\omega = 1.75$.

Table 7 : Performance of the different approaches

	Jacobi	Gauss-Seidel	SOR
Throughput (rows per clock cycle)	1	1/7	1/7
Iterations required for convergence	1338	919	197
Matrix passes required for convergence	1338	919	197
Clock cycles required for convergence	42821	205856	44128
Operations per second	9.6 billion	1.4 billion	1.4 billion
Memory bandwidth	7.7 GByte/s	1.1 GByte/s	1.1 GByte/s

The point Jacobi iteration method is very suitable for hardware implementation, giving a very high number of operations per second and sustained memory bandwidth utilization. Compared to the Jacobi method, the Gauss-Seidel and successive over-relaxation (SOR) methods have much superior numerical properties and converge in a lower number of iterations, but the pattern of data dependencies within the hardware means that the computation rate and memory bandwidth utilization drop significantly. As shown in Table 7, 205,856 clock cycles are required for Gauss-Seidel convergence, and 44,128 clock cycles are required for successive over-relaxation (SOR), whereas Jacobi just uses 42,821 clock cycles for convergence. Although the successive over-relaxation method converges almost 7 times faster than Jacobi, the performance of the hardware implementation of successive over-relaxation is slower due to the data dependence between rows of the memory. This is further exacerbated by the fact that the additional hardware complexity required for the Gauss-Seidel and successive over-relaxation (SOR) methods means that a smaller number of computational units can fit into the FPGA.

Next, the red-black successive over-relaxation scheme is considered in sub-section 5.3.1.3. The 2D finite difference grid is coloured with a red and black checkerboard, so the iteration scheme proceeds by alternating between update of the red squares and black squares. Use of the red-black successive over-relaxation method removes the requirement for each node to have immediate access to an updated value of two of its neighbours. In this case, the red-black successive over-relaxation gives an excellent compromise between the properties of the other iteration methods, which have either poor convergence characteristic or lower throughput (due to the pipeline data dependencies).

5.3.1.2 Floating-Point Hardware Implementation using Jacobi method

In 5.3.1.1, a solution using fixed-point arithmetic was implemented in order to conserve hardware resources. This would enable one entire 2-D section of the problem to fit in a single FPGA chip. Due to the area cost of floating point arithmetic, the FPGA chip cannot accommodate the whole of a single 2-D slice and its associated computational pipelines simultaneously. It is therefore necessary to use the FPGA to implement a smaller number of computational pipelines, and to read and write the slice data from and to the off-chip RAM. Communication is completely overlapped with computation, so the pipelines can always be doing useful work; this is not easily achievable with standard multi-purpose processors.

In this subsection, the Jacobi iteration scheme was used for the solution of the finite difference method using 32-bit floating-point arithmetic.

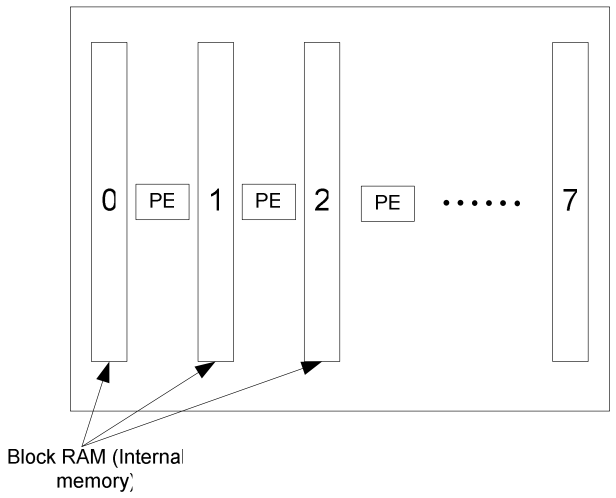


Figure 34 : Architecture of data path of Jacobi scheme

Figure 34 shows one of our hardware implementations with 8 columns of on-chip memory used within the FPGA. These are used as a cache that can hold up to eight columns of values for u, which are stored as 32-bit floating point. Initially these columns hold columns 0 to 7 of the domain. When column 0 has been processed, column 8 is loaded from external RAM into the column of block RAM previously occupied by column 0. This process then repeats with column 9 overwriting column 1 and so on. As a result each column will undergo 7 rounds of calculations between download and upload. This process continues until all columns have been completed.

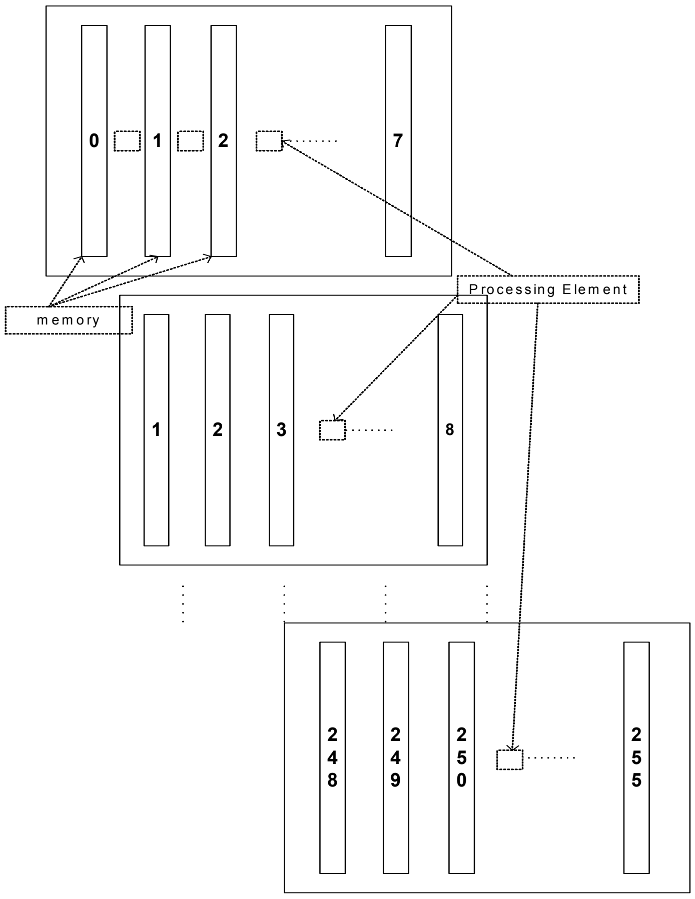


Figure 35 : Hardware architecture of data flow for the floating-point Jacobi solver

Figure 35 shows how the computation progresses. During each epoch, the contents of one Block RAM are streamed through the computation units.

Epoch 1

Columns held in Block RAM (internal memory)	none
Columns undergoing PEs	none
Columns undergoing Reac	0
Columns undergoing Update	none
Columns undergoing Write back	none

Epoch 2

Columns held in Block RAM (internal memory)	0
Columns undergoing PEs	none
Columns undergoing Reac	1
Columns undergoing Update	none
Columns undergoing Write back	none

Epoch 3

Columns held in Block RAM (internal memory)	0,1
Columns undergoing PEs	0,1
Columns undergoing Reac	2
Columns undergoing Update	1
Columns undergoing Write back	none

Epoch 4

Columns held in Block RAM (internal memory)	0,1,2
Columns undergoing PEs	0,1,2
Columns undergoing Reac	3
Columns undergoing Update	1,2
Columns undergoing Write back	none

Epoch 5

Columns held in Block RAM (internal memory)	0,1,2,3
Columns undergoing PEs	0,1,2,3
Columns undergoing Reac	4
Columns undergoing Update	1,2,3
Columns undergoing Write back	none

Epoch 6

Columns held in Block RAM (internal memory)	0,1,2,3,4
Columns undergoing PEs	0,1,2,3,4
Columns undergoing Reac	5
Columns undergoing Update	1,2,3,4
Columns undergoing Write back	none

Epoch 7

Columns held in Block RAM (internal memory)	0,1,2,3,4,5
Columns undergoing PEs	0,1,2,3,4,5
Columns undergoing Reac	6
Columns undergoing Update	1,2,3,4,5
Columns undergoing Write back	none

Epoch 8

Columns held in Block RAM (internal memory)	0,1,2,3,4,5,6
Columns undergoing PEs	0,1,2,3,4,5,6
Columns undergoing Reac	7
Columns undergoing Update	1,2,3,4,5,6
Columns undergoing Write back	0

Epoch 9

Columns held in Block RAM (internal memory)	8,1,2,3,4,5,6,7
Columns undergoing PEs	8,1,2,3,4,5,6,7
Columns undergoing Reac	8
Columns undergoing Update (write back)	2,3,4,5,6,7
Columns undergoing Write back	1

Epoch 10

Columns held in Block RAM (internal memory)	8,9,2,3,4,5,6,7
Columns undergoing PEs	8,9,2,3,4,5,6,7
Columns undergoing Reac	9
Columns undergoing Update (write back)	8,3,4,5,6,7
Columns undergoing Write back	2

And so on …

Figure 36 : Scheduling of the computation in the Jacobi solver

The memory columns are implemented in dual port block RAM, with the read address and write addresses being incremented in each clock cycle. The main design challenge is to arrange the scheduling of memory accesses for each column, and for the exchange of data between on-chip memory and external memory. It can be seen from Figure 36 that one of the columns of memory is being loaded with new data from external memory (SRAM bank0), one is uploading its results to external memory (SRAM bank1), and the remaining six columns are involved in computation. After all the columns in SRAM bank0 have been read and the new data have been written into SRAM bank1, the process is started again with data now downloading from SRAM bank1 and uploading to SRAM bank0. This process repeats until the convergence is achieved. With appropriate design, 8 copies of the data path shown in Figure 37 can operate in parallel, producing eight results per clock cycle.

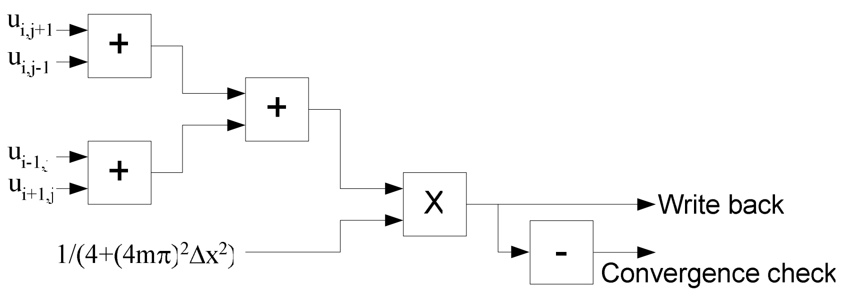


Figure 37 : Processing element for row update in Floating-point Jacobi solver

The datapath is shown in Figure 37. The floating-point adder and multiplier intellectual property (IP) cores are generated by the design software for Xilinx FPGAs in order to trade off latency with maximum clock frequency. The floating-point arithmetic pipeline operates at a 80 MHz clock rate (40 MHz PCI). It requires 28 clock cycles of total latency from the reading of one element's displacements to the write back of the new displacements. As a result, $5 \times 8 \times 80M = 3.2$ billion operations per second can be carried out per second using a memory bandwidth of 8×4 byte$\times$80MHz = 2.56 GByte/s.

5.3.1.3 Floating-Point Hardware Implementation using Red-Black successive over-relaxation

Based on the discussion in subsection 4.4.1.1, the Jacobi method, although simple and very suitable for hardware implementation, is well known to have poor convergence properties. On the other hand, the Gauss-Seidel and successive over-relaxation (SOR) methods converge faster than the Jacobi method, but these methods are more problematic to implement due to the data dependences introduced, which means operations cannot be fully pipelined. The computation must therefore stall until the required results have been computed.

This problem can be removed by the use of the red-black successive over-relaxation iteration scheme. The two dimensional finite difference grids are coloured with a red and black checkerboard, and then 16 columns of on-chip Block RAMs are used as a cache to hold up to 8 columns of values: the red values are stored into the first 8 block RAMs, whereas and the black values are stored into the remaining 8 block RAMs. Figure 38 shows one example of our on-chip memory architecture. Columns B_0 to B_7 hold the black grid values of columns 0 to 7 of the domain, and columns R_0 to R_7 hold the red grid values of columns 0 to 7 of the domain. The processing proceeds in a manner similar to the 8 columns Jacobi scheme, where the following columns of the domain are continuously fed into the internal memory columns following the red-black order. When the red columns compute their updated values, using the previously computed values of black columns, the black columns only exchange the data to and from external memory. Conversely, when the black columns are undergoing update using the newly calculated values of the red columns, the red columns are exchanging the data with external memory. As a result of using separated on-chip memory columns to store the red and black grid columns, each column will undergo 14 iterations between download and upload. This process continues until all columns have been completed.

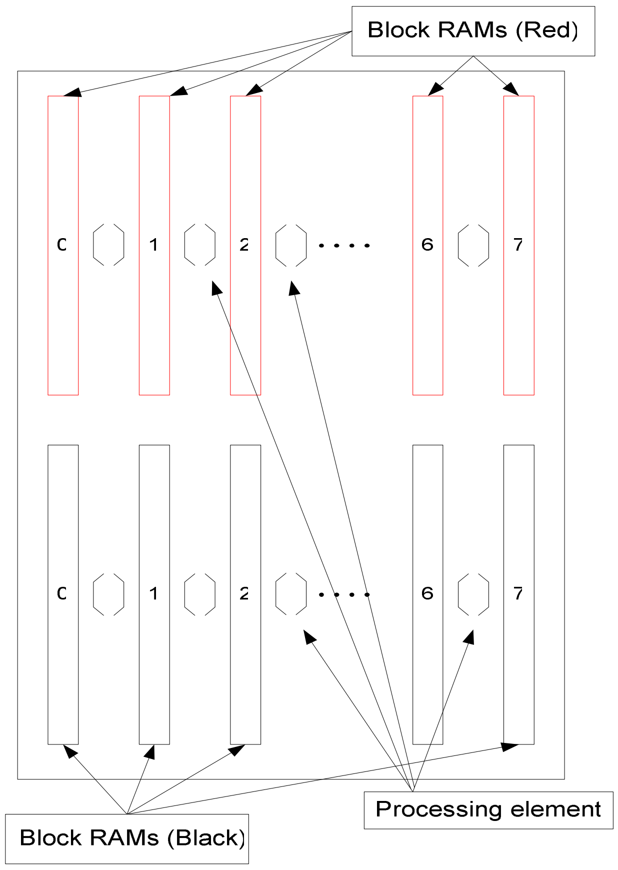
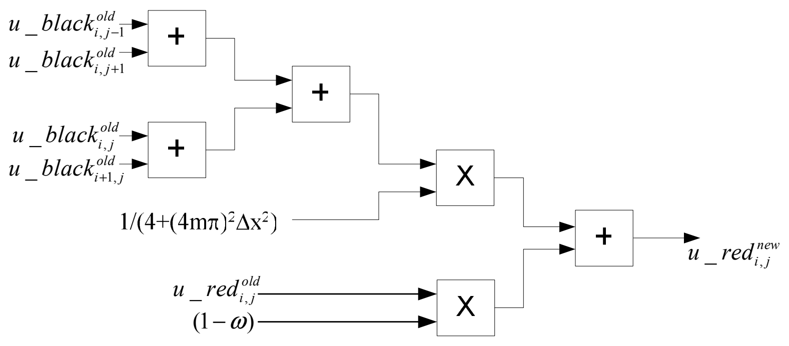
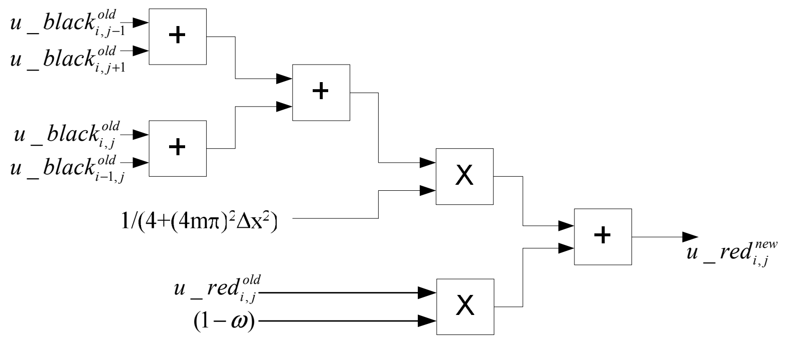


Figure 38 : Architecture of data path of red-black successive over-relaxation scheme

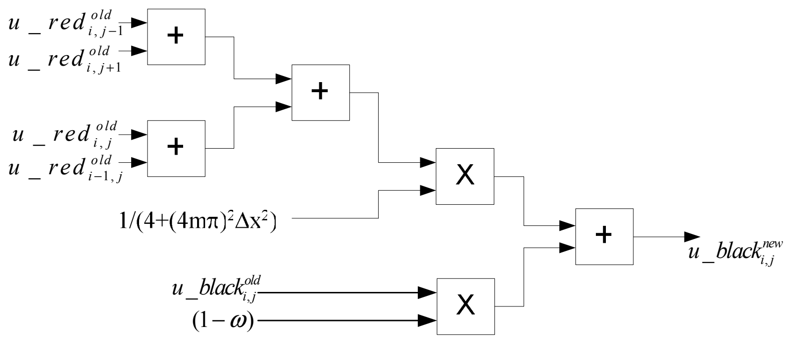
Figure 39 shows 4 processing elements representing the data path between columns. In this example, 16 copies of the data path can operate in parallel, producing one result each per clock cycle. The problem of data dependence is removed by the use of the modified hardware design.



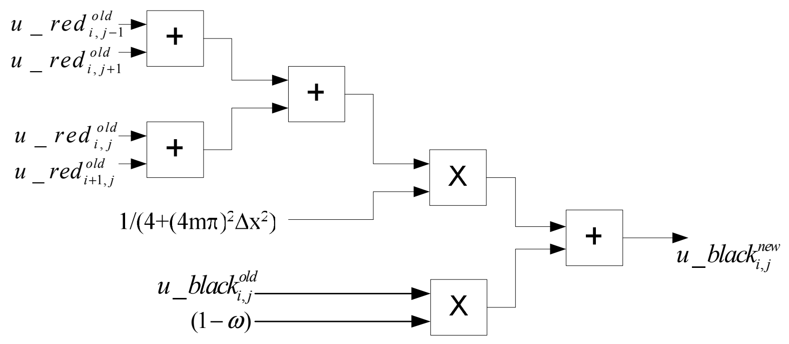
(a)



(b)



(c)



(d)

Figure 39 : Processing elements: (a) for the old red columns, (b) for the even red columns, (c) for the old black columns, (d) for the even black columns.

The floating-point arithmetic pipeline operates at a 90 MHz clock rate (45 MHz PCI). As a result, $6 \times 16 \times 90M = 8.64$ billion operations per second can be carried out per second using a memory bandwidth of 16×4 byte$\times$90MHz = 5.76 GByte/s.

5.3.2 Finite Element Method

In this section, one dimensional line finite elements, two dimensional rectangular finite elements and three dimensional tetrahedral finite elements are implemented in a way that provides efficient solutions by tackling the drawbacks of the finite element method: how to optimise the computationally expensive matrix decomposition for direct solution methods and how to optimise the matrix-vector multiplier for iterative solvers.

The hardware simulation uses MathWorks Simulink with Xilinx System Generator, as this tool is a high-level tool and provides interactive graphical model design and simulation for designing high-performance systems using FPGAs. However, the Xilinx Blockset in System Generator only provides a fully parameterized implementation of fixed-point arithmetic, with no floating-point arithmetic at all. However, it is widely accepted that floating-point is generally required for finite element analysis. Thus, simpler one-dimensional finite element method is illustrated using System Generator, and the two-dimensional rectangular finite element method and the full three-dimensional tetrahedral finite element method implementations use floating-point and were coded in VHDL.

5.3.2.1 One-Dimensional FEM

An explicit 1D linear solid element mesh, with n=8, is shown in Figure 40.

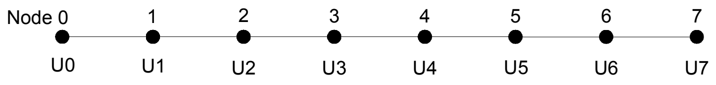


Figure 40 : 1D finite element mesh

As shown in Figure 41, 8 columns of block RAM memory are used to hold the columns of values for $u_i^{t+\Delta t}$, which are stored as 32-bit fixed point. The memory columns are implemented in dual port block RAM, with the read addresses and write addresses being incremented in each clock cycle.

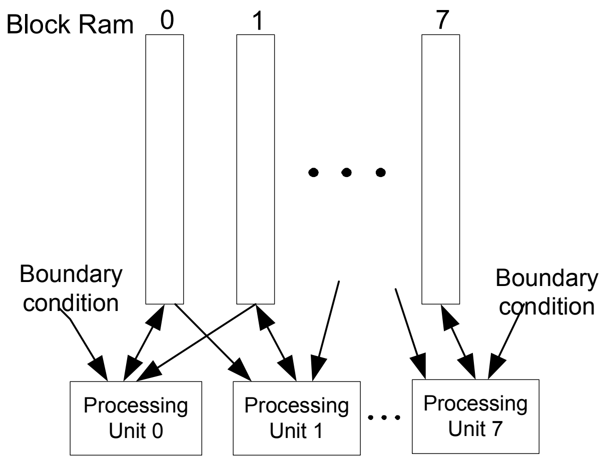


Figure 41 : Architecture of 1D FEM solver

The processing unit used to solve Eq. (83) is shown below in Figure 42.

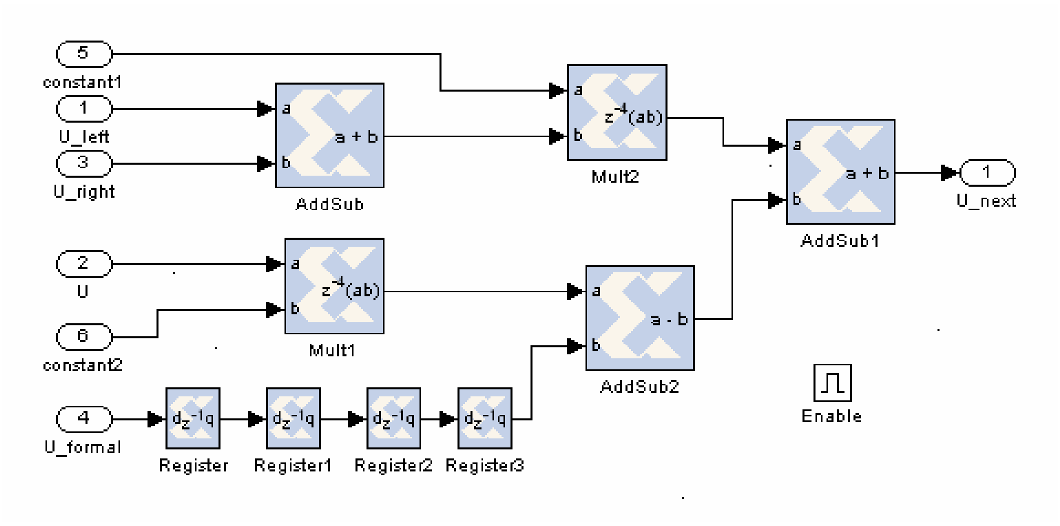


Figure 42 : Basic processing unit of 1D FEM solver in Matlab Simulink

5.3.2.2 Two-Dimensional FEM

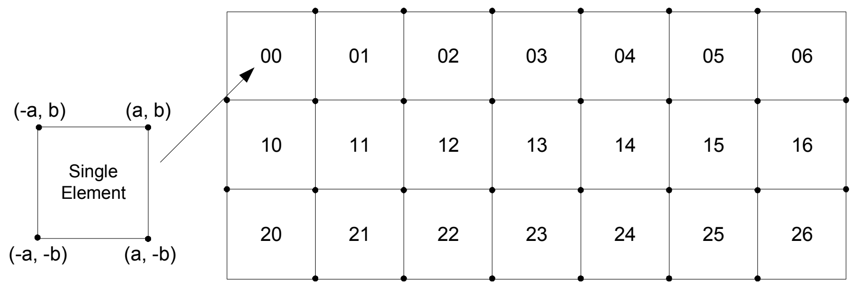


Figure 43 : Two-dimensional rectangular plane strain elements

Figure 43 shows a regular mesh using two-dimensional rectangular plane strain elements. Here there are 21 elements, which produce 64 equations, giving a 64×64 global stiffness matrix and mass matrix respectively. The procedure for solving the 2D FEM of Eq. (98) is very similar to the procedure for solving the 1D FEM, but the hardware design for the solution of the 2D FEM becomes more complex due to the huge memory required. Fortunately, the global matrix can be decomposed into a series of single element matrices and they can be processed independently with the result assembling together at the end, as shown in Figure 44 below.

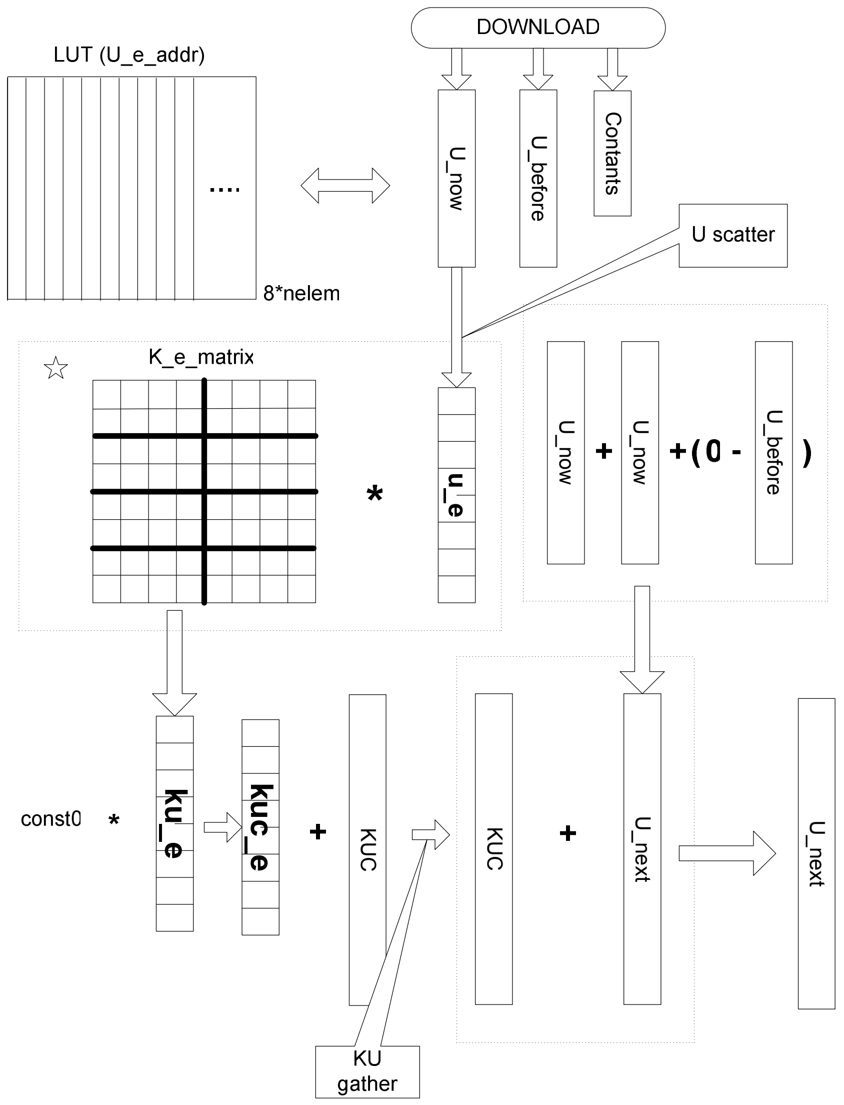


Figure 44 : Data Flow of 2D FEM Design

U_now is scattered by using a Look-up Table, which holds one-element displacements' address with respect to the read address of BlockRam U_now. Then each element is sent to the matrix multiplication block (indicated in Figure 44 with a star in its left-top corner; this block is expanded out in Figure 45). After matrix multiplication, u_e is gathered together as U_next, which contains the displacements for the next time step.

In Figure 45, M indicates a floating point multiplier, A indicates a floating point adder, and R indicates a register. Each element stiffness matrix is stored in 4 dual port block RAMs, thus the read-address is capable of being incremented in each clock cycle. Two clock cycles are needed to finish two rows accumulation, which is equivalent to one row being calculated in every clock cycle. The floating-point arithmetic pipeline operates at a 130 MHz clock rate (65 MHz PCI). It requires 40 clock cycles of total latency from the reading of one element's displacements to the write back of the new displacements. As a result, 8 × 18 × 130M = 18.72 billion operations per second can be carried out per second using a memory bandwidth of 8 ×4 byte×130MHz = 4.16 GByte/s.

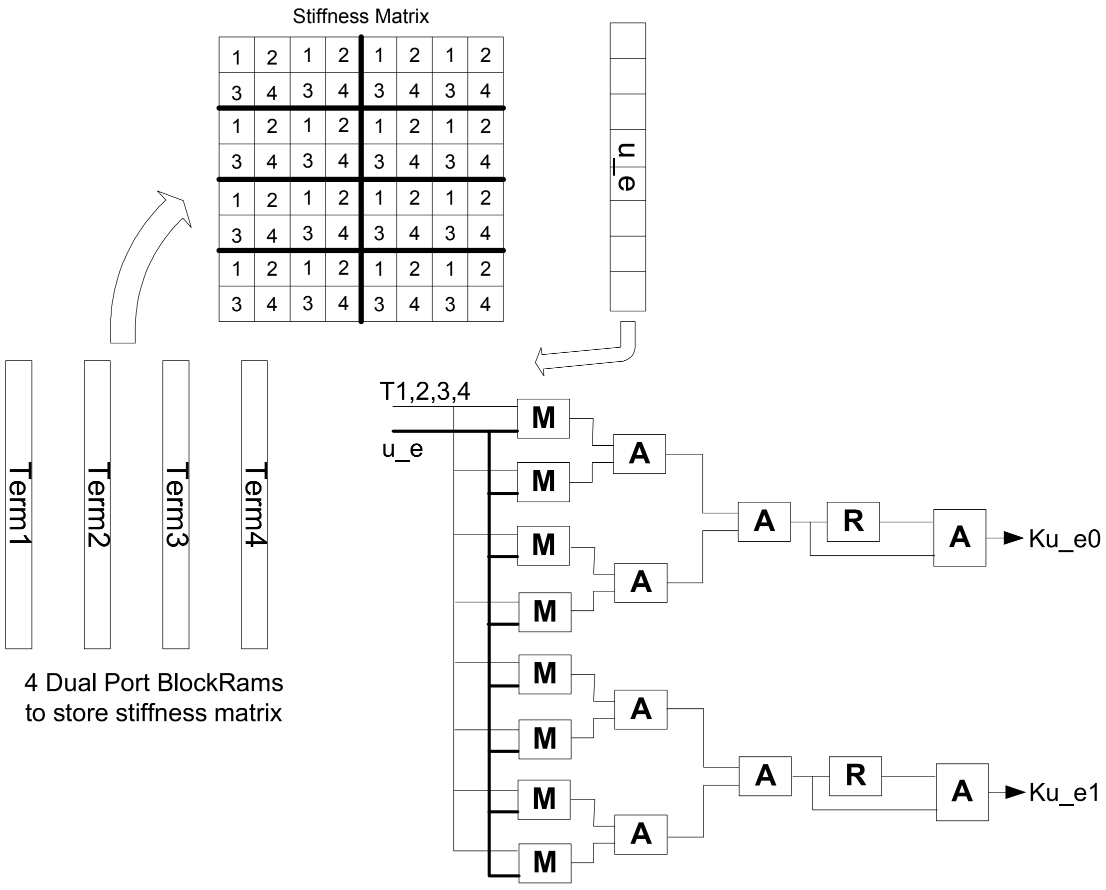


Figure 45 : Stiffness Matrix Multiplication

5.3.2.3 Three-Dimensional FEM

In section 4.2.3, based on the 3-dimensional tetrahedral finite element method, the global stiffness matrix A is built up by the assembling of each element's stiffness matrix K_e. Thus, the dataflow shown in Figure 46 on the left-hand side is used for the normal software solution, which assembles the global stiffness matrix A first, before performing the matrix multiplication. A very different approach is used in hardware as shown on the right-hand side diagram. In order to parallelize the matrix multiplication, the vector p_k is scattered element by element, and multiplied by each element's K_e, and then gather the vectors Kp_{e_i} together in order to get the same result as $\omega_k = Ap_k$.

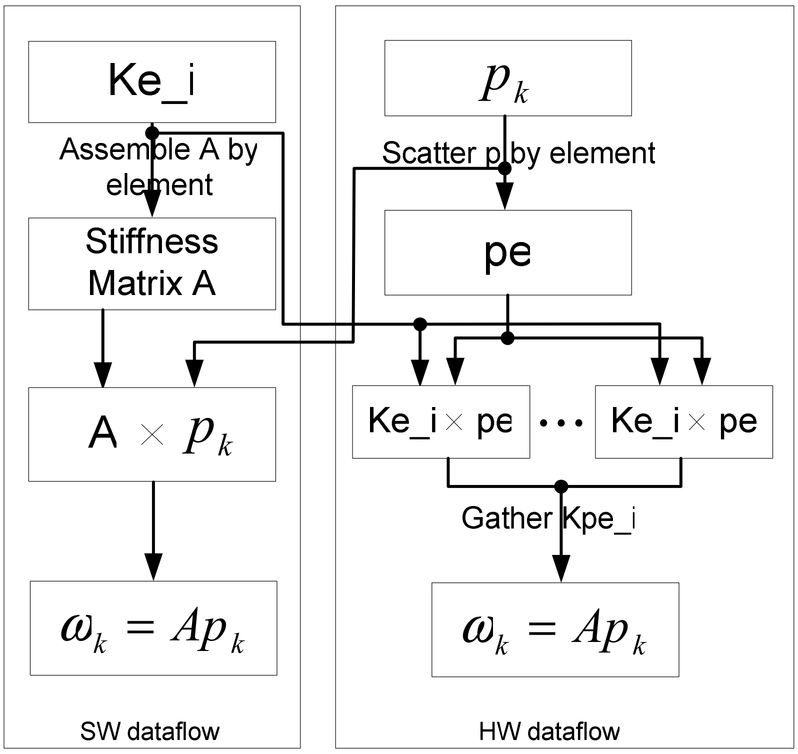


Figure 46 : Matrix Multiplication Parallelization using the element by element method.

K_e is initialized in 12 block RAMs, which hold the row values of each stiffness matrix and the length is 72 in total. By scattering p_k to pe and downloading pe into SRAM in a sequential element-by-element fashion, the matrix multiplication can be processed in parallel. Compared to the software system, whose matrix-vector multiplication requires the number of ($N(2N-1)$) operations, the iteration of the hardware matrix-vector multiplication only involves the number of ($6N$) operations, where N stands for the number of unknowns. Matrix-vector multiplication is implemented in hardware, shown as below in Figure 47.

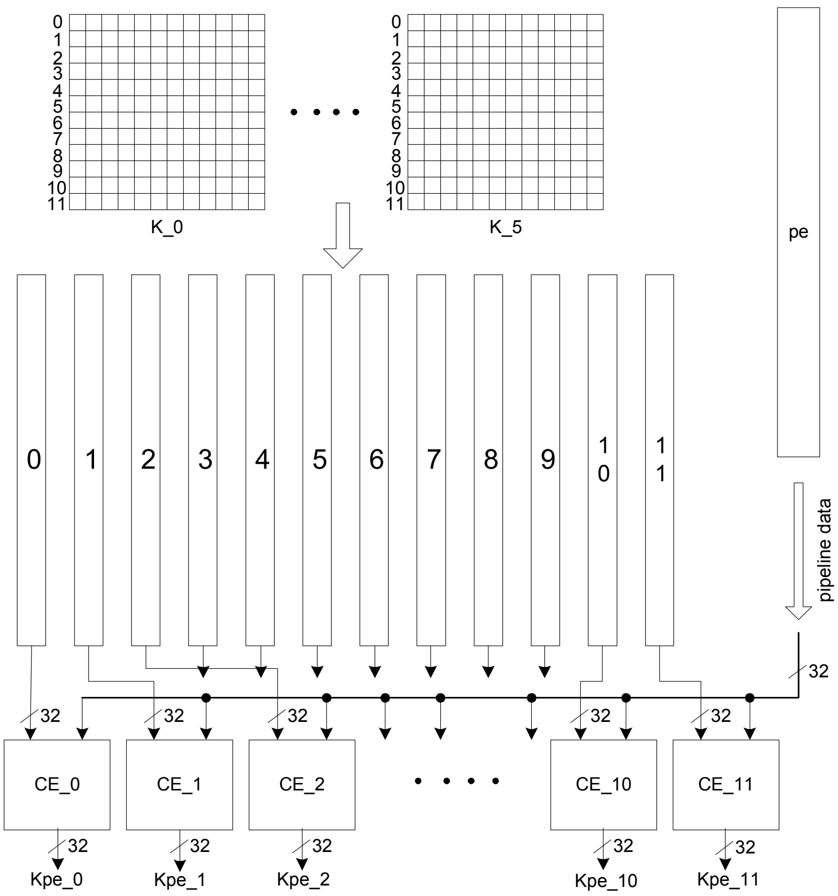
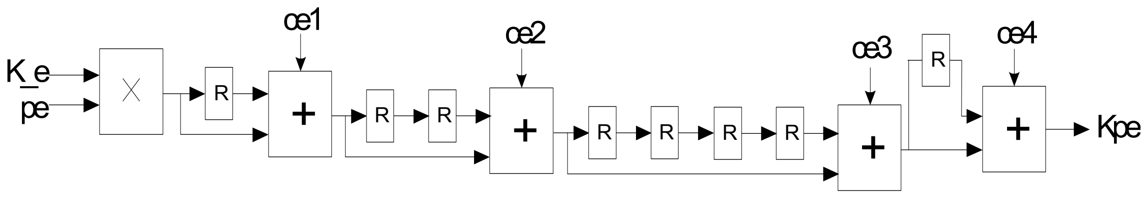


Figure 47 : Architecture of matrix multiplication within FPGA

The calculation element (CE) is shown in Figure 48, where R indicates a register that introduces a one clock cycle delay. The calculation latency can be fully overlapped by the deeply pipelined design, thus a new data item is read and a new result is written back for each clock cycle (multiplication results can be generated every clock cycle).



where ce1 to ce4 are Clock Enable signals: when ‘ce’ is deasserted, the clock is disabled, and the state of the core and its outputs are maintained.

Figure 48 : Calculation Element (CE) for 3D FEM.

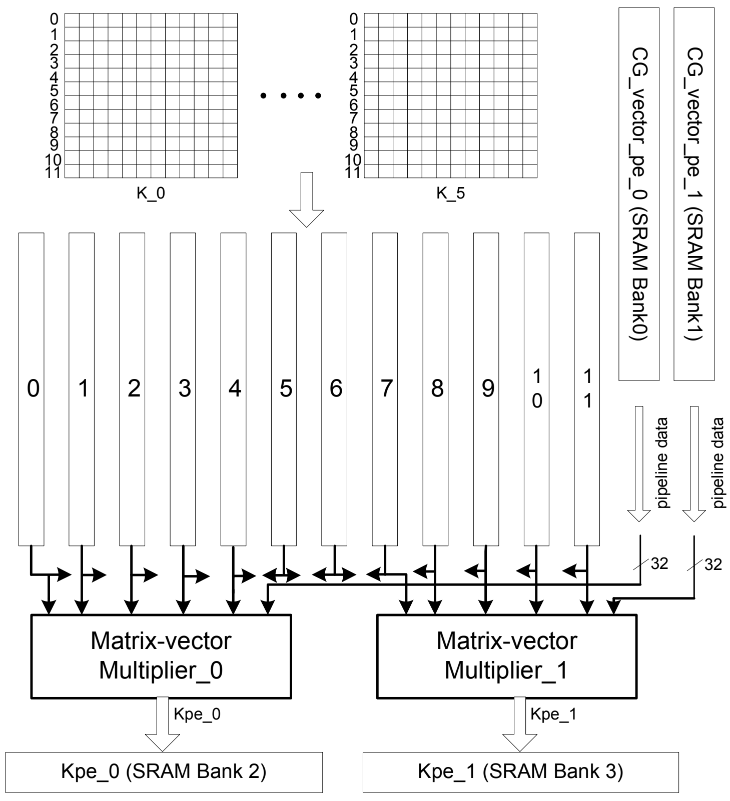

K_0
K_5
CG_vector_pe_0 (SRAM Bank0)
CG_vector_pe_1 (SRAM Bank1)
pipeline data
pipeline data
32
32
Matrix-vector Multiplier_0
Matrix-vector Multiplier_1
Kpe_0
Kpe_1
Kpe_0 (SRAM Bank 2)
Kpe_1 (SRAM Bank 3)

Figure 49 : Architecture of parallel matrix multiplications within FPGA

One implementation with two matrix-vector multipliers is shown in Figure 49. This was implemented in the Xilinx 4VLX160 FPGA on a RC2000 PCI bus plug-in card. The element stiffness matrices K_{e_i} are first downloaded into 12 dual-port block RAMs which hold the row values of each stiffness matrix and the length is 72 32-bit words each. The sub-domain conjugate vector pe_0 and pe_1 are stored in SRAM Bank0 and SRAM Bank1 respectively, and then fed into Matrix-vector Multiplier block0 and block1 in each clock cycle. Finally, the results are written into SRAM Bank2 and Bank3 respectively as Kpe_0 and Kpe_1 in each clock cycle. The implementation of two matrix-vector multipliers allows the maximum utilization of hardware resources and exploits the scalability of parallelizing the element-by-element FEM.

5.4 Summary

A brief introduction to the reconfigurable computing platform was given in this chapter. The hardware implementations of the finite difference method and the finite element method were described.

The first reconfigurable computing approach to the finite difference method made use of 32-bit customised fixed-point arithmetic, so this enabled one entire 2D subsection to fit in a single FPGA. Also Jacobi, Gauss-Seidel and Successive Over-Relaxation iteration methods were evaluated. Based on the same concepts, a floating-point Jacobi solver for 3D finite difference analysis was presented. Floating-point arithmetic was introduced as it is required for a wide range of numerical analysis problems. Nevertheless, as FPGAs, are increasing rapidly in logic capacity and speed, the loss of speed-up associated with floating-point arithmetic is likely to be offset in future. A more complex implementation of the finite difference method, which made use of red-black successive over-relaxation scheme, was also described.

The hardware implementations of finite element analysis were based on the element-by-element scheme, which removes the limitation of memory requirements and minimizes the communication overheads compared to traditional solution approaches.

The performance and results of the hardware implementations will be discussed in chapter 6, as well as the evaluation of the hardware parallelism achieved.

Chapter 6

Hardware and Software Comparison

6.1 Introduction

This chapter evaluates the performance of the hardware implementations for the Finite Difference Method and the Finite Element Method in terms of numerical precision, speed-up, and cost compared with the software implementations.

6.2 Numerical Precision

Scientific computing, such as computational mechanics, involves a set of computing tasks traditionally solved using uniprocessors or parallel computers. Such large scale simulations are normally characterized by large systems of partial differential equations, which often involve large regular or adaptive grid structures. The conventional methods require operations that typically employ double-precision floating-point computations. On the other hand, FPGAs were originally used only for small-scale glue logic applications. In recent years, the performance of floating point units in FPGAs has increased significantly, as built-in hardware multipliers have been incorporated, so that floating point operations can be performed at rates up to 230MHz. Therefore the current research was extended to use not only fixed-point arithmetic but also floating-point arithmetic within the reconfigurable hardware accelerators; however, loss of accuracy will still occur due to the limits of the number of bits used to represent the numbers. In considering the accuracy of the solutions produced, the main concepts used in numerical analysis [103] are:

Precision: Precision is the maximum number of non-zero bits representable.

Resolution: Resolution is the smallest non-zero magnitude representable.

Absolute Error Δ***:*** the absolute error is the distance between the number x and the estimate x'.

$$\Delta = |x - x'|.$$

Relative Error : the relative error measures the error relative to the size of the number itself.

$$\delta = \frac{\Delta}{|x|}.$$

Because the size (precision) of the number is determinate, in the following sections, the difference of numerical precision between the software and hardware implementations will be measured and analysed based on absolute error.

6.3 Speed-up

Speed-up is a very common criterion to evaluate the performance of a parallel system. As shown in Eq. (119), speed-up measures how much faster a computation finishes on a parallel computing system than on a uni-processor machine.

$$Speed\_up(n) = \frac{T^*(n)}{T_p(n)} \qquad \textit{Eq. (119)}$$

where n represents problem size.

p is the number of processors.

$T^*(n)$ is the optimal serial time to solve the computation.

$T_p(n)$ is the runtime of the parallel algorithm.

$Speed\_up(n)$ describes the speed advantage of the parallel algorithm compared to the best possible serial algorithm.

In the following sections, the speed-up is evaluated using the best software algorithm presented in chapter 4 and the hardware implementations presented in chapter 5.

6.4 Resource Utilization

Due to the limitations of hardware resources, resource utilization for the various implementations is considered. The Xilinx Virtex 2V6000 and Virtex 4VLX160 used in this research would have been typical of the state of the art several years ago, but newer, bigger and faster FPGA families are launched almost every year. Thus, a discussion is presented on how these designs would scale to larger and faster FPGAs, and also how the designs would scale to the use of 64 bit floating-point arithmetic. This is based on an extrapolation of the resource analysis of the hardware implementations presented in this thesis. FPGA resource utilization is the measure of spatial allocation of the functional units, such as on-chip memory, DSPs, and so on.

Table 8 compares the logic, memory and arithmetic capacity of the FPGAs used in this study.

Table 8 : The logic, arithmetic and memory capacity of the two FPGAs used in this research [95].

Family	FPGA	CLK(MHz)	Block RAM Blocks [1]		Block multipliers/DSP slices [2]	Logic slices [3]	DCMs [4]
			18 Kb	Max(Kb)			
Virtex II	**2V6000**	**400**	**144**	**2,592**	**144**	**33,792**	**12**
Virtex 4	**4VLX160**	**500**	**288**	**5,184**	**96**	**152,064**	**12**

[1] Block SelectRAM memory modules provide 18 Kb storage elements of dual-port RAM.

[2] Each Virtex-4 DSP slice contains one 18×18 multiplier, an adder and an accumulator, whereas the Virtex II uses a dedicated 18×18 multiplier block.

[3] A logic slice is equivalent to about 200 2-input logic gates. Each slice contains two LUTs and two flip-flops.

[4] DCM (Digital Clock Manager) blocks provide self-calibrating, fully digital solutions for clock distribution delay compensation, clock multiplication and division, coarse- and fine-grained clock phase shifting.

6.5 Finite Difference Method

Due to limitations of PC memory, only domains with 32×32×32 to 256×256×256 were generated, simulated and analysed. The performance of the software version was measured and compared with the results of the hardware version. The double/single precision floating-point software version was run on the same PC (2.4 GHz Pentium 4 PC with 1GByte RAM), and the code compiled in both debug mode and release mode. Debug mode is essential during development, but the downside is that it is significantly slower than its release-mode counterpart.

6.5.1 Numerical Precision Analysis

In this section, numerical precision is compared between hardware implementations and software implementations. Without using any approximations, exact analytical solutions to PDEs play a significant role in the proper understanding of qualitative features of many phenomena in various areas of natural science. Exact solutions can be used as test problems to verify the consistency and estimate errors of various numerical, asymptotic, and approximate analytical methods, but not every exact analytical solution for PDEs can be found easily. So far it has been supposed that the 64-bit floating-point software implementations using exact solutions can be considered as a benchmark versus the hardware results. However, in order to establish whether the results are affected by the precision applied, the 32-bit floating-point software was also simulated to compare against the double precision results. The absolute errors for each simulation are given in Table 9.

Table 9 : Absolute error in the hardware and software 3D FDM implementations compared to the double precision exact analytic solution.

N	SW_exact single_floating_point	SW_FDM double_floating_point	HW customised fixed_point	HW single_floating_point
32	3.234259e-007	0.001459	0.001462	0.001459
64	3.234259e-007	0.000369	0.000474	0.000358
128	4.846198e-007	0.000092	NA*	0.000181
256	5.123316e-007	0.000023	NA*	0.000168

* For hardware implementations using customised fixed point arithmetic, only 32 and 64 column design can be fitted on the Virtex 2V6000 FPGA.

As shown in Table 9, the software exact solution using single precision floating-point arithmetic was found to be almost identical to the software simulation using double precision floating-point arithmetic. The absolute errors for the FDM software simulation using double floating-point precision become smaller and smaller as the number of grid points is increased, and they are nearly identical to the errors from the hardware simulation. Therefore, the lower precision arithmetic used in the hardware implementations can be assumed to have safely satisfied the numerical requirements.

6.5.2 Speed-up

6.5.2.1 Hardware Fixed-point arithmetic vs Software (debug/release mode)

A 3-D FDM simulation using Fourier decomposition was carried out using the fixed-point hardware implementation, as described in section 4.3.1.1. The performance of the hardware version was compared with the software version in debug and release mode respectively. Table 10 shows the simulation time (in seconds) for different cube sizes (N=32, 64) each of the hardware fixed-point simulations and the software simulations compiled in debug mode. In order to better assess the software performance, the software simulations were re-run in release mode, and the results are shown in Table 11. For both of the two implementations, cube 32×32×32 and cube 64×64×64, T(SW) and T(SW_GS) indicate the simulation time of the software implementations using exact solution and Gauss-Seidel solution with Fourier Decomposition respectively. The hardware version uses Jacobi solution with an entire N×N domain fitted onto a single FPGA, giving an operation throughput up to 19.2 billion per second working at a clock speed of 60MHz.

Table 10 : Simulation time (in seconds) for the software (SW) and the hardware (HW) fixed-point implementations for FDM (Debug mode).

	T(HW)	T(SW_GS)	T(SW)	Speed-up (SW_GS)	Speed-up (SW)
32	0.001787	0.177349	0.015442	99.2	8.6
64	0.004700	2.065627	0.117739	439.5	25.1

Table 11 : Simulation time (in seconds) for the software (SW) and the hardware (HW) fixed-point implementations for FDM (Release mode).

	T(HW)	T(SW_GS)	T(SW)	Speed-up (SW_GS)	Speed-up (SW)
32	0.001787	0.037969	0.007454	21.2	4.2
64	0.004700	0.495538	0.056225	105.4	12.0

The results suggest that the Jacobi hardware fixed-point solution on the Virtex 2V6000 FPGA can outperform a 2.4 GHz Pentium4 PC with 1GByte RAM by a factor of approximate 100, using a full strength optimizing compiler.

6.5.2.2 32 bit Floating-point Jacobi Hardware vs Software (debug/release mode)

In this section, the performance of the single floating-point precision hardware implementations using the Jacobi and Red-black successive over-relaxation solutions is measured and compared with the software versions.

As described in section 4.3.1.2, the 32 bit floating-point hardware implementations of the FDM using the Jacobi solution were simulated and compared with the performance of the software version using the Gauss-Seidel solution. Table 12 and Table 13 show a comparison between the speed-up achieved by the hardware simulations as compared to the software running on a 2.4 GHz Pentium 4 PC. There are four architectures of hardware implementations, with 4/8/16/24 columns of on-chip memory used within the FPGA. The FDM running in hardware is found to be faster than in software. Speed-ups of a factor of approximately 32 can readily be obtained when 24 are columns used, and a minimum speed-up of 4.6 times can be achieved using the 4 columns design. Given that the FPGA runs at a clock speed far lower than the Pentium 4 PC microprocessor, it can be seen that the hardware implementations make very good use of the intrinsic parallelism of the algorithms.

Table 12 : Simulation time (in seconds) for the software and the 32bit floating-point hardware implementations for FDM using Jacobi iteration (Debug mode).

N	T(HW_Jacobi)				T(SW_GS)	Speed-up			
	4	8	16	24		4	8	16	24
32	0.014021	0.012975	0.012451	0.011963	0.177349	12.6	13.7	14.2	14.8
64	0.119067	0.106382	0.100040	0.096926	2.065627	17.3	19.4	20.6	21.3
128	0.883373	0.727367	0.649363	0.600363	24.11915	27.3	33.2	37.1	40.2
256	9.348993	7.283259	6.100391	5.858901	292.6948	31.3	40.2	48.0	50.0

Table 13 : Simulation time (in seconds) for the software and the 32bit floating-point hardware implementations for FDM using Jacobi iteration (Release mode).

N	T(HW_Jacobi)				T(SW_GS)	Speed-up			
	4	8	16	24		4	8	16	24
32	0.008175	0.007128	0.006900	0.006431	0.037969	4.6	5.3	5.5	5.9
64	0.061290	0.048596	0.042253	0.040139	0.495538	8.1	10.2	11.7	12.3
128	0.508869	0.352863	0.278529	0.248858	6.258212	12.3	17.7	22.5	25.1
256	5.599639	3.675338	2.728999	2.392471	77.20391	13.8	21.0	28.3	32.3

Figure 50 shows graphically the CPU simulation times required for the various implementations. The simulation times are plotted against the number of grid points in the cube on a log-log scale for problems ranging in size from 32^3 grid points to 256^3. The computing time for the software implementation grows linearly with the cube size, and the hardware implementations increase almost linearly. Thus, the speed-up achieved by the hardware does not saturate as the cube size become large.

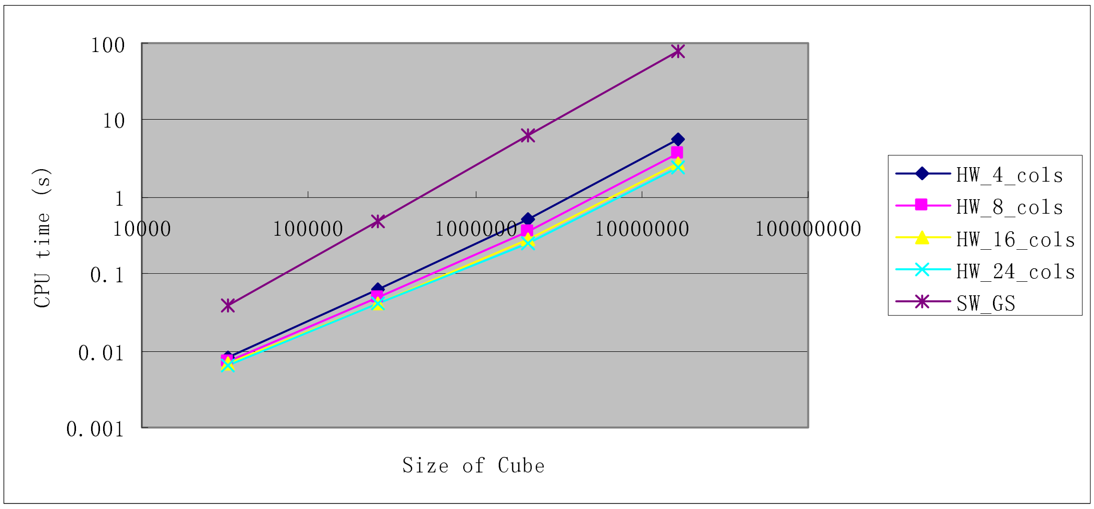


Figure 50 : 32 bit Floating Point Jacobi Implementation
(Software compiled in release mode).

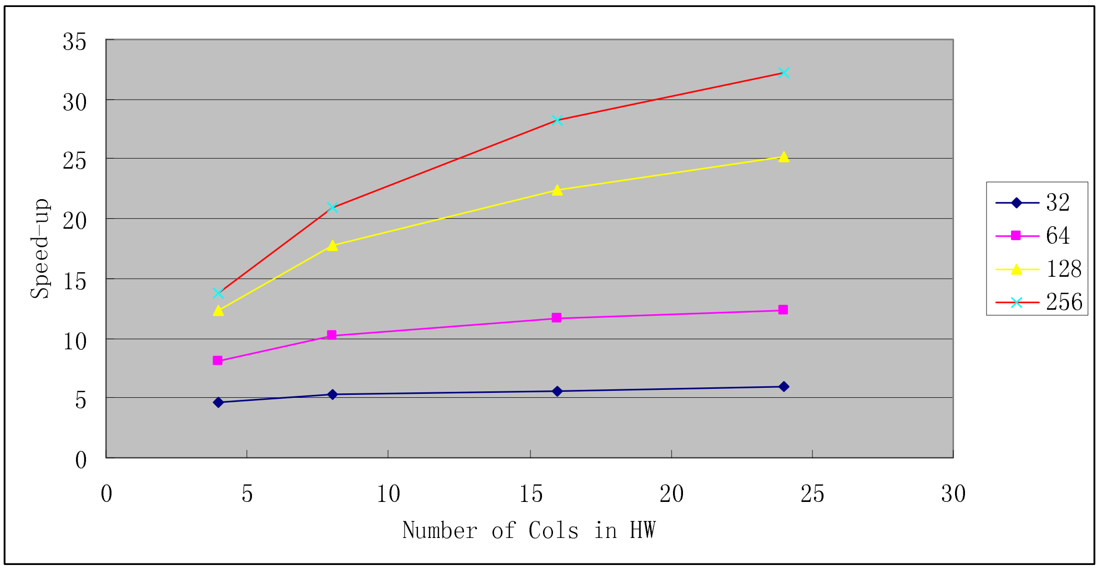


Figure 51a : 32bit Floating Point Jacobi Hardware Implementation Speed-up for grid points from 32^3 to 256^3 (Software compiled in release mode).

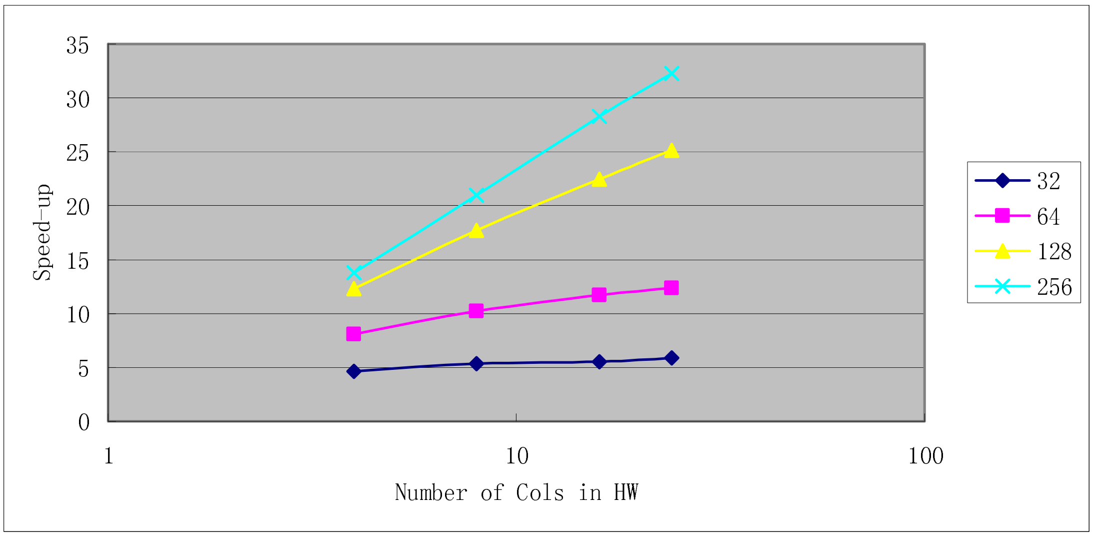


Figure 51b : 32 bit Floating Point Jacobi Hardware Implementation Speed-up for grid points from 32^3 to 256^3 (Software compiled in release mode) shown on a log-lin scale.

Figure 51 shows graphically the achieved speed-up using a full strength optimizing compiler (Figure 51a uses a lin-lin scale and Figure 51b shows the same data on a log-lin scale). The speed-up grows almost linearly with the increase in the level of hardware parallelism. In comparison with the processing time, which depends on the problem size, the number of iterations for convergence, and the level of parallelism in the hardware implementations, the time taken to transfer the data onto and off of the hardware board can be ignored. As shown in Table 14, the data transfer time from Host-to-SRAM and SRAM-to-Host is far smaller than the processing time when the matrix size increases.

Table 14 : Transfer duration vs. processing time using the 8 column design (in milliseconds)

Duration (in milliseconds) / Matrix Size	Transfer IN/OUT		Processing
	Host→SRAM	SRAM→Host	
32	0.091902	1.48631	0.763
64	0.363960	3.14218	9.22903
128	1.51749	3.27972	113.466
256	4.51255	4.37582	1399.51

6.5.2.3 32bit Floating-point Red-black Successive Over-Relaxation Hardware vs Software (debug/release mode)

Due to the poor convergence property of the Jacobi iteration method and the data dependency property of Gauss-Seidel/Successive Over-Relaxation iteration methods, the red-black successive over-relaxation solution was implemented. The performance of hardware implementations using the red-black successive over-relaxation scheme, described in section 5.3.1.3, was compared to double precision and single precision software implementations running on a 2.4 GHz Pentium 4 with 1 GByte of memory. As shown in the discussion in section 6.1, the results generated using single precision are almost identical with the results using double precision, so it can be concluded that the solutions obtained from both methods are almost equivalent.

The computing times for the double precision floating-point software simulation are shown in Table 15 and Table 16, compiled in debug mode and release mode respectively. The hardware implementation can achieve a speed-up of 38 compared to the 64-bit floating-point Gauss-Seidel software solution for a cube of dimensions 256×256×256 using a full strength optimizing compiler. The performance of the hardware solution is greater for larger systems, as the balance between data transfer and computation is improved. As the hardware implementation uses single precision floating-point arithmetic, the software simulator was modified to also use single precision to provide a fair basis for comparison. Table 17 and Table 18 show the performance comparison between hardware and single precision floating-point software solutions. The speed-up compared to the single precision floating-point software solutions is reduced (as the Pentium 4 processor on which the software simulations run is a 32 bit processor, so there is a time penalty for using double precision). The speed-up achieved was a factor of approximately 9.

As there are 16 copies of the data path implemented in parallel onto a single chip, the hardware architecture can be considered to be almost equivalent to the Jacobi 16 column hardware implementations. The performance of the Red-black successive over-relaxation solution is around 35% better than Jacobi solutions.

Table 15 : Simulation time (in seconds) for the 64bit floating-point software and the 32bit floating-point hardware implementation for FDM using red-black successive over-relaxation iteration (Debug mode).

N	T(HW_RB_SOR)	T(SW_double_floating_point)				Speed-up			
		Exact	GS	SOR	RB_SOR	Exact	GS	SOR	RB_SOR
32	0.012112	0.015442	0.177349	0.078719	0.078562	1.2	14.6	6.5	6.5
64	0.095743	0.117739	2.065627	0.885003	0.858751	1.2	21.5	9.2	9.0
128	0.595941	0.919706	24.11915	9.285473	9.284257	1.5	40.5	15.6	15.6
256	5.488757	7.309252	292.6948	201.5683	212.5884	1.3	53.3	36.7	38.7

Table 16: Simulation time (in seconds) for the 64bit floating-point software and the 32bit floating-point hardware implementation for FDM using red-black successive over-relaxation iteration (Release mode).

N	T(HW_RB_SOR)	T(SW_double_floating_point)				Speed-up			
		Exact	GS	SOR	RB_SOR	Exact	GS	SOR	RB_SOR
32	0.006267	0.007454	0.037969	0.013575	0.01098	1.2	6.1	2.2	1.8
64	0.038956	0.056225	0.495538	0.175817	0.147751	1.4	12.7	4.5	3.8
128	0.221447	0.422142	6.258212	2.213882	1.842309	1.9	28.3	10.0	8.3
256	2.022227	3.334128	77.20391	25.47644	23.38476	1.6	38.2	12.6	11.6

Table 17 : Simulation time (in seconds) for the 32bit floating-point software and the 32bit floating-point hardware implementation for FDM using red-black successive over-relaxation iteration (Debug mode).

N	T(HW_RB_SOR)	T(SW_single_floating_point)				Speed-up			
		Exact	GS	SOR	RB_SOR	Exact	GS	SOR	RB_SOR
32	0.012112	0.015543	0.041901	0.021365	0.017514	1.3	3.5	1.8	1.4
64	0.095743	0.120727	0.428179	0.18491	0.16526	1.3	4.5	1.9	1.7
128	0.595941	0.934129	4.398881	1.660374	1.752469	1.6	7.4	2.8	2.9
256	5.488757	7.440643	75.37364	35.3634	46.80425	1.3	13.7	6.4	8.5

Table 18 : Simulation time (in seconds) for the 32bit floating-point software and the 32bit floating-point hardware implementation for FDM using red-black successive over-relaxation iteration (Release mode).

N	T(HW_RB_SOR)	T(SW_single_floating_point)				Speed-up			
		Exact	GS	SOR	RB_SOR	Exact	GS	SOR	RB_SOR
32	0.006267	0.007467	0.014542	0.007118	0.007087	1.2	2.3	1.1	1.1
64	0.038956	0.055767	0.155769	0.068419	0.076462	1.4	4.0	1.8	2.0
128	0.221447	0.418622	1.568907	0.63955	0.734665	1.9	7.1	2.9	3.3
256	2.022227	3.3196	18.88989	10.15688	11.65903	1.6	9.3	5.0	5.8

The hardware design of the Red-Black successive over-relaxation method was also implemented on a Virtex 4 LX160 FPGA. The performance of the hardware designs on the Virtex 2V6000 and 4V LX160 FPGAs is compared in Table 19.

Table 19 : Simulation time (in seconds) for the 32bit floating-point Hardware implementation on different FPGA boards.

N	V2(110MHz)	V4(180MHz)	Speed-up
32	0.006267	0.002827	2.216
64	0.038956	0.016292	2.391
128	0.221447	0.0813515	2.722

The hardware design on the 4VLX160 FPGA ran twice as fast as the same design on the 2V6000 FPGA. However, the clock speed is not twice as fast. The remaining difference in simulation time is due differences in the PCs in which the FPGA boards were used, which results in a different speed of execution for the software versions that are used as the baseline in the two cases.

6.5.3 Resource Utilization

The resource utilizations for the various implementations of the FDM are shown in Table 20. The use of fixed-point arithmetic in the Jacobi implementation, running at a clock speed of 60 MHz, reduces the complexity of the computational pipelines within the hardware implementation, thereby allowing a greater level of parallelisms to be obtained. In order to fit a 64×64 subdomain onto a single chip, 64 columns of block RAM memory are used to store the 32bit fixed-point values. By contrast, the floating-point designs, which cannot accommodate the entire 2D subdomain onto a single chip, fit an appropriate number of data paths instead of the full number of data paths. The on-chip resources were almost exhausted for the 24 columns design at a 70 MHz clock rate, compared to the simplest 4 columns design running at a speed of 100 MHz. Each processing element, shown in Figure 37, uses 3 floating-point adders and 1 floating-point multiplier. In order to trade off latency with maximum clock frequency, the floating-point adder and multiplier are generated using intellectual property (IP) cores with maximum latency by Xilinx CORE Generator. On the Virtex 2V6000 FPGA, only logic implementation is possible for 32-bit floating-point adder, whereas 4 MULT18×18s can be used to assemble a 32-bit floating-point multiplier. Therefore, the total number of Mult18×18s used in the hardware designs is the number of columns implemented in hardware multiplied by 4. The processing element for the Red-black successive over-relaxation solutions includes 3 floating-point adders and 2 floating-point multipliers, as in Figure 39. There are 8 processing elements implemented in the 32-bit floating-point red-black successive over-relaxation design. Thus, the red-black successive over-relaxation design requires more on-chip resources in order to achieve the full pipelining of arithmetic operations in the same way that the 8 column Jacobi design does.

Table 20 : Hardware Resource Utilization (FDM) on Virtex 2V6000 FPGA.

		CLK (MHz)	Slices	%	FFs	%	LUTs	%	BRAMs	%	Mult18×18	%
Jacobi Fixed-point arithmetic (N=64)		60	14,384	42	18,042	26	12,769	18	64	44	0	0
Jacobi 32bit Floating-point arithmetic	4 cols	100	7,652	22	10,457	15	8,151	12	4	2	16	11
	8 cols	80	13,756	40	18,442	27	14,839	21	8	5	32	22
	16 cols	80	25,955	76	34,402	50	28,178	41	16	11	64	44
	24 cols	70	33,790	99	50,298	74	41,787	61	26	18	96	66
Red-black SOR 32bit Floating-point arithmetic		90	19,371	57	25,747	38	21,699	32	16	11	64	44

6.5.4 Results Analysis

The primary aim of this work is to implement a stable and fast solver for 3-dimensional finite difference solution of Laplace equations. Due to the use of Fourier decomposition, the 3-D problem was split into a series of independent 2-D sub-problems. The hardware designs using the Jacobi and Red-black successive over-relaxation iteration schemes, which were chosen as they are very amenable to hardware solution, were implemented and the performance of each was measured and compared.

The accuracy of the results has been demonstrated for both hardware and software implementations using fixed and floating point precision. The hardware and software simulations are reasonably equivalent.

The use of fixed-point arithmetic reduces the complexity of the computational pipelines within the FPGA, thereby allowing a greater level of parallelism (and thus performance) to be achieved. Speed-ups of a factor of approximately 100 can readily be obtained within a single FPGA. As this study is concerned with the solution of the Laplace equation, the fixed-point range can safely be identified a priori once the required boundary conditions are known. Trials have been carried out to determine that the selected range is greater than that required for equivalence to a single precision floating point.

Unfortunately, fixed-point arithmetic can not handle PDE problems with a wide dynamic range of data and overflow may occur, so single precision floating-point arithmetic designs for the Jacobi method have also been implemented in order to extend the range of problems for which hardware solution is beneficial. However, the floating-point arithmetic operators require much larger amount of logic resource compared to their fixed-point counterparts, which means that fewer computational pipelines can be fitted onto a single FPGA. Because of the lower level of parallelism, the speed-up is only up to 32 times. However, the performance of the hardware implementations was improved by using the red-black successive over-relaxation iteration scheme, which has a much better convergence property than Jacobi. For the same level of parallelism, the Red-black successive over-relaxation method runs almost twice as fast as the Jacobi method.

6.6 Finite Element Method

In order to measure the performance of the finite element hardware design, a 3D domain with 48, 384, 3072, 6000, 24576 and 48000 elements was generated and simulated until the iterative methods converged. The performance of the software version was measured and compared with the results of the hardware version. The double precision floating-point software version was run on the same PC (2.4 GHz Pentuim 4 PC with 1 GByte RAM), and the code complied using a full strength optimizing compiler.

6.6.1 Numerical Precision

Table 21 shows the number of iterations required for both software and hardware. The first column, SW(Global), shows the iteration count for a software solution that assembles and solves the global stiffness matrix including all the zero entries. The second column shows the iteration count for a software solution of the element-by-element approach. The third column, HW_1 shows the iteration count for the hardware implementation with a single matrix-vector multiplier block. The fourth column shows the corresponding data for the hardware implementation with two parallel matrix-vector multipliers blocks. The difference in iteration count between the software and hardware version is due to the fact that the software version used double precision whereas the hardware version used 32-bit single precision. The use of lower precision arithmetic produces the same answer, but at the cost of a larger number of iterations.

Table 21 : Iterations required for convergence.

Elements	SW(Jacobi)	SW(Global)	SW(EBE)	HW_1	HW_2
6	219	4	4	4	4
48	980	18	18	18	18
384	4,004	39	39	39	39
750	6,264	49	49	49	49
3,072	16,018	78	78	79	86
6,000	24,994	95	95	96	116
24,576	100,000	150	150	166	166
48,000	NA*	188	188	207	207

*Due to the memory limitation of the PC, the Jacobi method cannot converge for the problem with 48,000 elements.

6.6.2 Speed-up

Table 22 : Comparison of speedup obtained by hardware implementations

Elements	V2_HW_1(80Mhz)	V4_HW_1(140Mhz)	V4_HW_2(140Mhz)
48	3.02	4.92	5.41
384	7.46	10.36	15.32
3,072	9.84	15.37	28.69
6,000	10.02	15.27	29.39
24,576	10.40	15.94	31.65
48,000	11.66	20.30	40.58

Table 22 shows a speed-up comparison between the hardware and software solutions. If multiple FPGAs are used, the speed up should scale linearly, since the method used gives rise to minimal communication overhead. The timing cost of V2_HW_1 shows the result for the design with a single matrix-vector multiplier block implemented on Virtex 2V6000 FPGA.

V4_HW_1 is the design with one single matrix-vector multiplier block and V4_HW_2 has two such blocks; both are implemented on the Virtex 4VLX160 FPGA.

The single precision floating-point implementation operates at a 140 MHz clock rate, and computation within the pipelines can be fully overlapped. As a result, 24 × 5 × 140M = 16.8 billion single precision floating point operations can be carried out per second using a memory bandwidth of 24 × 4 byte × 140 MHz = 13.44 GByte/s.

.

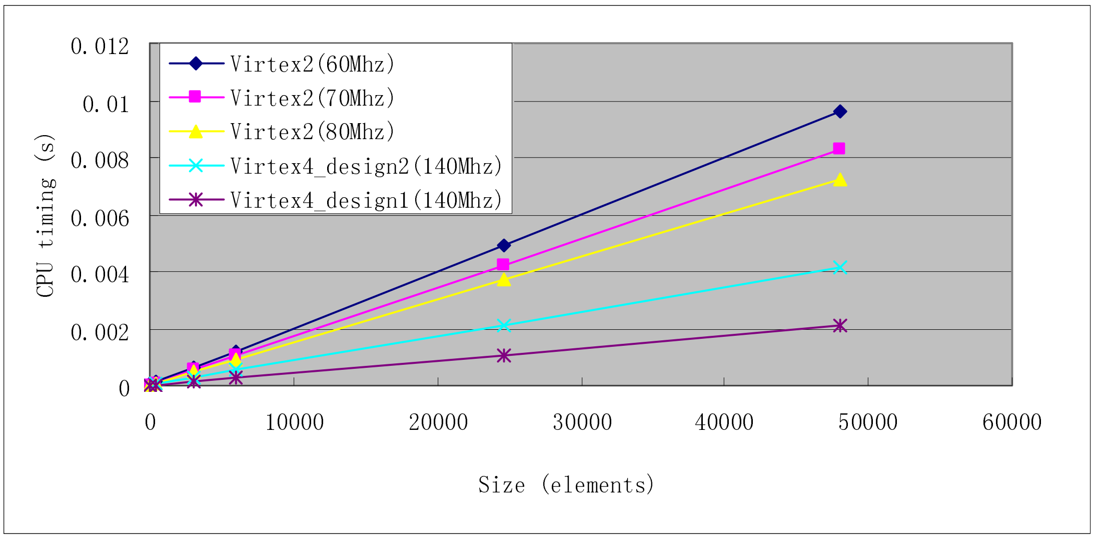


Figure 52a : FEM Hardware Timing Comparison.

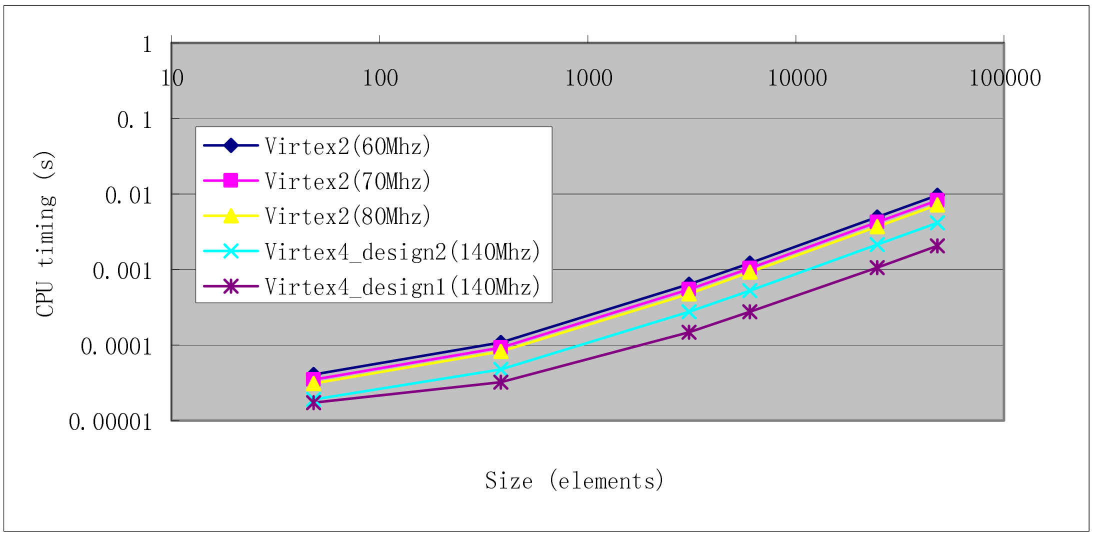


Figure 52b : FEM Hardware Timing Comparison (shown on a log-log scale).

Figure 52 shows the time required for a single Conjugate Gradient iteration using each of the various designs. The speed of the design scales approximately linearly in clock frequency and level of parallelism of the designs for problem sizes exceeding about a few hundred elements. For smaller problem sizes the latencies are not fully hidden, and the speed up is lower. The important feature shown by Figure 52 is that the speed-up does not saturate as the problem size becomes large.

6.6.3 Resource Utilization

Due to the use of the element-by-element scheme, the hardware resources utilization will be constant even when the number of elements in the mesh increases. Table 23 shows the percentage of the FPGA resources used for the various implementations of the FEM.. The Virtex 4VLX160 resources were saturated in Design 2 as the 24 calculation element (CE) units had depleted all DSP48 units. Design 1 has only 12 calculation elements implemented in hardware for both 2V6000 and 4VLX160. The resource utilization of the FPGA based reconfigurable hardware is independent of the matrix size, and the solution therefore scales well to larger problems. As more powerful and higher density FPGAs become available, the number of calculation elements instantiated on the FPGA can be increased, thus giving greater parallelism and greater speed-up.

Table 23 : Hardware resource utilization (FEM).

	FPGA	CLK (MHz)	Slices	%	FFs	%	LUTs	%	BRAMs	%	DSP48s	%	Mult18*18	%
Design1 (12 CEs)	Virtex 2V6000	80	25,610	75	33,557	49	31,332	46	12	8	----	----	48	33
	Virtex 4VLX160	140	25,439	37	32,137	23	31,912	23	12	4	48	50	----	----
Design2 (24 CEs)	Virtex 4VLX160	140	49,068	72	60,459	44	64,881	48	12	4	96	100	----	----

6.7 Summary

In this chapter, the results produced by the hardware and software implementations have been presented and compared in terms of speed-up. The precision of the hardware results has been investigated. The hardware implementations can achieve the same accuracy of results as the double precision floating-point software implementation, with the penalty that a few more iterations are required to reach convergence. Resource utilization has been described in order to analyse the scalability of hardware designs in the following chapter.

Chapter 7

SCALABILITY ANALYSIS

7.1 Introduction

For the design of any large scientific numerical system, architectural scalability is one of the most important evaluation parameters and will affect overall system efficiency, cost and upgradeability significantly.

A number of vendors offer platforms that enable a processor to offload computation to an FPGA-based accelerator, for example Cray [104] and SRC [105]. Normally, once a design has been implemented for one specific FPGA chip and board, the design is locked because of the specific memory architecture. Therefore, the design for the particular FPGA cannot be easily resized or transferred to a more powerful FPGA. This implies that the design can become outdated very quickly.

In the course of implementing numerical solutions in programmable logic, it becomes obvious that while hardware can typically outperform software, the degree of speed-up is strongly dependent on a number of factors, including the size and complexity of the problem, the parallelism of the algorithm used, and the hardware resources available. Several hardware implementations of 3-dimensional problems on one single FPGA board were described in chapter 5, but the speed-up achieved as shown in chapter 6 might not be enough for the simulations in the case of real engineering structures, which involve millions of elements. A high level of parallelism available in the design will offer a significant speed-up. In order to cater for the requirements of these structures, the scalability of our design will be analyzed.

The hardware implementation is dependent upon a number of factors, such as size of resources, peak number of operations, throughput and latency. The main aim of the implementation is to achieve a good balance in terms of optimizing these factors. This chapter will detail the principle potential trade-offs among these factors.

7.2 Scalability of the Finite Difference Method design

7.2.1 Performance of the hardware implementations of FDM

In this part, the performance of the Jacobi FDM system using floating-point arithmetic, which was described in 4.3.1.2, is analyzed.

Due to the area cost of floating-point arithmetic, the FPGA chip cannot accommodate the whole 2-D slice and its associated computational pipelines simultaneously in the way that was implemented in 5.3.1.1. It is therefore necessary to use the FPGA to implement a smaller number of computational pipelines, and to read and to write the data from and to the off-chip memory. There are n columns of on-chip memory within the FPGA as cache that can hold up to n columns of data. These n columns of data can be operated in parallel, producing one result on each clock cycle. It requires 28 clock cycles of total latency from the reading of n columns of data to the write back of the new data to these columns. In addition, computation within the pipeline can be fully overlapped so that the column data is updated on every clock cycle. As a result, using this parallelism, the total number of floating point operators is given in Eq. (120) and the memory usage is given in Eq. (121).

$$\text{Number of Floating Point Operators per clock cycle} = 5nF \qquad \textit{Eq.}\ (120)$$

$$\text{Bandwidth} = 4nF \qquad \textit{Eq.}\ (121)$$

where ***F*** is the system clock rate.

A key aspect of the design is scalability, having a hardware/software co-processor that can take advantage of more logic resources on FPGAs as they become available. To illustrate the scalability of the current systems, several versions of the finite difference designs were implemented with n processing elements. Table 24 gives the performance of 32-bit floating point arithmetic on the 2V6000 FPGA for each number n of processing elements, where n = 4, 8, 16 and 24 respectively. The fourth row in the table gives the performance relative to the 4 processing element design. With more processing elements, the processor can compute more floating-point operations per clock cycle. However, this also results in a decrease in the clock frequency as more resources are used on the FPGAs. The performance almost increases linearly with the number of processing elements, and the operating frequency decreases due to congestion of the hardware implementation. Table 25 gives the performance of 64-bit floating point arithmetic on 2V6000 FPGA for each number of processing elements, where n = 4, 6, 8. Double precision floating-point arithmetic requires more logic than single precision, which means that a smaller number of data paths can fit onto the FPGA. Compared to single precision floating-point arithmetic, the deterioration of system clock frequency for double precision floating-point arithmetic is significant, but the performance of double floating-point arithmetic implementations increases nearly linearly with the number of processing elements.

Table 24 : Performance of 32-bit floating point FDM using Jacobi on Xilinx 2V6000 FPGA.

No. of PEs	4	8	16	24
Clock Freq.	110 MHz	106 MHz	100 MHz	98 MHz
Bandwidth (GB/s)	1.8	3.4	6.4	9.4
GFLOPS (Single FP)	2.2	4.24	8.0	11.76
Performance Ratio	1.0	1.9	3.6	5.3

Table 25 : Performance of 64-bit floating poinr FDM using Jacobi on Xilinx 2V6000 FPGA.

No. of PEs	4	6	8	10
Clock Freq.	70 MHz	64 MHz	60 MHz	Overmapped
Bandwidth (GB/s)	2.24	3.07	3.84	Overmappedd
GFLOPS (Double FP)	1.4	1.92	2.4	Overmapped
Performance Ratio	1.0	1.4	1.7	Overmapped

7.2.2 Analysis of the restrictions

One point that is worth noting is that a floating-point adder is much more complicated than a fixed-point adder. This is because the input operands for a floating-point adder must be shifted until they have the same exponent prior to addition, and the resulting output must then be shifted to be correctly normalized. This consumes a large amount of the FPGA logic resources, and can also reduce the maximum speed that the circuit can achieve. By contrast, fixed-point arithmetic requires no special circuitry to provide input alignment or output normalization.
The large logic requirements of floating-point arithmetic mean that an insufficient number of data paths will fit onto the FPGA. There is a loss of speed-up due to the communication cost on reading the data from external RAM. Fortunately, this loss can be ameliorated by holding a column of data within the FPGA for many sweeps of the iteration scheme. Each column of block RAM can be read and written simultaneously. The internal block RAM of the FPGA is therefore acting as a cache with a very large I/O capability.

For the floating-point arithmetic operators, Floating-point operator v2.0 of Xilinx Core

Generators was used to generate them from a parameterized library of pre-designed useful components. Tradeoffs can be made in terms of the latency, resources and maximum clock frequency. The maximum latency means fewer resources occupied and a faster clock speed achieved. The floating-point operation units in this design were chosen to work at their maximum latency, and then given an implementation with a higher throughput operating on multiple processing elements simultaneously.

There is another restriction to implement a hardware design scalably. This limitation involves the setting up and initiation of the system. In section 4.1.6, a host processor was introduced that is designed to be able to configure the FPGA and to pre-process the input data, then to initiate transfer to the external memory connected to the FPGA. The downloading of one of the sub-problems is accomplished whilst the FPGA board is computing the solution of a previous sub-problem. This eliminates communication and synchronization delays between the subdomains.

7.2.3 Scalability of the hardware implementation of FDM

The design implemented in section 5.3.1 used a medium speed grade of the 2V6000 FPGA in the Xilinx Virtex 2 series. This FPGA would have been typical of the state-of-the-art almost ten years ago, but newer, bigger and faster FPGA families are launched almost every year. In this section, we consider how the design would scale to larger and faster FPGAs, and also consider how the design would scale to the use of 64-bit floating point arithmetic.

Table 26 compares the logic, memory and arithmetic capacity of the FPGAs used in this study with the largest FPGAs available in the more modern Virtex 5 and Virtex 6 series. There are four different families of Virtex 5 FPGAs: the LX series (optimized for high-performance logic), the LXT series (optimized for high-performance logic with low power serial connectivity), the SXT series (optimized for DSP and memory-intensive applications with low-power serial connectivity) and the FXT series (optimized for embedded processing and memory-intensive applications with highest-speed serial connectivity). There are two sub-families of Virtex 6 FPGAs: the LXT series (high-performance logic with advanced serial connectivity) and SXT series (highest signal processing capability with advanced connectivity.

Table 26 : The logic, arithmetic and memory capacity of FPGAs [35].

Family	FPGA	CLK(MHz)	Block RAM Blocks [1]			Block multipliers/DSP slices [2]	Logic slices [3]	DCMs/CMTs /MMCMs [4]
			18 Kb	36 Kb	Max(Kb)			
Virtex II	**2V6000**	**400**	**144**	**---**	**2,592**	**144**	**33,792**	**12**
Virtex 4	**4VSX55**	**500**	**320**	**---**	**5,760**	**512**	**24,576**	**8**
	4VLX160	**500**	**288**	**---**	**5,184**	**96**	**152,064**	**12**
Virtex 5	5VLX330	550	576	288	10,386	192	51,840	6
	5VLX330T	550	648	324	11,664	192	51,840	6
	5VSX240T	550	1,032	516	18,576	1,056	37,440	6
	5VFX200T	550	912	456	16,416	384	30,720	6
Virtex 6	6VLX760	600	1,440	720	25,920	864	118,560	18
	6VSX475T	600	2,128	1,064	38,304	2,016	74,400	18

[1] Block RAMs in Virtex-5 and Virtex-6 FPGAs, which are fundamentally 36 Kbits in size, can be used as two independent 18 Kb blocks.

[2] Each Virtex-5/6 DSP slice contains one 25×18 multiplier, an adder and an accumulator, each Virtex-4 DSP slice contains one 18×18 multiplier, an adder and an accumulator; whereas the Virtex-II uses a dedicated 18×18 multiplier block.

[3] A logic slice is equivalent to about 200 2-input logic gates. Each Virtex-6 FPGA slice contains four LUTs and eight flip-flops, and each Virtex-5 slice contains four LUTs and four flip-flops, whereas earlier series used two LUTs and two flip-flops.

[4] Each CMT contains two DCMs and one PLL in Virtex-5 FPGAs, and each CMT contains two mixed-mode clock managers (MMCM) in Virtex-6 FPGAs.

Table 27 shows the number of logic slices and block multipliers required to build fixed and floating point adders and multipliers in 32-bit and 64-bit wordlength. (It should be noted that if the FPGA runs out of block multipliers, the multipliers can instead be constructed in the logic slices, but this will entail a speed penalty.) The floating point cores were generated by Xilinx CORE GENERATOR floating-point operator v2.0 in Virtex-2 and Virtex-4, by floating-point operator v3.0 in Virtex-5, and by floating-point operator v5.0 in Virtex-6. The hardware designs of FDM were implemented on Virtex 2V6000, Virtex 4VLX160 and Virtex 4VSX55 FPGAs, and the number of data paths that were successfully implemented on each FPGA is listed in Table 28. In addition, the maximum number of data paths, as shown in Figure 53, can be projected in Virtex-5 and Virtex-6 FPGAs by using information from their data sheets.

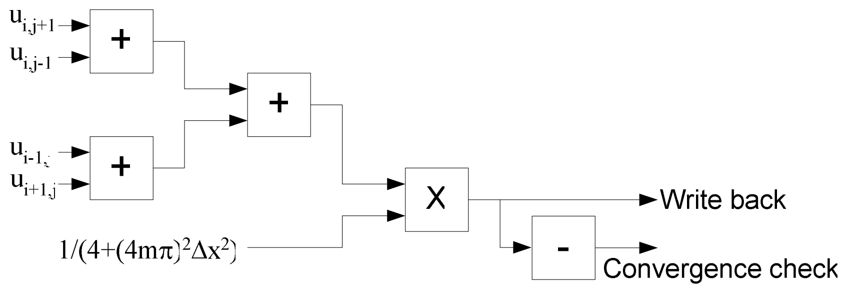


Figure 53 : Processing element for row update in FDM.

Table 27 : The resources consumed by addition and multiplication.

	32-bit fixed point		64-bit fixed point		32-bit float point		64-bit float point	
	Block Mult	Logic slices	Block Mult	Logic slices	Block Mult	Logic slices	Block Mult	Logic slices
Adder	0	16	0	32	0	329	0	692
Multiplier	2	73	4	146	4	139	16	540

(a) Implementation on Virtex-2 [106] [107]

	32-bit fixed point		64-bit fixed point		32-bit float point		64-bit float point	
	DSP48s	Logic slices	DSP48s	Logic slices	DSP48s	Logic slices	DSP48s	Logic slices
Adder	0	16	0	32	0	329	0	692
Multiplier	2	73	4	146	4	118	16	491

(b) Implementation on Virtex-4 [106] [107]

	32-bit fixed point		64-bit fixed point		32-bit float point		64-bit float point	
	DSP48Es	Logic slices	DSP48Es	Logic slices	DSP48Es	Logic slices	DSP48Es	Logic slices
Adder	0	8	0	16	0	141	0	265
Multiplier	2	37	4	73	2	53	12	168

(c) Implementation on Virtex-5 [108] [107]

	32-bit fixed point		64-bit fixed point		32-bit float point		64-bit float point	
	DSP48E1s	Logic slices	DSP48E1s	Logic slices	DSP48E1s	Logic slices	DSP48E1s	Logic slices
Adder	0	8	0	16	0	123	0	229
Multiplier	2	37	4	73	2	39	10	93

(d) Implementation on Virtex-6 [108] [109]

The floating-point operation units in this design were chosen to work at their maximum latency (in order to maximize throughput). In Table 28, the maximum numbers of data paths, as shown in Figure 53, can be built in the various series of Xilinx FPGAs.

Table 28 : The number of copies of Figure 53 that can be built in each of the FPGAs

	32-bit fixed	64-bit fixed [+]	32-bit float	64-bit float
2V6000	64	36	24	8
4VSX55	64	36	16	8
4VLX160	128	72	48	16
5VLX330 [+]	288	144	108	48
6VLX760 [+]	720	360	290	144

[+] The results estimated using the data in Table 26 and Table 27.

The single precision floating-point implementation on the 2V6000 operates at 98 MHz and the double precision at 60 MHz. On the Virtex 4, the frequency of operation for the datapath of Figure 53 (both single and double precision) rises to 110 MHz. The optimal implementations for the 5VLX330 and 6VLX760 are expected to run at a saturated clock frequency of 130MHz, due to the behaviour of the surrounding memory. Table 29 shows how the capacity estimates of Table 28, combined with data on the frequency of operation of the data paths, translate into projections for the number of computations that can be performed per second in Eq. (120) and memory bandwidth in Eq. (121).

Table 29 : Performance and bandwidth achievable on each FPGA.

	Single Floating-point Precision		Double Floating-point Precision	
	Performance (GFLOPs)	Bandwidth (GByte/s)	Performance (GFLOPs)	Bandwidth (GByte/s)
2V6000	11.76	9.4	2.4	3.8
4VSX55	8.8	7.0	4.4	7.0
4VLX160	26.4	21.1	8.8	14.1
5VLX330$^+$	70.2	56.1	31.2	49.9
6VLX760$^+$	188.5	150.8	93.6	149.7

The aim of our study is to explore the possibilities of using the internal memory of the FPGAs as a cache that can supply a very high memory bandwidth to support a very large number of parallel pipelines. In our design, each column of block RAM can be read and written simultaneously. The internal block RAM of the FPGA is therefore acting as a cache with a very large I/O bandwidth.

Moreover, it should be noted that these results refer to the speed-up achieved for a single FPGA board. If the problem is solved across a number n of FPGA boards, the speed-up will scale linearly with n for values of n less than about 16 [110]. This is due to the domain decomposition approach used, which renders the sub-problems independent, thus avoiding saturation of speed-up due to communications overheads.

7.3 Summary

The purpose of this chapter was to analyze the potential for high levels of parallelism within the FPGAs by using very deep computational pipelines on numerical solutions for partial differential equations.

To realize the full potential of such approaches, the underlying algorithms must be inherently parallelizable. Reconfigurable hardware devices have been shown to outperform general-purpose CPUs in the computationally intensive applications considered, because they can exploit a great degree of pipelinability and parallelism in the algorithm in a much more thorough way than can be done with uni-processor or parallel computing.

Firstly, the performance at low and high levels of parallelism has been discussed. The performance of both single precision and double precision floating-point arithmetic implementations increases nearly linearly with the number of processing elements, even when there is a loss of system frequency due to congestion arising from the double precision floating-point operators. Then, the features of the system that limit the speed-up achievable were analysed, with a discussion of the effect that future developments in reconfigurable computing architecture will have on these figures. Finally, a study of how great a performance level can be achieved by using the largest and fastest FPGA that is currently readily available has been given.

The method has been chosen to minimize communication overheads, maximize parallelism, to be scalable for large problem sizes, and to be scalable to larger FPGAs. Overall, it has been clearly demonstrated that the architecture of the hardware design scales linearly with multiple processors working in parallel. Furthermore, the design is scalable for the features of new FPGAs and will provide very promising results in future.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, the hardware/software implementation of a number of numerical solutions to Partial Differential Equations in reconfigurable computing systems based on FPGAs has been investigated. After providing relevant background to PDEs, and basic methodologies of PDE problems, a literature review of some previous parallel implementations of solutions to PDEs was presented. However, due to communication overheads and load balancing problems, speed-ups were far less than linear. The aim of this study, consequently, is to investigate the inherent parallelism inside the numerical algorithms and implement the re-organized algorithms onto FPGAs in order to accelerate the computations. By using domain decomposition, the main arithmetic operations can be implemented in parallel. Details of these implementations have then been given. An analysis of the results of hardware implementations with the comparison to the software equivalents and the requirements of such a system has been presented. The effect on speed-up and bandwidth has been discussed for complex problems.

The following topics have been considered in this study:

- FPGA-based reconfigurable computing system for the 3-D finite difference solution of the Laplace Equation

A system was designed whose software host uses Fourier decomposition to decompose the 3D finite difference problem into a series of 2D problems that can be downloaded into the reconfigurable computing boards. This decomposition gives a much better match between the processor/memory capabilities of the FPGAs and the properties of the sub-problems that they are used to solve. Downloading of one of the sub-problems is accomplished whilst the FPGA board is computing the solution of a previous sub-problem. This approach is based on the efficient use of the on-chip RAM within the FPGA as a cache to store the working set for the current sub-problem, and also to perform background upload and download of data for the previous and following sub-problem to be tackled by the FPGA.

- Reconfigurable hardware acceleration for a Preconditioned Conjugate Gradient solver of the 3D tetrahedral finite elements

A novel parallel hardware architecture for the preconditioned Conjugate Gradient solution of an element-by-element approach to finite element analysis has been presented. The solution of the large linear system equations is the most time-consuming part of the FEM. By using an Element-by-Element (EBE) scheme, the large sparse matrix-vector multiplication is divided into a number of independent sub-multiplications in order to improve data reusability and save external memory accesses. The use of FPGA hardware can help to deliver the solution of the large system of equations resulting from 3D FE analysis in a faster and more cost effective manner.

- Domain decomposition for 3-D problems

Domain decomposition is necessary for 3-D PDEs problems to be able to exploit the high level of parallelism that can be achieved by reconfigurable computing. The requirement for very high levels of parallelism means that it is important to ensure that the hardware pipelines do not become starved of data. By using suitable domain decomposition methodologies, the data independency is exploited in order to achieve a high throughput of data from memory and to improve the speed-up.

- Parallel architectures of several iterative methods

Stationary iterative methods and non-stationary iterative methods have been discussed and implemented. An efficient memory hierarchy has been adopted in order to alleviate the memory bandwidth bottleneck.

- Customized fixed-point arithmetic operations on hardware/software FPGA-based co-processors

Customized fixed-point arithmetic has been used in order to conserve hardware resources. This enables one entire 2-D section of the problem to fit in a single FPGA chip and demonstrates how well the inherent parallelism of the algorithms can be exploited.

- Floating-point arithmetic operations on hardware/software FPGA-based co-processors

As fixed point is suitable only for a restricted class of problems, which have very favourable error propagation characteristics and a very small dynamic range of problem parameters, floating-point arithmetic is considered for the most implementations of this study in order to achieve a good level of accuracy. One consequence of the use of floating point arithmetic is that we can not fit as high a level of parallel computations onto the FPGA as with fixed-point arithmetic, thus increasing the complexity of the hardware design. An optimized memory hierarchy has been introduced. The significant differences between the fixed-point and floating-point hardware versions have been investigated.

Some parts of the applications are well-suited to computation on an FPGA, while others are better processed in software. By using a hardware/software co-processor, it has been shown that a reconfigurable computing accelerator based on FPGAs can outperform software implementation for these computationally expensive algorithms. A speed-up of 105 has been achieved for 64×64×64 FDM problem using fixed-point arithmetic, and a speed-up of 38 has been achieved by using floating-point arithmetic. Because of the complexity of the floating-point arithmetic operators, a smaller number of data paths can be implemented parallel and the speed-up of the hardware implementation is decreased. The speed-up of the finite element method was a factor of 40 for a 3D tetrahedral finite element problem with 48,000 elements using the preconditioned conjugate gradient method.

This study is a good start to explore the inherent parallelism inside several numerical solutions to the partial differential equations and use FPGA-based reconfigurable hardware to accelerate these computationally expensive problems. In civil engineering, problems such as the modelling of water seepage through a dam or earthquake displacement analysis may take several days or even longer for numerical approximation. The use of domain decompositions avoids saturation of speed-up due to communication overheads, so the reconfigurable hardware can rival expensive parallel computers to solve such timing consuming problems and achieve a linear speed-up without saturation as the problem size becomes large. In addition, because the architecture of the hardware designs scales linearly with multi-processor implemented in parallel, the hardware designs can be easily resized or transferred to a latest, largest and fastest FPGA in order achieve further speed-up.

8.2 Future Work

The scalability of the parallel system has only been demonstrated on a single FPGA board. It would be useful to implement a system with a large number of FPGAs, as in a reconfigurable supercomputer, in order to evaluate how well the load can be balanced and the size of the communication overheads. The use of multiple FPGAs in parallel will achieve further speed-up.

FPGAs are increasing rapidly in logic and speed, and the loss of speed-up associated with floating-point arithmetic can be offset by migrating to higher speed grades of more modern FPGAs. A larger numerical precision is desirable for this study. As more modern FPGAs contain more embedded multipliers, there is a much greater freedom to implement 64-bit floating-point precision hardware solutions in order to investigate of the effect of numerical precision.

Solutions for other numerical PDE problems would be another useful extension of this study, as the problem targeted in the thesis is mainly the solution to the Laplace equation.

REFERENCES:

[1] A. Jacob, J. Lancaster, J. Buhler, and R. D. Chamberlain, "FPGA-accelerated seed generation in Mercury BLASTP," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*: IEEE Computer Society, 2007.

[2] B. D. Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks, "FPGA accelerator for real-time skin segmentation," *2006 IEEE/ACM/IFIP workshop on Embedded Systems for Real Time Multimedia,* pp. pp. 93-97, 2006.

[3] Z. K. Baker and V. K. Prasanna, "Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs," in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*: IEEE Computer Society, 2005.

[4] A. Das, S. Misra, S. Joshi, J. Zambreno, G. Memik, and A. Choudhary, "An efficient FPGA implementation of principle component analysis based network intrusion detection system," in *Proceedings of the conference on Design, automation and test in Europe* Munich, Germany: ACM, 2008.

[5] H. Brezis and F. Browder, "Partial differential equations in the 20th century," *Advances in Mathematics,* vol. 135, pp. 76-144, Apr 15 1998.

[6] G. D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford: Oxford University Press, 1978.

[7] G. Strang and G. Fix, *An analysis of the finite element method*: Englewood Cliffs, Prentice Hall, 1973.

[8] R. LeVeque, *Finite volume methods for hyperbolic problems*. Cambridge: Cambridge University Press, 2002.

[9] P. K. Banerjee, *Boundary element methods in engineering*. London: McGraw-Hill, 1994.

[10] G. R. Liu, *Mesh Free Methods.* Singapore: CRC Press, 2002.

[11] D. Gottlieb and S. Orzag, *Numerical Analysis of Spectral Methods: Theory and Applications*. Philadephia: SIAM, 1977.

[12] Y. Saad, *Iterative methods for sparse linear systems*: Philadelphia SIAM, 2003.

[13] J. D. Hoffman, *Numerical Methods for Engineers and Scientists*. Singapore: McGraw-Hill, Inc., 1992.

[14] A. D. Polyanin, *Handbook of Linear Partial Differential Equations for Engineers and Scientists*. Hoboken: CRC Press Company, 2002.

[15] A. J. Davies, *The finite element method: a first approach.* Oxford: Oxford University Press, 1980.

[16] T. J. R. Hughes, I. Levit, and J. Winget, "An element-by-element solution algorithm for problems of structural and solid mechanics," *Computer Methods in Applied Mechanics and Engineering,* vol. 36, p. 241, 1983.

[17] G. F. Carey, E. Barragy, R. McLay, and M. Sharma, "Element-by-element vector and parallel computations," *Communications in Applied Numerical Methods,* vol. 4, pp. 299-307, 1988.

[18] E. W. Weisstein, "Gaussian Elimination." vol. 2010: MathWorld-A Wolfram Web Resource.

[19] G. Golub and J. M. Ortega, *Scientic Computing: An Introdction with Parallel Computing*. New York: Academic Press INC, 1993.

[20] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. d. Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[21] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method without the Agonizing Pain," 1994.

[22] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Reseach: A View from Berkeley," EECS Department, University of California, Berkeley UCB/EECS-2006-183, December 18 2006.

[23] P. Chow and M. Hutton, "Integrating FPGAs in high-performance computing: introduction," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays* Monterey, California, USA: ACM, 2007.

[24] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems." vol. 2006: Hindawi Publishing Corp., 2006, pp. 13-13.

[25] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004*

International Conference on Management of Data, 2004.

[26] L. Nyland, M. Harris, and J. Prins, "Fast N-Body simulation with CUDA." In *GPU GEMs 3*. H. Nguyen, ed. Addison-Wesley, 2007.

[27] M. Januszewski and M. Kostur, "Accelerating numerical solution of stochastic differential equations with CUDA," *Computer Physics Communications,* vol. 181, pp. 183-188, 2010.

[28] A. Humber, "New NVIDIA Tesla GPUs Reduce Cost of Supercomputing by A Factor of 10." vol. 2010, 2009.

[29] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt, "Have GPUs made FPGAs redundant in the field of video processing?," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on* Singapore, 2005.

[30] B. Cope, "Implementation of 2D Convolution on FPGA, GPU and CPU," Imperial College, College Report.

[31] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute intensive applications with GPUs and FPGAs," in *In 6th IEEE Symposium on Application Specific Processors*, Anaheim CA USA, 2008.

[32] L. W. Howes, P. Price, O. Mencer, O. Beckmann, and O. Pell, "Comparing FPGAs to graphics accelerations and the Playstation 2 using a unified source description," in *Proceeding of the 2006 International Conference of Field Programmable Logic and Applications*, 2006, pp. 1-6.

[33] M. Dubash, "Moore's Law is dead, says Gordon Moore." vol. 2010, 2005.

[34] Xilinx, "How Xilinx Began.", http://www.xilinx.com/company/history.htm, 2009. [Accessed May 24, 2010]

[35] http://www.xilinx.com/products/devices.htm. [Accessed May 24, 2010]

[36] G. R. Morris and V. K. Prasanna, "An FPGA-based floating-point Jacobi iterative solver," in *Parallel Architectures,Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, 2005, pp. 420-427.

[37] G. Gokul, L. Zhuo, S. Choi, and V. A. P. V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004, p. 149.

[38] J. Foertsch, J. Johnson, and P. Nagvajara, "Jacobi load flow accelerator using FPGA," in *Power Symposium, 2005. Proceedings of the 37th Annual North American*, 2005, p. 448.

[39] E. Motuk, R. Woods, and S. Bilbao, "FPGA-based hardware for physical modelling sound synthesis by finite difference schemes," in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, 2005, p. 103.

[40] K. D. Underwood and K. S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*: IEEE Computer Society, 2004.

[41] P. A. Cundall, "A discontinuous future for numerical modelling in geomechanics," *Proceedings of the Institution of Civil Engineers-Geotechnical Engineering,* vol. 149, pp. 41-47, Jan 2001.

[42] N. Tredennick and B. Shimamoto, "Reconfigurable systems emerge," in *Proceedings of the 2004 International Conference on Field Programmable Logic and Its Applications, ser. Lecture Notes in Computer Science*, 2004, pp. 2-11.

[43] S. Hauck, "The roles of FPGAs in reprogrammable systems," *Proceedings of the IEEE,* vol. 86, pp. 615-638, April 1998.

[44] www.xilinx.com/support/sw_manuals/xilinx8/index.htm. [Accessed May 24, 2010]

[45] J. M. Ortega and R. G. Voigt, "Solution of Partial Differential Equations on Vector and Parallel Computers," *Society for Industrial and Applied Mathematics,* vol. 27, 1985.

[46] J. Ericksen, "Iterative and direct methods for solving Poisson's euqation and their adaptability to ILLIAC IV," Univ Illinois, Urbana-Champaign 1972.

[47] http://research.microsoft.com/en-us/um/people/gbell/Computer_Structures_Principles_

and_Examples/c sp0769.htm. [Accessed May 24, 2010]

[48] L. Hayes, "Comparative analysis of iterative techniques for solving Laplace's equation on the unit square on a parallel processor," in *Dept. Mathematics*. vol. M. S. Austin: Univ. Texas, 1974.

[49] L. N. Long and J. Myczkowski, "Solving the Boltzmann equation at 61 gigaflops on a 1024-node CM-5," in *Supercomputing '93. Proceedings*, 1993, pp. 528-534.

[50] X.-H. Sun and R. D. Joslin, "A Massively Parallel Algorithm for Compact Finite Difference Schemes," in *Proceedings of the 1994 International Conference on Parallel Processing - Volume 03*: IEEE Computer Society, 1994.

[51] M. M. Shearer, "Computational optimization of finite difference methods on the CM5," *Parallel Computing,* vol. 22, pp. 465-481, March 1996.

[52] Y. Lu and C. Y. Shen, "A Domain Decomposition Finite-Difference Method for Parallel Numerical Implementation of Time-Dependent Maxwell's Equations," *IEEE TRANSACTIONS ON ANTENNAS AND PROPAGATION,* vol. 45, pp. 556-562, March 1997.

[53] T. Bohlen, "Parallel 3-D viscoelastic finite difference seismic modelling," *Computers & Geosciences,* vol. 28, pp. 887-899, Oct. 2002.

[54] Peter Widas, "Introduction to Finite Element Analysis," http://www.sv.vt.edu/classes/MSE2094_NoteBook/97ClassProj/num/widas/history.html, April 8th, 1997. [Accessed May 24, 2010]

[55] M. J. Turner, R. W. Clough, H. C. Martin, and L. P. Topp, "Stiffness and deflection analysis of complex structures," *J. Aeronautical Society,* vol. 23, 1956.

[56] O. O. Storaasli, L. Adams, J. Knott, T. Crockett, and S. W. Peebles, "The Finite Element Machine: An Experiment in Parallel Processing," in *Proc. Symposium on Advances & Trends in Structural & Solid Mechanics* Washington DC, 1982.

[57] R. E. Fulton, "The finite element machine: an assessment of the impact of parallel

computing on future finite element computations." vol. 2: Elsevier Science Publishers B. V., 1986, pp. 83-98.

[58] T. Nakata, Y. Kanoh, N. Koike, H. Okumura, K. Ohtake, T. Nakamura, and M. Fukuda, "Evaluation of finite element analysis on the parallel simulation machine Cenju," in *NAL, Proceedings of the Ninth NAL Symposium on Aircraft Computational Aerodynamics*, 1991, pp. 83-89.

[59] Y. Kanoh, T. Nakata, H. Okumura, K. Ohtake, and N. Koike, "Large deformation finite element analysis on the parallel machine Cenju," *NEC research & development,* vol. 34, pp. 350-359, 1993.

[60] Lee Margetts, http://www.rcs.manchester.ac.uk/research/avp. [Accessed May 24, 2010]

[61] L. Margetts, M. Pettipher, and I. Smith, "Parallel Finite Element Analysis," in *CSAR Focus*, 10 ed, pp. 7-8.

[62] R. Baxter, S. Booth, M. Bull, G. Cawood, K. D'Mellow, X. Guo, M. Parsons, J. Perry, A. Simpson, and A. Trew, "High-Performance Reconfigurable Computing - the View from Edinburgh," in *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*: IEEE Computer Society, 2007.

[63] K. J. Paar and P. M. Athanas, "Accelerating finite-difference analysis simulations with a configurable computing machine," *Microprocessors and Microsystems,* vol. 21, pp. 223-235, 30 December 1997.

[64] O. Storaasli, R. C. Singleterry, and S. Brown, "Scientific Applications on a NASA

Reconfigurable Hypercomputer," in *5th Military and Aerospace Programmable Logic Devices (MAPLD) Conference*, 2002.

[65] O. Storaasli, "Computing Faster without CPUs: Scientific Applications on a Reconfigurable, FPGA-based Hypercomputer," in *6th Military and Aerospace Programmable Logic Devices (MAPLD) Conference*, 2003.

[66] Z. Ling and K. P. Viktor, "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2005.

[67] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 FPGA Supercomputer," in *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems*: IEEE Computer Society, 2007.

[68] C. Benjamin, S. n, fer, F. Q. Steven, and H. C. C. Andrew, "Scalable Implementation of the Discrete Element Method on a Reconfigurable Computing Platform," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*: Springer-Verlag, 2002.

[69] Z. Ling and K. P. Viktor, "Sparse Matrix-Vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, Monterey, California, USA, 2005, pp. 63-74.

[70] G. R. Morris and V. K. Prasanna, "Sparse Matrix Computations on Reconfigurable

Hardware." vol. 40: IEEE Computer Society Press, 2007, pp. 58-64.

[71] A. R. Lopes and G. A. Constantinides, "A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation," in *Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications* London, UK: Springer-Verlag, 2008.

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

[73] J. R. Marek, M. A. Mehalic, and A. J. Terzuoli, "A dedicated VLSI architecture for Finite-Difference Time Domain (FDTD) calculations," in *8th Annual Review of Progress in Applied Computational Electromagnetic*, 1992, pp. 546-553.

[74] P. Placidi, L. Verducci, G. Matrella, L. Roselli, and P. Ciampolini, "A custom VLSI architecture for the solution of FDTD Equations," *IEICE Transactions on Electronics,* vol. E85-C, pp. 572-577, 2002.

[75] Y. El-Kurdi, D. Giannacopoulos, and W. J. Gross, "Hardware Acceleration for Finite-Element Electromagnetics: Efficient Sparse Matrix Floating-Point Computations With FPGAs," *Magnetics, IEEE Transactions on,* vol. 43, p. 1525, 2007.

[76] C. Dufour, J. Belanger, S. Abourida, and V. A. L. V. Lapointe, "FPGA-Based Real-Time Simulation of Finite-Element Analysis Permanent Magnet Synchronous

Machine Drives," in *Power Electronics Specialists Conference, 2007. PESC 2007. IEEE*, 2007, p. 909.

[77] D. Gregg, C. M. Sweeney, C. McElroy, F. A. C. F. Connor, S. A. M. S. McGettrick, D. A. M. D. Moloney, and D. A. G. D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, 2007, p. 786.

[78] G. A. Gravvanis and K. M. Giannoutakis, "Parallel approximate finite element inverse preconditioning on distributed systems," in *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on*, 2004, p. 277.

[79] P. Subramaniam and N. Ida, "A parallel algorithm for finite element computation," in *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, 1988, p. 219.

[80] L. Yu, K. Ramdev, and K. K. Tamma, "An efficient parallel finite-element-based domain decomposition iterative technique with polynomial preconditioning," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 2006, p. 6 pp.

[81] E. M. Ortigosa, L. F. Romero, and J. I. Ramos, "Parallel scheduling of the PCG method for banded matrices rising from FDM/FEM," *J. Parallel Distrib. Comput.,* vol. 63, pp. 1243-1256, 2003.

[82] J. Sun, G. Peterson, and O. Storaasli, "Sparse Matrix-Vector Multiplication Design on FPGAs," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 2007, p. 349.

[83] G. Goumas, K. Kourtis, N. Anastopoulos, V. A. K. V. Karakasis, and N. A. K. N. Koziris, "Understanding the Performance of Sparse Matrix-Vector Multiplication," in *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, 2008, p. 283.

[84] M. G. Ancona, *Computational Methods for Applied Science and Engineering: An Interactive Approach.* Paramus USA: Rinton Press, 2003.

[85] M. R. Gosz, *Finite Element Method: Applications in Solids, Structures, and Heat Transfer*. USA: CRC Press, 2005.

[86] D. J. Evans, *The analysis and application of sparse matrix algorithms in the finite element method*. London: Academic Press, 1973.

[87] B. H. V. Topping and A. I. Khan, *Parallel Finite Element Computations.* Edinburgh: Saxe-Coburg Publications, 1996.

[88] S. W. Bova and G. F. Carey, "A distributed memory parallel element-by-element scheme for semiconductor device simulation," *Computer Methods in Applied Mechanics and Engineering,* vol. 181, p. 403, 2000.

[89] T. J. R. Hughes, R. M. Ferencz, and J. O. Hallquist, "Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients," *Comput. Methods Appl. Mech. Eng.,* vol. 61, pp. 215-248, 1987.

[90] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 2, pp. 365-367, September 1994.

[91] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transaction on CAD of Integrated Circuits and Systems,* vol. 22, pp. 1432-1442, 2003.

[92] G. A. Constantinides, "Word-length optimization for differentiable nonlinear systems," *ACM Transaction on Design Automation of Electronic Systems,* vol. 11, pp. 26-43, 2006.

[93] C. T. Ewe, "Dual fixed-point: an efficient alternative to floating-point computation for DSP applications," in *Field Programmable Logic and Applications, 2005. International Conference on*, 2005, pp. 715-716.

[94] http://www.alpha-data.co.uk/products.php?product=adm-xrc-ii. [Accessed May 24, 2010]

[95] www.xilinx.com/products/silicon_solutions/fpgas/virtex/. [Accessed May 24, 2010]

[96] www.agilityds.com/literature/rc2000_datasheet_01000_hq_screen.pdf. [Quick View accessed May 24, 2010]

[97] J. Hu, E. Stewart, S. Quigley, and A. Chan, "Solution of the 3D finite difference equations using parallel reconfigurable hardware accelerators," in *the Eighth International Conference on Computational Structures Technology*, 2006.

[98] L. A. Win, "RC2000 (ADM-XRC-II) API v1.04," 2006.

[99] IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std. 754-1985.

[100] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. Numerical Recipes in FORTRAN 77: The art of scientific computing. Cambridge. Cambridge University Press, 1992.

[101] Peter P. Silvester and Ronald L. Ferrari. Finite Elements for Electrical Engineers, Third Edition. Cambridge, Cambridge University Press, 1996.

[102] Singiresu S. Rao. The Finite Element Method in Engineering. Oxford, Elsevier Butterworth-Heinemann, 2005.

[103] A. Wood, *Introduction to Numerical Analysis*. Harlow: Addison-Wesley,1999.

[104] http://www.cray.com/Home.aspx. [Accessed May 24, 2010]

[105] http://www.srccomp.com/index.asp. [Accessed May 24, 2010]

[106] "Xilinx ISE CORE GENERATOR floating-point operator v2.0.pdf."

[107] M. Bečvář and P. Štukjunger, "Fixed-Point Arithmetic in FPGA," *Acta Polytechnica,* vol. 45, pp. 67-72, 2005.

[108] "Xilinx ISE CORE GENERATOR floating-point operator v3.0.pdf."

[109] "Xilinx ISE CORE GENERATOR floating-point operator v5.0.pdf."

[110] B. Carrion Schafer, S. F. Quigley, and A. H. C. Chan, "Acceleration of the Discrete

Element Method (DEM) on a reconfigurable co-processor," *Computers & Structures,* vol. 82, p. 1707, 2004.

APPENDIX: PUBLICATIONS

International conference papers

1) J. Hu, EJC Stewart, SF Quigley, AHC Chan, "*FPGA-based reconfigurable computing system for the three-dimensional finite difference solution of the Laplace Equation*", the 14th Annual Conference of the Association for Computational Mechanics in Engineering (ACME) 2006.
2) J Hu, EJC Stewart, SF Quigley and AHC Chan, "*Solution of the 3D Finite Difference Equations using Parallel Reconfigurable Hardware Accelerators*", the Eighth International Conference on Computational Structures Technology 2006.
3) J Hu, EJC Stewart, SF Quigley and AHC Chan, "*FPGA-based Acceleration of 3D Finite Difference Floating Point Solution of the Laplace Equation*", the 15th Annual Conference of the Association for Computational Mechanics in Engineering (ACME) 2007.
4) J Hu, SF Quigley and AHC Chan, "*Reconfigurable Hardware Acceleration for a Preconditioned Conjugate Gradient Solver of the 3D Tetrahedral Finite Elements*", the 16th Annual Conference of the Association for Computational Mechanics in Engineering (ACME) 2008.
5) J Hu, SF Quigley and AHC Chan, "*Reconfigurable Hardware Acceleration for a Preconditioned Conjugate Gradient Solver of the 3D Tetrahedral Finite Elements*", the Ninth International Conference on Computational Structures Technology 2008.

International conference posters

6) J Hu, Steven F. Quigley, Andrew Chan, "An Element-by-element Preconditioned Conjugate Gradient Solver of 3D Tetrahedral Finite Elements on an FPGA Coprocessor", the 18th International Conference on Field Programmable Logic and Applications 2008.