A Multiple-FPGA Parallel Computing Architecture for Real-time Simulation of Deformable Objects

A MULTIPLE-FPGA PARALLEL COMPUTING ARCHITECTURE FOR REAL-TIME SIMULATION OF DEFORMABLE OBJECTS

BY

SEYED BEHZAD MAHDAVIKHAH MEHRABAD, B.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2009) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	A Multiple-FPGA Parallel Computing Architecture for Real-time Simulation of Deformable Objects
AUTHOR:	Seyed Behzad Mahdavikhah Mehrabad B.Sc., (Electrical Engineering) Sharif University of Technology, Tehran, Iran
SUPERVISOR:	Dr. Shahin Sirouspour
NUMBER OF PAGES:	xi, 125

Abstract

In recent years there has been a growing interest in computer-based surgical planning, virtual-reality enabled training of medical procedures, and computer gaming all involving non-rigid deformable objects. High-fidelity simulations of haptic interaction with deformable objects is computationally demanding. The Finite Element Method (FEM) is known to produce relatively accurate solution for continuum mechanics-based models of soft-object deformation. Linear elastic FE models require solving a large sparse system of equations. The solution accuracy can be improved by increasing the resolution of the finite element mesh resulting in a larger number of equations and hence greater computational complexity. Depending on the mechanical characteristics of the soft-object, to maintain stability and high fidelity in haptic interaction, the update rate should be in the range of 100-1000Hz. This, for example, means that for a moderately-sized three-dimensional mesh of 6000 nodes, a set of 18000 linear equations must be solved within 1-10ms.

In this thesis, hardware-based parallel computing is proposed for finite-element (FE) analysis of soft-object deformation models. In particular, a distributed implementation of the (CG) algorithms on N Field Programmable Gate Array (FPGA) devices connected in a ring configuration is developed. This Parallel architecture can be utilized to solve the large system of equations arising from FE models at

high update rates required for stable haptic interaction. Massive parallelization of the computations is achieved by customizing the hardware architecture to the problem at hand and employing a large number of adaptive fixed-point computing units in parallel. The proposed hardware architecture satisfies three important criteria: (i) it meets the haptic rendering timing constraint by enabling an update rate of 400Hz; (ii) it attempts to simulate as many nodes as possible, given the available resources on the FPGA devices employed in this work and (iii) it is scalable both within an FPGA and also across multiple FPGA devices.

This research builds upon our group's earlier work in [1]. In that paper a novel highly parallelized single-FPGA architecture was proposed for solving system of equations arising from FEM using Conjugate gradient method. In this thesis, a multiple-FPGA architecture based on that design has been proposed. The contributions in the new multiple-FPGA design can be summarized as follows.

- proposing a novel method for expansion of conjugate gradient (CG) algorithm to multi processors.
- a new sparse matrix by vector multiplication unit, performing as the kernel of our hardware-based CG solver.
- proposing a novel storage format (SMVIS) for storing a vector, pre-multiplied by an sparse matrix which tremendously reduces the memory required vector storage.
- developing a new memory architecture for storing vectors in the CG algorithm, making the design capable of performing vector operations in the CG

algorithm regardless of their lengths.

- developing a novel communication scheme for Inter-FPGA communications in multiple-FPGA implementation of the CG algorithm.

An implementation of this scalable hardware accelerator on a quad-FPGA system has enabled real-time simulation of haptic interaction with a three-dimensional FE model of 6000 nodes at update rate of 400Hz. Both static and dynamic linear elastic models have been successfully simulated.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the early 20th century, psychophysicists introduced the word haptics from the Greek *haptesthai* meaning to touch in order to label the subfield of their studies that addressed human touch-based perception and manipulation [5]. In the early 1990s a new usage of the word haptics began to emerge. "The confluence of several emerging technologies made virtualized haptics, or computer haptics, possible" [6], [5]. Computer haptics allows a user to interact with the virtual objects, receiving kinesthetic, force and tactile feedback. This interaction between user and the virtual environments is fulfilled using bidirectional human-machine interfaces called *haptic devices* which allow users to receive force-feedback based on an interaction model. Fig. 1.1 illustrates an example for such haptic interaction with a virtual soft-object utilizing a haptic device.

Modern computers have the capability for visual and auditory interactions with their users. The next generation of computer interfaces will add the haptic feel

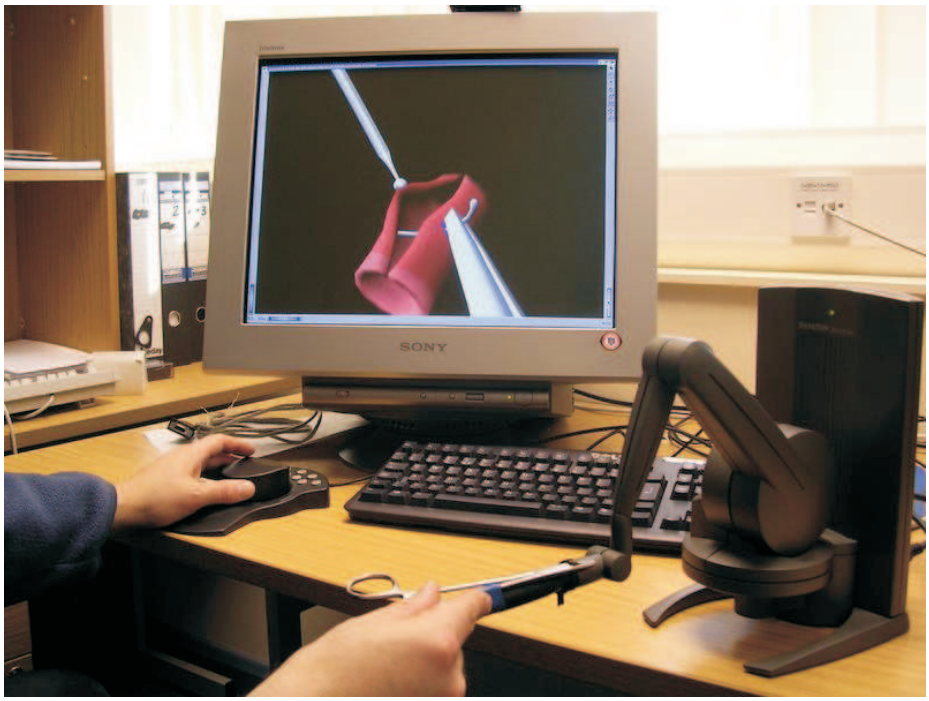

Figure 1.1: An example of employing haptic interfaces for modeling a soft-object. figure from [2]

to the computers, allowing their users to feel and touch the virtual objects. Such computer interactions have been already developed in simple forms such as vibrations and force feedback in computer games using rudimentary joysticks or racing wheels. The current force feedback in these applications is predominantly from interaction with rigid objects. Modeling of such interactions is relatively straightforward and has been extensively studied in the literature [7]. The ultimate form of computer haptics would involve simulating interactions with both deformable and rigid objects.

Real-time haptic rendering of deformable objects can be helpful in applications involving interaction with a non-rigid soft object. It is especially of a great interest in gaming, surgical training and surgery planning.

In surgical training, having a realistic simulation of biological soft-tissue allows

surgeon to rehearse operation without exposing the patient to undue risk. This application of soft-tissue modeling has become more relevant since the emergence of robotics-assisted in minimally invasive surgery. The novelty of such systems for most surgeons demands extend training to acquire the required skills. Such training can be enabled by virtual-reality based systems involving haptics and visual feedback. Moreover in surgical planning for procedures such as needle insertion, pre-operative plans usually need to be updated in real time due to soft-tissue deformations or organ movements during the operation. In such cases, a simulator can allow the surgeon to examine different task scenarios using the latest information available and choose the best course of action.

1.2 Problem Definition

Real-time haptic and deformation rendering of soft objects can be challenging due to massive computations that most be performed within a very short time to achieve the required high simulation update rate. Violating a minimum update rate can degrade the quality and accuracy of the simulation and even lead to interaction instability when haptic feedback is involved. This accentuates the need for powerful computational engine capable of performing these necessary computations within the permissible time.

Continuum mechanics based models can be utilized to accurately model soft-object deformation. Such a modeling approach will produce boundary condition Partial Differential Equations (PDEs) which generally do not have a closed-form solution except in special cases [8]. A numerical method such as the FEM can be employed to find an approximate solutions for these PDEs.

The application of the FEM to linear elastic deformation models will result in a large sparse set of equations in the form of $\mathbf{Kx} = \mathbf{b}$, where **K** is the stiffness matrix, **b** is the load vector and x is the vector of unknown deformations. In the FEM, the object is represented with a mesh of nodes forming 2D (e.g. triangular) or 3D (e.g. tetrahedral) elements. The total number of nodes determines the size of K matrix which would be $3n * 3n$ for a 3D model, where n is the total number of nodes in the mesh. As can be presumed intuitively, increasing the number of nodes in the mesh would improve the accuracy of the approximate solution obtained with the FE method.

While an update rate as low as 15-30Hz would be acceptable for graphics rendering, stable haptics interaction with virtual objects would require update rates in the range of 100-1000Hz. This higher update rate needed for haptics rendering imposes a stringent timing constraint on the simulation. The main computational problem in simulating FE-based deformation models is in solving the set of linear equations $\mathbf{Kx} = \mathbf{b}$ in real time. To obtain an accurate solution to the deformation problem using the FEM, a high-resolution mesh must be used in the simulation, usually involving several thousands nodes. For example in this thesis a FE mesh of 6000 nodes is used to represent the object resulting in a K matrix of size $18000*18000$. Solving such system of equations in real time (1-10miliseconds) using state-of-the-art conventional CPUs is impractical. For this purpose, we propose a hardware-based highly parallelized solver for the FE-based equations. This solver, find the solution to the sparse system of $Kx = b$ employs the CG method.

A critical decision in hardware implementation of the CG algorithm is the choice

between fixed-point and floating point computing. Using floating point representation of numbers will in a high hardware cost, limiting the achievable parallelism. On the other hand fixed-point implementation will reduce the dynamic range of values' representation which can be problematic specially for iterative methods by increasing relative error. This can result in numerical instability. We utilize a custom fixed-point implementation of the CG algorithm proposed by mafi et.al in [1]. More details about our fixed-point implementation and it's numerical stability analysis is given in Chapter 4. In this approach customized fixed-point computing can greatly increase the parallelism without noticeable decrease in accuracy.

Another problem concerns the scalability of the utilized solver for the system of equations $Kx = b$. Accuracy of the FEM, to a large extend, depends on the number of nodes in FE mesh. For example, typical meshes for biological soft-tissue may involve several thousand nodes. On the other hand, due to insufficient resources available on any processor, non of the current available processors (CPUs, GPUs, FPGAs, etc.) are powerful enough to solve such big set of equations in expected time frame.

Taking these into account, utilization of multiple-processors for solving such system of equations is inevitable. The scalability of the hardware architecture for the solver is important in two senses:

1. The hardware design utilized for doing the computations should be scalable on multiprocessors.

   Thus breaking down the computations involved such that it creates the minimum overlap among the processors both in data and timing sense is necessary. This means that there should be a minimum amount of data needed to

be shared between the processors for less memory usage on each processor and moreover to reduce the communication required among them. In addition minimizing the timing overhead requires that the processors should work such that none of them be idle at any time waiting for results from other processors.

2. The proposed architecture design should be scalable in a way that it can take advantage of newer FPGA devices that will have greater amount of on-chip memory and DSP resources

1.3 Thesis Contributions

The main premise of this work is that the steps involved in the FE simulation of object deformation can be classified as: (i) performing algorithmically complex but computationally inexpensive routines; (ii) solving a large linear system of equations. The latter can be delegated to a customized parallel-computing platform whereas the former can be simply executed on a conventional computer.

In [1], we developed an FPGA-based accelerator for solving a sparse system of equations using the iterative method of Conjugate Gradient [9]. In this thesis, the hardware solution will be generalized to a multiple-FPGA configuration with increased parallelism in order to solve larger FE problems using the Conjugate Gradient (CG) method.

It is worth mentioning that although this architecture has been developed mainly for modeling soft-tissues using linear models which result in equations in the form $Kx = b$, it can be also utilized for some non-linear deformation models by solving

an incremental form of such equations. This incremental form of these equations can be derived as described in [10].

The main contributions of this thesis are in:

- proposing a novel highly parallelized, scheme for fixed point CG implementation on an FPGA with a new sparse matrix by vector multiplication unit, vector by vector operation unit and a new memory architecture comparing to [1] .

  this design is scalable to multi processors with linear increase in resource usage by increase in number of FPGAs.

- Proposing a new storage method for storing a dense vector, pre-multiplied by a sparse matrix (Sparse matrix vector Indexing Scheme(SMVIS)) which dramatically decreases the memory usage while parallelizing the matrix by vector multiplication by performing multiplication for some rows at the same time.

- proposing a novel method for performing the CG method on multiple processors.

- proposing a novel architecture for implementing multi-processor CG method on multiple FPGAs. The implementation on 4 Stratix III Altera devices yields 302 Giga Operations per Second and a memory bandwidth of 7.257 TB/s while doing the matrix by vector multiplication.

- Designing a novel communication scheme for required inter-FPGA data exchanges.

1.4 Thesis Outline

The rest of this thesis is organized as follows:
In the Chapter 2 a brief review of the prior work is presented, Chapter 3 introduces the finite element method and formulations, Chapter 4 briefly discusses the fixed-point implementation of the Conjugate Gradient method. In Chapter 5 the algorithm for the expansion of the design on multiple FPGAs is described. Chapter 6 presents the proposed hardware architecture employed for carrying CG on multiple FPGAs. Chapter 7 covers the system performance and experimental results and Finally, the thesis is concluded in Chapter 8 where some possible directions for future research are also discussed.

1.5 Related Publication

- R. Mafi, S. Sirouspour, B. Moody, B. Mahdavikhah, K. Elizeh, A. Kinsman, N. Nicolici, M. Fotoohi and D. Madill, "Hardware-based Parallel Computing for Real-time Haptic Rendering of Deformable Objects" *IROS 2008. IEEE/RSJ International Conference on Intelligent Robots and Systems Volume , Issue , 22-26 Sept. 2008 Page(s):4187 - 4187, Digital Object Identifier 10.1109/IROS.2008.4651242.*

- Ramin Mafi, Shahin Sirouspour, Behzad Mahdavikhah, Brian Moody, Kaveh Elizeh, Adam Kinsman and Nicola Nicolici, "A Parallel Computing Platform

for Real-time Haptic Interaction with Deformable Bodies" submitted to the *IEEE Transactions on Haptics*(revised).

Chapter 2

Literature Review

This chapter presents an overview of related works on real-time modeling of soft tissue deformation. At first we will have a survey on different deformable body models described in the literature and as will be explained in we have chosen FEM based continuum mechanics modeling in our application. This would be followed by an introduction to different mathematical methods and computational platforms for real-time FEM based modeling of soft tissue deformation. The last section will review the efforts done for FEM based modeling of soft tissue by employing multi processors.

2.1 Deformable Object Modeling

In this section some of the most common methods used for deformable body modeling are briefly introduced and then we will focus on finite element based continuum mechanics based model, which is the core model used for our simulation.

2.1.1 Soft Object Modeling Methods

Deformable object models are used to find the new formation of a deformable body after applying external or internal forces on it. There has been a considerable research in modeling of real-time interaction with non-rigid deformable objects, e.g. see [11, 12]. Generally soft object modeling methods are divided into two groups, physical based models and non-physical models. Non-physical models are based on heuristic geometric techniques or use simplified physical principals to obtain an acceptable model of the tissue. [13, 14]. For more details the reader is referred to [15].

Two popular non-physics based models are:

- Spline modeling: In this approach, both planar and 3D curves and surfaces are represented by a set of control points. The deformation of the object is then defines as a function of these control points. By varying the positions of these points or adding or removing or changing the weight of some of these points the new formation of the complex objects would be determined [15,16]. As a result of inaccuracy of this model model due to it's non-physics base modeling, this model is no longer used [15].

- ChainMail modeling: In this approach, the soft tissue is represented with cubic lattices, where each of the cubs an move slightly with respect to its neighbors and the position of cubes are finally adjusted by minimizing the total potential energy. ChainMail has very low computational cost but it is not realistic and imposes severe constraints on what mesh topology may be used [17]. Moreover, simultaneous multiple contact points in this model

result in an extensive computational cost for determining the propagation of the imposed displacements on the elements [11]. Fig. 2.1.1 depicts the ChainMail model representation of a 2D object.

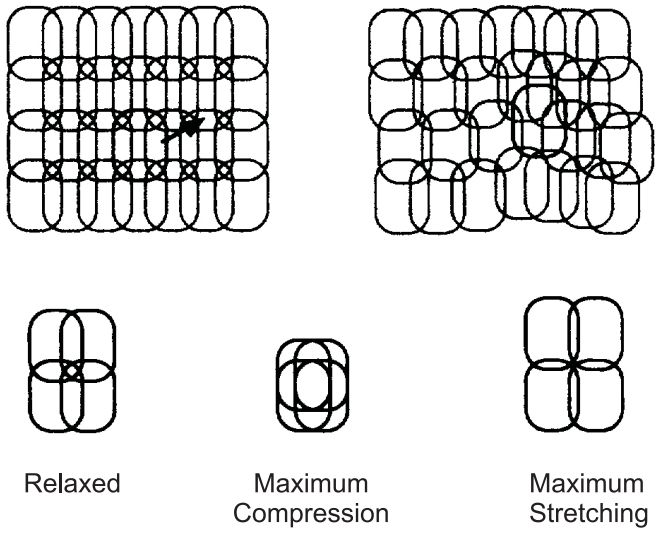

Figure 2.1: ChainMail model representation of 2D object before and after deformation *(Figure from Sarah F.F. Gibson [3]).*

The physical methods are based on solving the equations from physics principles or more specifically the theory of elasticity considering material properties. These models are computationally demanding because they require solving partial differential equations. Physically based models result in more accurate and realistic results. Terzopoulos [18], Waters [19] and Platt [20] showed the advantages of the physically-based models on previous computer animation techniques [21]. Physical based models can be divided into following categories:

- Mass Spring models
- Linear Green Function (GF) models
- Continuum mechanics-based models

Mass Spring Models

Mass Spring models represent the object with a mesh of nodal concentrated masses with a network of massless springs connecting the nodes together. Also each nodal mass is connected to its initial position with a damper element. Fig. 2.2 illustrates the mass-spring representation of a 2D rectangle. The elastic behavior of the springs can be changed to match physical properties of the material. The springs can have linear behavior, but non-linear or volumetric material properties may also be used [22, 23].

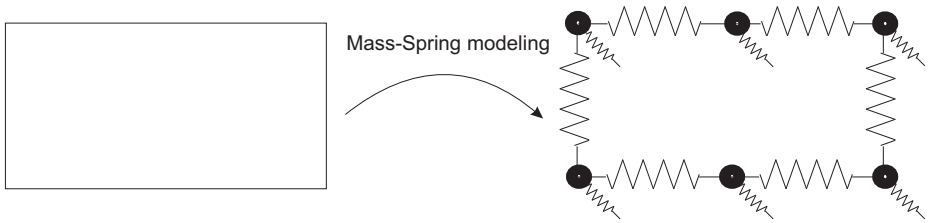

Figure 2.2: Mass spring model of a 2D rectangular shape

Therefore for node N of the mesh we can write the equation of motion:

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{b}\dot{\mathbf{x}} + \sum_{i} \mathbf{F_i} = \mathbf{F_{ext}} \tag{2.1}$$

where $\mathbf{x}$, $\ddot{\mathbf{x}}$ and $\dot{\mathbf{x}}$ are position, velocity and acceleration of the node respectively, $\mathbf{M}$ denotes the mass connected to node $\mathbf{N}$, $\mathbf{F_i}$ represents the internal force exerted to node $\mathbf{N}$ from node $\mathbf{i}$ which is connected to node $\mathbf{N}$ through a spring connector. Also, $\mathbf{F_{ext}}$ is the total external force applied to node $\mathbf{N}$. After assemblage and writing the equations for all of the nodes in a matrix format we will come up with Eq. 2.2 for the dynamic case which will reduce to Eq. 2.3 for a static case.

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{C}\dot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{f} \tag{2.2}$$

$$\mathbf{K}\mathbf{x} = \mathbf{f} \tag{2.3}$$

Here $\mathbf{M}$ and $\mathbf{C} \in \Re^{3n}$ are diagonal matrices for mass and damping and $\mathbf{K} \in \Re^{3n}$ is a sparse stiffness matrix.

Mass-spring model is the simplest and the least computationally intensive physical based method but it not necessarily accurate. Moreover, most of systems based on mass-spring models are not convergent [12].

That is, as the mesh is refined, the simulation does not converge on the true solution. Instead, the behavior of the model is dependent on the mesh resolution and topology. In practice, spring constants are often chosen arbitrarily, and one can say little quantitatively about the material being modeled.Utilizing learning algorithms to find model parameters is an interesting approach in mass-spring models. e.g. [24, 25] and references therein. Despite all efforts, the integration of realistic tissue properties into parameters of mass-spring model is not a trivial task. The construction of an optimal network of springs in 3D is a complicated process and particle systems can become oscillatory or even unstable under certain conditions [26].

Another deficiency of Mass-spring modeling is that large mass-spring-damper meshes can impede rapid global propagation of deformations resulting in a localized deformation [11].

Linear Green Function (GF) Models

One of the earliest approaches introduced in [27] uses Greens functions (GFs) and fast low-rank updates based on Capacitance Matrix Algorithms. GFs form a basis for describing all possible deformations of a linear elastic model. This method works based on a huge amount of pre-computations assuming a linear deformation model. It uses a look-up table for different deformations and applies superposition principle. This approach is not suitable for objects with nonlinear elastic models and for large number of nodes and contact nodes. [27].

Continuum Mechanics Based Models

Constitutive models based on continuum mechanics have been proposed to accurately model soft-object deformation [28]. Continuum mechanics provides a physics based-model for modeling soft object deformation. In continuum mechanics a material is known to have definite densities of mass, momentum, and energy in the mathematical sense. The mechanics of such a material continuum is continuum mechanics [29]. Therefore, for such an object the continuum model establishes a set of relationships between the shape of the body, constraints and internal deformations within this solid and the external forces applied to it. The behavior of the tissue would be governed by equations arising from continuum mechanics. Such modeling approach yields partial differential equations involving the deformation filed, applied forces to the object, and a set of boundary conditions. These equations are defined by physical principals such as the laws of conservation of mass, the balance of momenta and the balance of energy, as well as the constitutive equation that mathematically expresses the mechanical property of the body

material [30]. The continuum equations are obtained by computational methods for solving the continuum mechanics based equations of the body.

Some of the most common computational methods for finding solution to continuum based models are:

- **Finite Difference Method:** In this method continuous derivatives in PDEs are substituted by finite difference equations in related domain. As a result of this process, the PDEs will reduce to a set of algebraic equations [31]. The applications of this method are limited because the discretization of objects with irregular geometry becomes extremely dense. [14]

- **Boundry Element Method(BEM):** This approach transforms the integral of the motion equations for a volume in surface integrals using Green-Gauss theorem. Thus this method is considerably faster than other modeling methods since it will only deal with 2D equations. One drawback of this method is that it requires homogeneity of the object [12] which is not always the case in biological tissue material. Furthermore, it does not allow volumetric changes to the object which means it can not compute the displacement of any interior point. So it will be incapable of simulating medical operations such as cutting or suturing.

- **Finite Element Method(FEM):** This method can be used to discretize the model in the spatial domain in the absence of an analytical solution to such problems in most cases. This is achieved by partitioning the object into small elementary shapes and then derive the equations of motion for each element as a function of the mesh node displacements [32].

  FEM is applicable to solve PDEs in irregular grids. In this method the object

is divided into some elemental shapes e.g. triangles in 2D space or tetrahedrons in a 3D space, then the formulation for each element is done based upon the Principle of Minimum Potential Energy. After this step for each element the relation between stress, strain and nodal displacements has been determined. The next step in FEM is assemblage, in which the equations relating the external and internal forces and nodal displacements are derived for the whole object. Thus the continues problem in 3D spatial domain converts to a set of discrete set of unknown position equations [12], which can be solved more easily by numerical methods. However, the need to manipulate large matrices and solve large numbers of differential equations imposes performance difficulties to apply FEM to real time interactive haptic feedback applications. [33]. In [34] and [35] interpolation of forces is proposed for calculation of high rate haptic fedback from a low rate model. The other approach to speed up run-time simulation is to isolate some procedures of the real-time computation, precompute them and later combine precomputed results with states of a simplified run-time model [33].

2.2 Solvers for Linear Systems of Equations

For performing the matrix by vector operation on a CPU , the memory bandwidth limit would be the bottleneck, which lowers the performance of CPU down to 10-33% of their peak performance while doing matrix by vector operation [36]. The situation is even worse when the matrix is sparse [36].

In general, there has been a prevailing tendency in the research community towards using algorithmic software-based solutions for addressing the problem

of real-time object deformation simulation. Notable exceptions to this trend are a number of papers proposing parallel computing using multiple-CPU computer clusters [37] or Graphics Processing Units (GPU) [38].

CPU and GPU architectures are bounded by their maximum computational and memory bandwidth. Their efficiency mainly depends on the implemented operations. In the case of sparse matrix by vector operations, both computational and memory bandwidth efficiency are low in GPUs and CPUs [39].(computational band width refers to the measure of amount of operations a processor can do in the unit of time).

In the last few years, FPGAs have significantly advanced both in terms of speed and resources, i.e. the number of arithmetic units, programmable logic cells and embedded memories. Such solution offers a real advantage over networked parallel computers by providing enormous computation power in a compact and relatively inexpensive package. Compared with general-purpose GPUs, FPGA-based custom computing architectures can provide greater parallelism by tailoring the hardware computing unit to the problem at hand.

In [40], the authors have presented fast algorithms for the solution of large linear equation systems as they typically arise in finite element discretisations. They used both GPU and FPGA devices for implementing solvers for these systems, using Emulated and mixed precision solvers. The emulation utilizes two single float numbers to achieve higher precision by using lower precision processing elements, while the mixed precision iterative refinement computes residuals and updates the solution vector in double precision but solves the residual systems in single precision using CG or multigrid solver.

They show that implementation of the emulation technique on GPU is significantly slower for the PDE problem than the mixed precision iterative refinement on FPGA.

Elkurdi et al. [41] presented an architecture and implementation of an FPGA-based sparse matrix by vector multiplier (SMVM) for use in the iterative solution of large, sparse systems of equations arising from FEM applications, their architecture benefits from a hardware-oriented matrix striping algorithm utilizing the matrix structure. The implemented SMVM-pipeline prototype contains 8 Processing elements(PEs) and is clocked at 110 MHz obtaining a peak performance of 1.76 GFLOPS. For 8 GB/s of memory bandwidth typical of recent FPGA systems, this architecture can achieve 1.5 GFLOPS sustained performance. Mafi [42] proposed a highly parallelized hardware implementation of CG algorithm. In that design they utilized a novel fixed point method for representation of floating point numbers with static and dynamic scaling to mitigate quantization errors. This design works on a single FPGA and achieves 18 GOPs for implementation on a Stratix II EP2S60.

It is worth mentioning that hardware-based solutions for sparse matrix by vector multiplication and for solving linear systems of equations have been discussed in a few previous papers, e.g. see [43, 44]. These approaches, which use floating point operations, are rather abstract and cannot be scaled to solve practical problems using existing FPGA devices. Floating-point implementations have a high hardware cost, severely limiting the parallelism and thus the performance of the hardware accelerator. In contrast, the proposed hardware architecture reaches a

good tradeoff among error, calculation time and hardware cost by using a novel, modified type of fixed point operations and utilizing fixed-point computing units. A key novelty of the proposed solution is in its ability to continuously supply data operands to a large number of computing units within a scalable architecture. The hardware solver is largely independent of the FE mesh configuration and can be scaled up based on available FPGA resources to solve problems of larger size.

Due to the properties of computation intensiveness and computation locality , it is very attractive to implement FEM on multi processors. For this objective two general approaches can be considered. One is exploiting domain decomposition methods to subdivide the physical domain into smaller regions or subdividing the large linear systems into smaller subsystems whose solution can be used to produce a pre-conditioner for the system of equations that results from discretizing the PDE on the entire domain. [45]. The second is to apply multi processing methods for solving equations from the single level equations, that is performing the computations involved in single level equations on multiple processors.
Generally all domain decomposition methods, require iteration on solution to system of linear equations on each sub-domain in order to find the result for interface nodes, located between two sub-partitions. Aside from convergence issues, this makes them slower comparing to the second method which only requires one time of solving a set of linear equations.

Another significant disadvantage of the domain decomposition methods is that the mesh levels must be generated offline. This prohibits, for example, local mesh refinement when tissue is cut. [46]

Ulrike et al. [47] performed the parallel implementation of CG algorithm on Cedar(4 multiple processor clusters which are connected through an inter connection network). They achieved a speed of 38 milliseconds for 1 CG iteration for n=255, where n is the dimension of matrix. The performance is worse for a multi processor architecture with 16 processors working in parallel which diminishing to 5% of the peak performance of whole system. [36]

A floating point implementation of sparse matrix by vector multiplication on multi FPGAs has also been proposed by delorimier et al. [36], in this design they project 1.5 double precision Gflops/FPGA for a single VirtexII-6000-4 and 12 double precision Gflops for 16 Virtex IIs (750Mflops/FPGA).

In this thesis, a parallel computing platform is proposed that employs multiple Field Programable Gate Array (FPGA) devices to greatly speed up the calculations in the FE-based deformation analysis. In the proposed method a customized fixed-point operation proposed by Mafi et al. in [48].
The implementation on 4 Stratix III Altera devices achieves a peak performance of 302 GOPs(giga operations per second) and a memory bandwidth of 7.257 TB/s while doing the matrix by vector multiplication.

Chapter 3

Finite Element Modeling

As discussed in Chapter 2, for modeling the soft-object deformations, we use Finite Element Method to obtain physical modeling of the object based on continuum mechanics differential equations. In this chapter the formulation of the Finite Element Method will be presented for both static and dynamic simulations. In our haptic simulator, the user interacts with a virtual object using a force feed-back haptic interface. The simulation uses a model of interaction to calculate the object motion/deformation as well as the interaction force. In an impedance-type simulation, the haptic device imposes its displacement on the virtual object at the contact point and in response the deformation of the entire body and interaction force are computed by the model.

In finite element method we need to take some steps for discretizing the PDEs governing the object behavior to some linear equations coming from assemblage of the equations for elemental shapes constituting the object. The steps (quoted from [49]) are as follows :

1. ”The continuum is separated by imaginary lines or surfaces into a number of ’finite elements’.

2. The elements are assumed to be interconnected at a discrete number of nodal points situated on their boundaries. The displacements of these nodal points will be the basic unknown parameters of the problem.

3. A set of functions is chosen to uniquely define the state of displacement within each ’finite element’ in terms of its nodal displacements.

4. The displacement functions now uniquely define the state of strain within an element in terms of the nodal displacements. These strains together with nodal initial strains and the constitutive properties of the material, will define the state of stress throughout the element and on the boundaries.

5. A system of forces concentrated at the nodes and equilibrating the boundary stresses and any distributed loads is determined, resulting in a stiffness relationship of the form of equation 3.1.

$$\mathbf{Kx} = \mathbf{f} \tag{3.1}$$

This approach is also known as Displacement formulation [50, 51].
One of the most important and challenging steps in finite element analysis is the derivation of the finite element characteristics [52]. In stress and structural analysis, the finite element characteristics can be derived by applying the principle of minimum potential energy.
This approach will result in the **stiffness** (or displacement) method, wherein the

primary unknowns are the nodal displacements.

The material used in this section are mostly borrowed from [28, 32, 52] and the reader is referred to them for more details.

3.1 Basic Elasticity Concepts

Generally the term **stress** in elasticity theory refers to the force per unit area. The term **strain** refers to elongation per unit length [52]. An elastic object will return to its natural shape (the formation it had before applying the force on it) after releasing the force being exerted on it, if the applied stress would have been less than its elastic limit. The elastic objects are divided into two types, linear elastic and non-linear elastic materials. Linear elastic materials have a stress-strain characteristic as shown in Fig. 3.1 and some non-linear objects have a stress-strain characteristic as shown in Fig. 3.2. In the following sections first modeling for static linear elastic models would be presented and then it will be expanded to dynamic modeling of linear objects.

Kinematic Relations

The modeled tissue will be defined in 3D spatial domain as Ω. Ω consists of the particles with positions $\mathbf{x} = [\mathbf{xyz}]$ while the tissue is not deformed. Ω is made by two partitions, Γ_1 and Γ_2, where the nodes which are in Γ_1 are constraint nodes, that is before and after deformation their location does not change but nodes in Γ_2 will be displaced when the object is under stress. These displacements are referred as $\mathbf{u} = [\mathbf{uvw}]$. Therefore, the new position of the particle **x** after deformation would

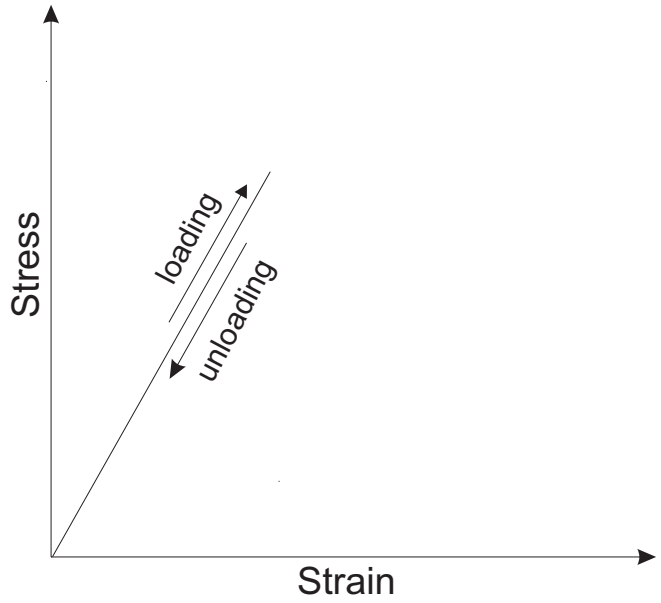

Figure 3.1: Strain-stress characteristics of a linear elastic material

be $\mathbf{x} + \mathbf{u}$.

Fig. 3.3 illustrates a 2D object and its partitions in xy plane. In this figure s corresponds to surface traction forces, b represents body force and $\mathbf{f_p}$ shows the external forces applied to the object. For small deformations stress and strain are related to each other as:

$$\begin{aligned} \epsilon_{xx} = \frac{\partial u}{\partial x} \;\; \epsilon_{yy} = \frac{\partial v}{\partial y} \;\; \epsilon_{zz} = \frac{\partial w}{\partial z} \\ \epsilon_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \epsilon_{yz} = \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \\ \epsilon_{zx} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \end{aligned} \tag{3.2}$$

Therefore, the relationship between the strain vector $\epsilon = [\epsilon_{xx} \; \epsilon_{yy} \; \epsilon_{zz} \; \epsilon_{xy} \; \epsilon_{yz} \; \epsilon_{zx}]^T$

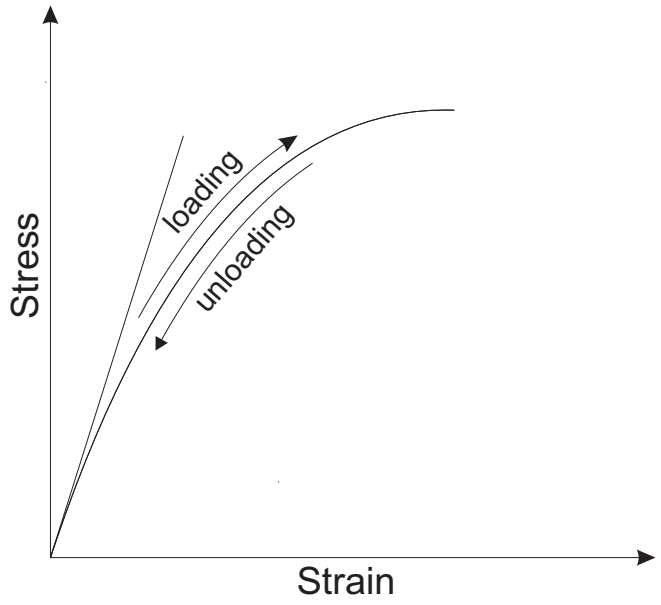

Figure 3.2: Strain-stress characteristics of a non-linear elastic material

and the displacement vector $\mathbf{u}^T = [\ u\ v\ w]$ can be expressed as in Equation 3.3, where the $\mathbf{L}$ is known as the strain-displacement matrix.

$\mathbf{L}$ matrix is a linear operator and is derived as in Equation 3.4.

$$\epsilon = \mathbf{L}\mathbf{u} \tag{3.3}$$

$$\mathbf{L} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \end{bmatrix} \tag{3.4}$$

Hook's law states that the normal stress σ is proportional to normal strain ϵ in a

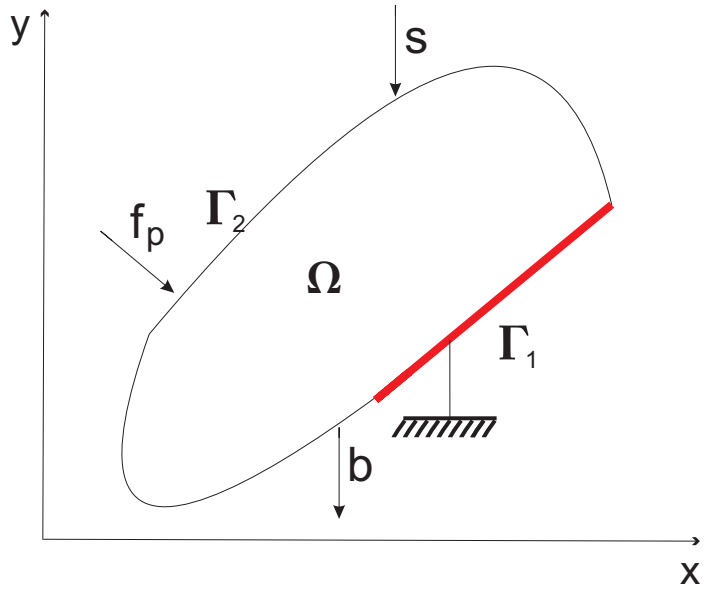

Figure 3.3: A 2D object with external possible forces applied to it. In this figure, s represents the surface traction b is the body force fp represents the point loads Γ1and Γ2 refer to constraint and non-constraint nodes of domainΩ.

uniaxial state of stress, or :

$$\sigma = E\epsilon \tag{3.5}$$

where **E** is known as Young's modulus, the elastic modulus or the modulus of elasticity. Hook's law in this form is of rather limited use. A much more general version of hook's law is given by Equation 3.6 [49,52].

$$\sigma = D(\epsilon - \epsilon_0) + \sigma_0 \tag{3.6}$$

where **D** is referred to as the material property matrix, σ the stress vector, ϵ the strain vector, ϵ_0 the self-strain vector, and σ_0 the initial or residual stress vector. In 3D the ϵ is defined as before and ϵ_0, σ, σ_0 are defined as in Equation 3.7 and the **D** for a linear elastic isotropic material is given by [49] as in Equation 3.8. In this

equation the μ represents poisson's ratio. In addition the material property matrix **D** is symmetric [52].

$$\begin{aligned} \epsilon_0 &= \left[\epsilon_{xx0}\ \epsilon_{yy0}\ \epsilon_{zz0}\ \epsilon_{xy0}\ \epsilon_{yz0}\ \epsilon_{zx0}\right]^T \\ \sigma &= \left[\sigma_{xx}\ \sigma_{yy}\ \sigma_{zz}\ \sigma_{xy}\ \sigma_{yz}\ \sigma_{zx}\right]^T \\ \sigma_0 &= \left[\sigma_{xx0}\ \sigma_{yy0}\ \sigma_{zz0}\ \sigma_{xy0}\ \sigma_{yz0}\ \sigma_{zx0}\right]^T \end{aligned} \tag{3.7}$$

$$D = \frac{E}{(1+\mu)(1-2\mu)} \begin{pmatrix} 1-\mu & \mu & \mu & 0 & 0 & 0 \\ \mu & 1-\mu & \mu & 0 & 0 & 0 \\ \mu & \mu & 1-\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\mu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\mu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\mu}{2} \end{pmatrix} \tag{3.8}$$

The initial vector σ_0 represents pre-stresses that are known to exist in a material before it is loaded. The FEM or any other method can not predict these parameters and they must be determined by analyst. The self-strain vector ϵ_0 may be a result of crystal growth, shrinkage or most common temperature changes.

3.2 Principle of Minimum Potential Energy

The total potential energy (Π) is defined to be the sum of internal potential energy, the strain energy (U_i), and the external potential energy from external forces (U_e) [52], or:

$$\Pi = U_i + U_e \tag{3.9}$$

For conservative systems the loss in the external potential energy is equal with the work done by external forces.

$$U_e = -W_e \;\; \Rightarrow \;\; \Pi = U_i - W_e \tag{3.10}$$

The principle of minimum total potential energy states that Π must be minimum for stable equilibrium, therefore the first variation of the total potential energy (as a result of variation in displacement) should be zero. That is,

$$\delta\Pi = \delta U_i - \delta W_e = 0 \;\; \Rightarrow \;\; \delta U_i = \delta W_e \tag{3.11}$$

Moreover, using the matrix notation form for strain and stress vector for the strain energy we will have:

$$\delta U_i = \int_V (\delta\epsilon)^T \sigma dV \tag{3.12}$$

As shown in Fig.3.3, generally three types of external forces are applied to the body, s which represents the surface traction e.g. the hydrostatic pressure (force per unit area) on the water filled side of a dam, the body force $\mathbf{b}$ (per unit volume) and up to N point loads $\mathbf{f_p}$. The first variation of the work done by these three external forces is

$$\delta W_e = \int_V (\delta\mathbf{u})^T b dV + \int_S (\delta\mathbf{u})^T \mathbf{s} dS + \sum_{p=1}^{N} (\delta\mathbf{u})^T \mathbf{f}_p \tag{3.13}$$

where the body force, $\mathbf{b}$, the surface traction vector, $\mathbf{s}$, the point load $\mathbf{f}_p$ and the $(\delta\mathbf{u})^T$ are given by Equation 3.14.

$$\begin{aligned}\mathbf{b} &= [b_x b_y b_z]^T \\ \mathbf{s} &= [s_x s_y s_z]^T \\ \mathbf{f}_p &= [f_{px} f_{py} f_{pz}]^T \\ \delta\mathbf{u}^T &= [\delta u\ \delta v\ \delta w]^T\end{aligned} \tag{3.14}$$

Equations 3.6, 3.11, 3.12, 3.13 will result in

$$\int_V (\delta\epsilon)^T \sigma dV = \int_V (\delta\mathbf{u})^T \mathbf{b} dV + \int_S (\delta u)^T \mathbf{s} dS + \sum_{p=1}^{N} (\delta\mathbf{u})^T \mathbf{f}_p \tag{3.15}$$

$$\begin{aligned}\int_V (\delta\epsilon)^T \mathbf{D}\epsilon dV = \int_V (\delta\epsilon)^T \mathbf{D}\epsilon_0 dV - \int_V (\delta\epsilon)^T \sigma_0 dV + \int_V (\delta\mathbf{u})^T \mathbf{b} dV \\ + \int_S (\delta u)^T \mathbf{s} dS + \sum_{p=1}^{N} (\delta\mathbf{u})^T \mathbf{f}_p\end{aligned} \tag{3.16}$$

Now by discretizing the region into some elemental shapes as in Fig. 3.2, the integrations for the object can be interpreted

$$\begin{aligned}\int_V (\delta\epsilon)^T \mathbf{D}\sigma dV &= \sum_{e=1}^{M} \int_{V^e} (\delta\epsilon)^T \mathbf{D}\sigma dV \\ \int_S (\delta u)^T \mathbf{s} dS &= \sum_{e=1}^{M} \int_{S^e} (\delta u)^T \mathbf{s} dS +\end{aligned} \tag{3.17}$$

where $\mathbf{S}^e$ and $\mathbf{V}^e$ denote the surface and volume for an element and M is the total number of elements. Now we can write Equation 3.15 for each of the finite elements as in

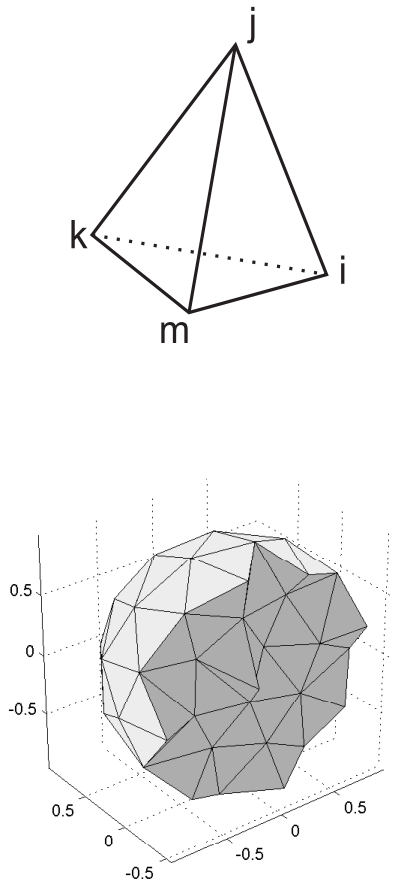

Figure 3.4: Discretizing a sphere into tetrahedron elemental shapes

$$\int_{V^e} (\delta\epsilon)^T \mathbf{D}\sigma dV = \int_{V^e} (\delta\mathbf{u})^T \mathbf{b} dV + \int_{S^e} (\delta u)^T \mathbf{s} dS + \sum (\delta\mathbf{u})^T \mathbf{f}_p \qquad (3.18)$$

Here $\mathbf{f_p}$ are the external point loads applied to each node of the element.

For a 3D elemental shape e.g the tetrahedron in Fig. 3.4, we can relate the x, y and z components of the displacement vector $\mathbf{u}$ of an element to the x, y and z components of the displacement vector for each of its nodes using the shape functions $\mathbf{N}$ as follows

$$
\begin{aligned}
u &= N_i(x,y,z)u_i + N_j(x,y,z)u_j + N_k(x,y,z)u_k + N_m(x,y,z)v_m \\
v &= N_i(x,y,z)v_i + N_j(x,y,z)v_j + N_k(x,y,z)v_k + N_m(x,y,z)v_m \\
w &= N_i(x,y,z)w_i + N_j(x,y,z)w_j + N_k(x,y,z)w_k + N_m(x,y,z)v_m
\end{aligned} \tag{3.19}
$$

where u_i, u_j, u_k, u_m are the x components of nodal displacements and in a similar way elements with v and w in their names are associated with y and z components. Moreover $\mathbf{N}_i$, $\mathbf{N}_j$, $\mathbf{N}_k$ and $\mathbf{N}_m$are the four shape functions. Now if we define $\mathbf{u}$ and $\mathbf{a}^e$ as in Equation 3.20, we can obtain 3.21, with shape function $\mathbf{N}$ defined in Equation 3.22.

$$
\begin{aligned}
\mathbf{u} &= [u\ v\ w]^T \\
\mathbf{a}^e &= [u_i v_i w_i\ u_j v_j w_j\ u_k v_k w_k\ u_m v_m w_m]^T
\end{aligned} \tag{3.20}
$$

$$
\mathbf{u} = \mathbf{N}\mathbf{a}^e \tag{3.21}
$$

$$
\mathbf{N} = \begin{pmatrix}
N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m & 0 & 0 \\
0 & N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m & 0 \\
0 & 0 & N_i & 0 & 0 & N_j & 0 & 0 & N_k & 0 & 0 & N_m
\end{pmatrix} \tag{3.22}
$$

combining Equations 3.3 and 3.21, we can define matrix $\mathbf{B} = \mathbf{LN}$ as the *strain-nodal displacement matrix* and therefore:

$$\epsilon = \mathbf{B}\mathbf{a}^e \tag{3.23}$$

L is a linear operator and if the shape function **N** only contains linear functions (in x y and z) then the **B** matrix contains constant elements. Thus

$$\delta\epsilon = \delta(\mathbf{B}\mathbf{a}^e) = \mathbf{B}\delta\mathbf{a}^e \tag{3.24}$$

and

$$(\delta\epsilon)^T = (\mathbf{B}\delta\mathbf{a}^e)^T = (\delta\mathbf{a}^e)^T\mathbf{B}^T \tag{3.25}$$

More over for $(\delta\mathbf{u})^T$ we will have:

$$(\delta\mathbf{u})^T = (\mathbf{N}\delta\mathbf{a}^e)^T = (\delta\mathbf{a}^e)^T\mathbf{N}^T \tag{3.26}$$

By substituting Equations 3.25 and 3.26 in 3.18 we can get:

$$(\delta\mathbf{a}^e)^T)\{\int_{V^e}(\mathbf{B}^T\sigma dV) - \int_{V^e}(\mathbf{N})^T\mathbf{b}dV - \int_{S^e}(\mathbf{N})^T\mathbf{s}dS - \sum(\mathbf{N})^T\mathbf{f}_p\} = 0 \tag{3.27}$$

The term $\delta\mathbf{a}^e$ represents the first variation of the nodal displacements which must satisfy the displacement boundary conditions, therefore in any event $\delta\mathbf{a}^e$ is non-zero [52]. So it can be concluded that in general

$$\int_{V^e}(\mathbf{B}^T\sigma dV) - \int_{V^e}(\mathbf{N})^T\mathbf{b}dV - \int_{S^e}(\mathbf{N})^T\mathbf{s}dS - \sum(\mathbf{N})^T\mathbf{f}_p = 0 \tag{3.28}$$

Therefore, for a linear elastic material for which Equation 3.6 applies, considering Equation 3.28, we can write:

$$\{\int_V \mathbf{B}^T\mathbf{D}\mathbf{B}dV\}\mathbf{a}^e - \int_V \mathbf{B}^T\mathbf{D}\epsilon_0 dV + \int_V \mathbf{B}^T\sigma_0 dV - \int_V \mathbf{N}^T\mathbf{b}dV - \int_S \mathbf{N}^T\mathbf{s}dS - \sum \mathbf{N}^T\mathbf{f}_p = 0 \tag{3.29}$$

Finally, this equation can be written in the form

$$\mathbf{K}^e\mathbf{a}^e = \mathbf{f}^e \tag{3.30}$$

where

$$\mathbf{f}^e = \mathbf{f}^e_{\epsilon 0} - \mathbf{f}^e_{\sigma 0} + \mathbf{f}^e_b + \mathbf{f}^e_s + \mathbf{f}^e_{PL} \tag{3.31}$$

and matrix **K** is

$$\mathbf{K}^e = \int_V \mathbf{B}^T\mathbf{D}\mathbf{B}dV = \mathbf{B}^T\mathbf{D}\mathbf{B}V \tag{3.32}$$

V is the volume of the element which for the tetrahedron depicted in Fig. 3.2 is given by Equation 3.33.

$$V = \frac{1}{6}det\begin{pmatrix} 1 & x_i & y_i & z_i \\ 1 & x_j & y_j & z_j \\ 1 & x_k & y_k & z_k \\ 1 & x_m & y_m & z_m \end{pmatrix} \tag{3.33}$$

The five elemental force vectors are given by

$$\begin{aligned}
\mathbf{f}^e_{\epsilon 0} &= \int_V \mathbf{B}^T \mathbf{D} \epsilon_0 dV \\
\mathbf{f}^e_{\sigma 0} &= \int_V \mathbf{B}^T \sigma_0 dV \\
\mathbf{f}^e_b &= \int_V \mathbf{N}^T \mathbf{b} dV \\
\mathbf{f}^e_s &= \int_S \mathbf{N}^T \mathbf{s} dS \\
\mathbf{f}^e_{PL} &= \sum \mathbf{N}^T \mathbf{f}_p
\end{aligned} \tag{3.34}$$

$\mathbf{K}^e$ is called the *elemental stiffness matrix*. Now the only remaining step is the assemblage of the global equations based on the elemental equations. For the global stiffness and force matrix we have:

$$\begin{aligned}
\mathbf{K} &= \sum global(\mathbf{K}^e) \\
\mathbf{f} &= \sum global(\mathbf{f}^e)
\end{aligned} \tag{3.35}$$

where global() is a mapping function from element node numbers to global node numbers. For the tetrahedral elements like the one shown in Fig. 3.2, the global stiffness matrix can be obtained through following two steps as in Equations 3.36 and 3.37.

$$\mathbf{K^e} = \begin{pmatrix} \mathbf{K^e_{i,i}} & \mathbf{K^e_{i,j}} & \mathbf{K^e_{i,k}} & \mathbf{K^e_{i,m}} \\ \mathbf{K^e_{j,i}} & \mathbf{K^e_{j,j}} & \mathbf{K^e_{j,k}} & \mathbf{K^e_{j,m}} \\ \mathbf{K^e_{k,i}} & \mathbf{K^e_{k,j}} & \mathbf{K^e_{k,k}} & \mathbf{K^e_{k,m}} \\ \mathbf{K^e_{m,i}} & \mathbf{K^e_{m,j}} & \mathbf{K^e_{m,k}} & \mathbf{K^e_{m,m}} \end{pmatrix} \tag{3.36}$$

Using the elemental stiffnes matrices, we can assemble the global stiffness matrix

as 3.37.

$$\mathbf{K} = \begin{pmatrix} \mathbf{K_{1,1}} & \mathbf{K_{1,2}} & \mathbf{K_{1,3}} & \cdots & \mathbf{K_{1,N}} \\ \mathbf{K_{2,1}} & \mathbf{K_{2,2}} & \mathbf{K_{2,3}} & \cdots & \mathbf{K_{2,N}} \\ \mathbf{K_{3,1}} & \mathbf{K_{3,2}} & \mathbf{K_{3,3}} & \cdots & \mathbf{K_{3,N}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{K_{N,1}} & \mathbf{K_{N,2}} & \mathbf{K_{N,3}} & \cdots & \mathbf{K_{N,N}} \end{pmatrix} \tag{3.37}$$

$$\mathbf{K_{I,J}} = \sum_{e=1}^{M} \mathbf{K}^{e}_{I,J} \quad I = 1:N, J = 1:N$$

where M is the maximum number of elements and N is the maximum number of nodes in these equations. $K^e_{I,J}$ will be assumed a null matrix if element e does not consist nodes I and J. It should be noted that I and J indices here refer to node numbers in global node numbering.

3.3 Applying Constraints

Without the substitution of a minimum number of constraint nodes to prevent rigid body movements of the structure, it is impossible to solve this system. The reason is that the displacements cannot be uniquely determined by the forces in such a situation. This obvious physical fact will interpret as singularity of **K** matrix in mathematics aspect [49]. Therefore, some of the nodes need to be fixed to predetermined positions. To this end, the global stiffness matrix has to change. The change is applied by making the corresponding rows and columns to that node zeros and making the diagonal elements one. Then the force vector should

be modified to reflect the change in stiffness matrix, only the elements of force vector corresponding to the fixed nodes will not change. For example assume that we want to make N^{th} node fixed, that is the displacement vector for $\mathbf{u}^N = [u_N v_N w_N]$ will be set to zero, in the global stiffness matrix all the elements in the rows and columns with indexes $3N-2$, $3N-1$, $3N$ should be set to zero except the diagonal elements which will be set to ones, Equation 3.38 illustrates the changes in K matrix and f vector after applying changes to row and column $3N$.

$$\mathbf{K} = \begin{pmatrix} k_{1,1} & \cdots & k_{1,3N-1} & 0 \\ \vdots & \vdots & \vdots & \vdots \\ k_{3N-1,1} & \cdots & k_{3N-1,3N-1} & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix} \mathbf{u} = \mathbf{f} = \begin{pmatrix} f_1 - w_N^0 k_{1,3N} \\ \vdots \\ f_{3N-1} - w_N^0 k_{3N-1,3N} \\ w_N^0 \end{pmatrix} \tag{3.38}$$

3.4 Dynamic FE Modeling

In static analysis of FE, we came up with the equation $\mathbf{Ku} = \mathbf{f}$, hence we neglected the terms in force resulting from the acceleration and velocity while the general formula for obtaining the force is:

$$\mathbf{M\ddot{u}} + \mathbf{C\dot{u}} + \mathbf{Ku} = \mathbf{R} \tag{3.39}$$

Where **M, C** and **K** are mass, damping and stiffness matrices respectively and **R** is the external load vector defined as

$$\mathbf{R}(t) = \mathbf{F}_I(t) + \mathbf{F}_D(t) + \mathbf{F}_E(t) \tag{3.40}$$

Where $\mathbf{F}_I(t)$ corresponds to inertia forces, $\mathbf{F}_I(t) = \mathbf{M}\ddot{u}$, $\mathbf{F}_D(t)$ represents the damping forces $\mathbf{F}_D(t) = \mathbf{C}\dot{u}$ and $\mathbf{F}_E(t)$ are the elastic forces, $\mathbf{F}_E(t) = \mathbf{K}u$. [32]
the decision that which of the dynamic or static models should be used is an engineering judgement because dynamic analysis would increase the computational effort needed to solve the equations. Equation 3.39 is a second order differential equation with constant coefficients [53, 54].
In the following we will describe the *Newmark* direct integration method which we have applied in our dynamic simulation.

The Newmark Method

In the Newmark integration scheme the following assumptions are used:

$$\begin{aligned} \dot{\mathbf{u}}^{t+\Delta t} &= \dot{\mathbf{u}}^t + [(1-\delta)\ddot{\mathbf{u}}^t + \delta\ddot{\mathbf{u}}^{t+\Delta t}]\Delta t \\ \mathbf{u}^{t+\Delta t} &= \mathbf{u}^t + \dot{\mathbf{u}}^t\Delta t + [(\frac{1}{2}-\alpha)\ddot{\mathbf{u}}^t + \alpha^{t+\Delta t}\ddot{\mathbf{u}}^t]\Delta t^2 \end{aligned} \tag{3.41}$$

Where α and δ are parameters that can be determined to obtain stability and accuracy. In general these parameters should satisfy the conditions [32]:

$$\begin{aligned} \delta &\geqslant 0.5 \\ \alpha &\geqslant 0.25(0.5+\delta)^2 \end{aligned} \tag{3.42}$$

Newmark originally proposed the trapezoidal rule for unconditionally stable scheme with $\delta = 0.5$ and $\alpha = 0.25$.
Also Δt in the simulation should not be bigger than a certain value Δt_{cr} in order to maintain stability. This value is determined by the characteristics of the worst

element in the mesh.

For solving the dynamic equation after finding the **K, M** and **C** matrices and determining parameter values we form $\hat{\mathbf{K}} = \mathbf{K} + a_0\mathbf{M} + a_1\mathbf{C}$ then at each step of the simulation we have to do the following tasks [32]:

1. calculate effective loads at time $t + \Delta t$

$$\hat{\mathbf{R}}^{t+\Delta t} = \mathbf{R}^{t+\Delta t} + \mathbf{M}(a_0\mathbf{u}^t + a_2\dot{\mathbf{u}}^t + a_3\ddot{\mathbf{u}}^t) + \mathbf{C}(a_1\mathbf{u}^t + a_4\dot{\mathbf{u}}^t + a_5\ddot{\mathbf{u}}^t) \tag{3.43}$$

2. solve displacements at time $t + \Delta t$:
   $\hat{\mathbf{K}}\mathbf{u} = \hat{\mathbf{R}}^{t+\Delta t}$

3. calculate accelerations and velocities at time $t + \Delta t$

$$\begin{aligned} \ddot{\mathbf{u}}^{t+\Delta t} &= a_0(\mathbf{u}^{t+\Delta t}) + a_2\dot{\mathbf{u}}^t + a_3\ddot{\mathbf{u}}^t \\ \dot{\mathbf{u}}^{t+\Delta t} &= \dot{\mathbf{u}}^t + a_6\ddot{\mathbf{u}}^t + a_7\ddot{\mathbf{u}}^{t+\Delta t}) \end{aligned} \tag{3.44}$$

Where a_0 to a_7 are defined as:

$$\begin{aligned} a_0 = \frac{1}{\alpha\Delta t^2} \quad a_1 = \frac{\delta}{\alpha\Delta t} \quad a_2 = \frac{1}{\alpha\Delta t} \quad a_3 = \frac{1}{2\alpha} - 1 \\ a_4 = \frac{\delta}{\alpha} - 1 \quad a_5 = \frac{\Delta t}{2}(\frac{\delta}{\alpha} - 2) \quad a_6 = \Delta t(1-\delta) \quad a_7 = \delta\Delta t \end{aligned} \tag{3.45}$$

Chapter 4

Conjugate Gradient Algorithm

As shown in Chapter 3, the application of the FEM to a linear elastic deformation model will result in a large but sparse system of equations in the form of:

$$\mathbf{Ku} = \mathbf{b} \tag{4.1}$$

In this chapter we will discuss different solvers for such equations and will review our fixed point implementation of CG algorithm which is based on the method proposed by Mafi et al. in [48].

4.1 Algorithms for Solving $\mathbf{Kx} = \mathbf{b}$

For solving linear systems of equations, in principle, there are two choices: direct methods and iterative methods, each having some advantages over the other depending on the matrix type and application of interest, several direct and iterative

methods have been developed for solving large and sparse systems of equations [55, 56]. One of the main advantages of iterative methods in the case of sparse systems is their lower memory consumption since only the non-zero elements of the matrix need to be stored. This reduces the usage of memory resources from an $O(N^2)$ to $O(N)$, where N is the size of the matrix. However this is not necessarily the case in direct methods. For example, direct methods based on matrix factorization can lead to the decomposition of a sparse matrix into new matrices which are not as sparse as the original matrix. Furthermore, iterative methods lend themselves for hardware-based parallelization better than direct methods [55].

It should also be noted that in haptics applications, even when a linear elastic model is used, matrix $\mathbf{K}$ can change depending on the contact node [1]. Moreover, nonlinear FE modeling of deformation can lead to a matrix $\mathbf{K}$ which would be dependent on the deformation $\mathbf{X}$. Therefore, this precludes the possibility of performing off-line calculations to obtain the inverse matrix or its factorization. Among the the iterative techniques, the Conjugate Gradient (CG) method is probably the most suitable method for our application. The system of equations generated by the FEM is a sparse, symmetric and positive definite as required by the original CG algorithm . From a numerical perspective, the CG method is usually more robust and less computationally intensive than some other more general iterative methods such as BiCGStab or GMRES [55, 57].

4.2 Fixed-Point Implementation of CG on Multiple FPGAs

In [1], we proposed an adaptive fixed-point implementation of the Conjugate Gradient (CG). Compared to floating-point operations, fixed-point calculations use significantly less hardware resources allowing for much greater parallelization of the computations. Static and dynamic scalings were proposed to reduce inaccuracies due to quantization errors and to avoid data overflows.

In this thesis the same method has been applied for the CG implementation with increased resolution for vectors and scalars which results in more accuracy of results. A pseudo-code of the CG algorithm for solving $\mathbf{Kx} = \mathbf{b}$ for $\mathbf{x}$ is given below. Here $\#m$ is the maximum number of iterations and ε is the error tolerance. Theoretically, getting the results from the CG algorithm requires $m = lengthofbvector$, but in our simulation we could obtain acceptable results in less CG iterations as will be discussed in this section.

1. $\mathbf{x} = \mathbf{init}$; %initial guess for solution of Kx=b
2. $\mathbf{b} = \mathbf{b}/\mathbf{Static\_factor}$; %static scaling for b vector
3. $\mathbf{r} = \mathbf{b} - \mathbf{K} * \mathbf{x}$; %residue
4. $\mathbf{d} = \mathbf{r}$; %initial "search direction"
5. $cntr = 1$;
6. $rr = \mathbf{r}' * \mathbf{r}$;
7. $rr0 = rr$;

8. $dyn\_factor = 0$;
9. $dyn\_flag = 0$;
10. while $(rr > \varepsilon^2 * rr0 \quad \& \quad cntr < \#m)$
11. if (rr $<$ dynamic scaling threshold) dyn_flag=1, dyn_factor=dyn_factor+1; % applying dynamic scaling
12. α = static_factor * rr / $(\mathbf{d}' * \mathbf{K} * \mathbf{d})$;
13. α = $\alpha <<$ dyn_flag; %dynamic scaling effect on α
14. $\mathbf{x} = \mathbf{x} + (\alpha * \mathbf{d}) >> dyn\_factor$; %update approximate solution, applying dynamic scaling
15. $\mathbf{rn} = \mathbf{r} - \alpha / static\_factor * \mathbf{K} * \mathbf{d}$; %update the residue
16. $rrn = \mathbf{rn}' * \mathbf{rn}$;
17. $\beta = (rrn/rr) >> (2 * dyn\_flag)$;
18. $\mathbf{d} = \mathbf{rn} + \beta * \mathbf{d}$; %update search direction
19. $\mathbf{r} = \mathbf{rn}$;
20. $rr = rrn$;
21. if (dyn_flag==1) dyn_flag=0;
22. $cntr = cntr + 1$;
23. end

For fixed point Implementation of CG algorithm we utilize the same approach as proposed by Mafi et.al in [1]. The numerical stability and convergence of this method has been proved in the same reference. In this method two different types of scaling have been utilized to improve its numerical accuracy, reliability and convergence rate. Namely *static scaling* and the *dynamic scaling*. In our implementation the elements of **K** matrix as well as **b** vector are defined as 18 bits. **x** vector elements have 36 bits. These bit widthes have been chosen considering limitations imposed by FPGA resources while trying to maximize number of nodes handled by each FPGA and accuracy of results. More details on choosing bit widthes will be presented later in this chapter.

One can note that without proper scaling, the result of the multiplication Kx with the given bitwidths for the K and x can easily overflow the 18-bit vector b. To tackle this problem, the proposed static scaling scheme, vectors **b**, **r** and **d** are scaled down by static scaling factor 2^m. This is performed by scaling down **b** vector which is performed in Step 2 of the CG algorithm. To compensate for this scaling down when updating **x** vector in Step 12, the value used for α is up scaled with the static scaling factor. The proper choice of scaling factor m generally depends on the norm of the actual matrix **K**. Fig. 4.1 The static scaling method saves us a huge amount of memory due to less memory bits needed compared to non-scaled approach for storing the values of the vectors. However it may result in loss of some bits of data in LSB position for b vector.

Moreover by looking at the pseudo code of the CG, one can observe that as the algorithm nears the actual solution, the vectors **r** and **d** will become smaller down

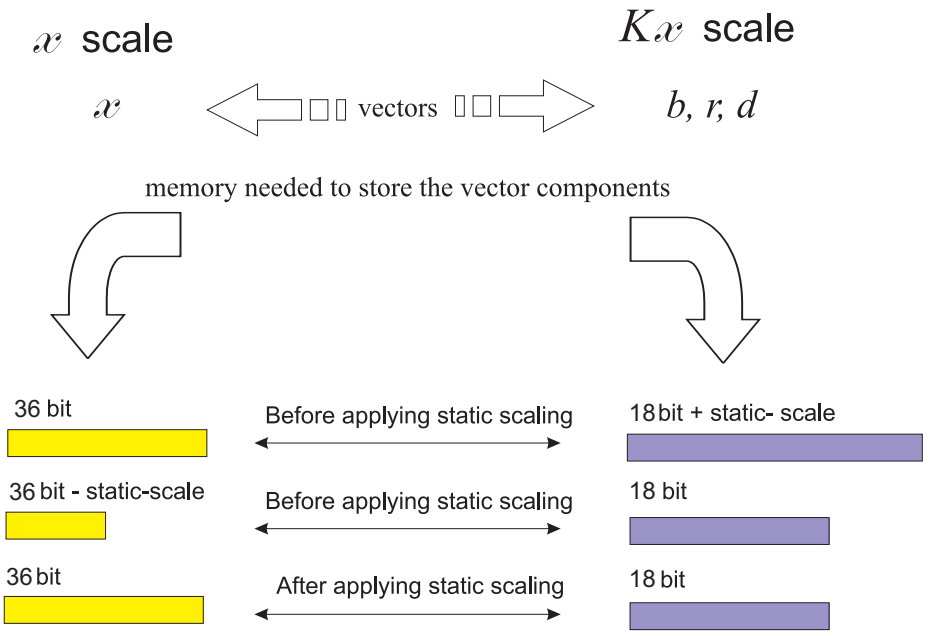

Figure 4.1: Static Scaling. original figure with different bitwidths by Ramin Mafi

to the values comparable with quantization errors. Hence as the size of vector becomes smaller, quantization errors can significantly alter its direction. In the CG algorithm, the updated **d** vector in each iteration is supposed to be a new basis vector for the span of K matrix and by updating **x** vector, we add the projection of the actual solution on new **d** vector direction to vector **x**. In this algorithm, **r** vector is used to find the length of this projected vector.

Therefore, such errors in **r** and **d** vector results in **x** vector updates leading the **x** to wrong directions. This will make quantization errors significant. To mitigate this problem, a dynamic scaling has been employed which scales up **r** and **d** when $\|r\|$ falls down a certain value. This up-scaling of **r** and **d** vectors will decrease relative error in these vectors caused by quantization errors. Finally, for correct update in **x** the added vector to **x** in Step 14 of the pseudo code would be scaled down by the same factor. For further details on the fixed point CG implementation the reader is

referred to [42].

The bitwidth selections for the matrix and various vectors in the implementation of CG are illustrated in Table. 4.1. As will be addressed in Section 6.1, the data bitwidths have been chosen to be multiples of 9. On the other hand, increase in data bitwidths will reduce quantization errors and increase accuracy but will increase resource usage of the FPGAs,resulting in reduced size of K matrix. For a compromise between the accuracy of the fixed-point implementation results and the size of FE mesh we chose the bitwidth for vectors to be 18. Obviously **Kd** would need more bits. Due to our experiments to avoid over flows in **Kd**, it requires 36 bits. Finally the reason to choose 36 bitwidth for **x** vector is to get more accurate results from our fixed-point divider unit which calculates α and β values. For his objective we up-scale the dividend, resulting in up-scaled α and β coefficients. For compensation, we should down-scale the resulted α and β, But for getting more accuracy in results, while updating **x** vector in the algorithm we use the same up-scaled version of α in step 14 of the pseudo code. Hence, to avoid overflow in **x** vector its bitwidth is chosen to be 36 instead of 18. This will increase accuracy in **x** vector by using 9 more M9k memory blocks in our design which is a negligible increase in resource usage of the FPGA.

The fixed-point CG algorithm for solving system of 1707 equations has been implemented using the bitwidth assignment in Table. 4.1 and Matlab's fixed point toolbox. The error in the result vector is depicted in Table. 4.2 for different number of iterations. Fig. 4.2 also depicts the change in norm of the error as a function of the number of iterations. It is observed that after performing 30 CG iterations, the relative error in **x** vector drops under 5%. Furthermore, in Chapter 7, monte

	bitwidth
K	18
x	36
r	18
b	18
d	18
Kd	36
rr	64
dkd	64
α	18
β	18

Table 4.1: Bitwidths assigned for scalars, vectors and K matrix in the fixed-point implementation of the CG

.

carlo type simulations have been carried out for test of numerical stability of this platform. In those tests the dynamic scaling is set to a smaller value which results in faster convergence before 30 iterations and slower rate afterwards.

It is worth mentioning that for this simulation the initial guess for **x** vector has been set to zero, while in haptic simulation in each simulation step we use the result of the previous simulation step as the initial guess. This starting point for x vector is often more close to the actual result, and will result in a more precise answer for **x** vector in a fixed number of CG iterations. The reason for the last obtained **x** vector being a good initial guess for the **X** vector of the next step is that, because of the high update rate (100-1000Hz) the deformation changes in the object caused by the human operator is usually small during a sample time. Thus the new deformation vector would be close to the deformation vector in the last simulation step.

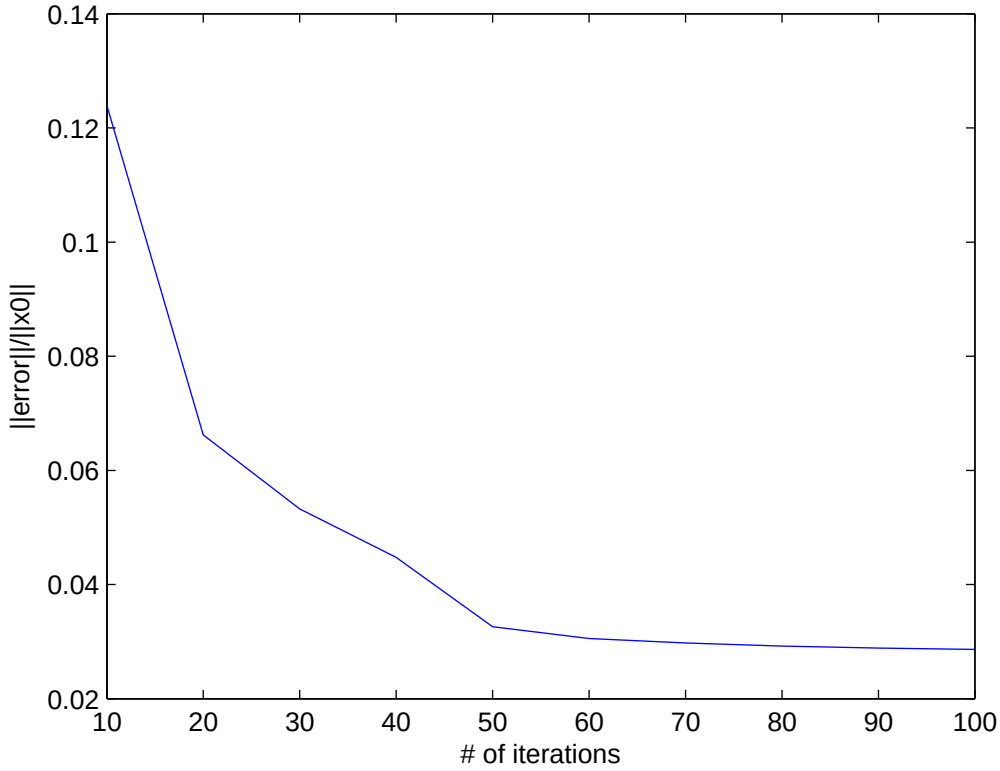

Figure 4.2: Changes in the norm of error in fixed point implementation of the CG as a function of the number of iterations

Preconditioning techniques can be employed to improve the condition number of the matrix $\mathbf{K}$ and ensure that a solution is obtained within the fixed number of iterations imposed by the real-time response constraint. In [48], the Jacobi preconditioning [56] has been applied to the fixed-point CG algorithm. A pseudo-code for the preconditioned conjugate gradient (PCG) is given below. Comparing the PCG pseudo code with the CG pseudo code presented in the beginning of this section, it can been observed that our implementation of the CG algorithm can be enhanced for working PCG algorithm with no structural changes needed in the design. For this purpose the main change needed in the hardware-based CG solver is to add

#iterations	10	20	30	40	50	60	70	80	90	100
$\|\mathbf{x_0}\|$	1.895e+5									
$\|\mathbf{err}\|$	23494	12561	10097	8492	6186	5792	5646	5540	5475	5429
$\|\mathbf{err}\|/\|\mathbf{x_0}\|$	1.24e-1	6.62e-2	5.32e-2	4.47e-2	3.26e-2	3.05e-2	2.97e-2	2.92e-2	2.88e-2	2.86e-2

Table 4.2: Simulation results for fixed-point CG algorithm for a **K** matrix of size 1707*1707.

Step 11 of the PCG pseudo code to the CG, which requires implementation of a block for updating **z** vector.

1. $\mathbf{x} = \mathbf{init}$; %initial guess for solution of Ax=b
2. $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$; %residue
3. $\mathbf{z} = \mathbf{P^{-1}r}$;
4. $\mathbf{d} = \mathbf{z}$; %initial "search direction"
5. $cntr = 1$;
6. $zr = \mathbf{z^T r}$;
7. while $(cntr < \#m)$
8. α = zr/$(\mathbf{d^T Ad})$;
9. $\mathbf{x} = \mathbf{x} + \alpha\mathbf{d}$; %update approximate solution
10. $\mathbf{rn} = \mathbf{r} - \alpha\mathbf{Ad}$; %update the residue
11. $\mathbf{zn} = \mathbf{P^{-1}rn}$;
12. $zrn = \mathbf{zn^T rn}$;

13. $\beta = zrn/zr$;

14. $\mathbf{d} = \mathbf{zn} + \beta\mathbf{d}$; %update search direction

15. $\mathbf{r} = \mathbf{rn}$;

16. $\mathbf{z} = \mathbf{zn}$;

17. $zr = zrn$;

18. $cntr = cntr + 1$;

19. end

A detailed discussion on numerical stability and convergence of Preconditioned Conjugate Gradient algorithm(PCG) is offered in [48]. Finally the DSPF CG algorithm introduced in this chapter has been chosen as a solver for the equations arose by applying FEM in our simulation. This solver, as observed by simulation test results delivers accurate results. Moreover as observed in CG pseudo-code, the computational bottle neck in this algorithm is the matrix by vector multiplication which is very desirable for performing parallel computing to speed up the design. This makes this solver also suitable for being implemented on a parallel hardware architecture.

Chapter 5

Multi-FPGA Design Scheme

In modern FPGA systems several interconnected FPGA chips can reside on a single board providing an opportunity to increase the level of parallelism beyond what is achievable on a single-FPGA design. The need for the simulation of high-resolution FE meshes which can be inhibited the limitations of hardware resources on a single FPGA and justifies a multi- FPGA design approach. In FE analysis, the size of matrix increases linearly by the number of nodes in the mesh representing the deformable object. Increasing the number of nodes in mesh results in higher mesh resolution and an improved approximation of deformation response. On a single-FPGA design, the architecture resource usage grows linearly by the number of nodes in the FE mesh. To achieve an acceptable level of accuracy in practical applications, FE meshes of several thousand nodes may be needed. Unfortunately even the most advanced FPGA devices today lack the sufficient resources to simulate models of such sizes. A multi-FPGA approach in which several devices concurrently solve the system of equations can remedy this problem One of our main

goals in this thesis is to develop a multi-FPGA scheme capable of handling an arbitrary number of nodes in a mesh through increasing the in number of FPGAs working in parallel. In particular, a multi-FPGA CG solver is proposed for solving the system of equation $\mathbf{Kx} = \mathbf{b}$ in a FE analysis. The multiple-FPGA hardware design presented in this thesis has been achieved after undergoing three fundamental design evolutions, where the basis for the first evolution has been the single-FPGA design proposed by Mafi et al. in [1]. Before introducing the new multi-FPGA architecture design, a brief overview on the single FPGA design on Altera EP2S60 by [1] will be given in this chapter. After this overview each of these three design evolutions will be discussed in detail.

5.1 An Overview of the Proposed Single-FPGA Design in [1]

This architecture implements the CG algorithm for solving the system of equations generated by the FE model of a deformable object. As already discussed the bottleneck in computations involved in the CG algorithm is the matrix by vector multiplication where the multiplicand vector is the **d** vector in the CG pseudo code given in Chapter 4. To meet the real-time computation requirement, the hardware architecture must employ as many number of multipliers as possible in parallel. However memory bandwidth limitation restricts the amount of data that can be fed to the multipliers at each clock cycle which prevents exploiting maximum number of multipliers available on the chip. A simple approach to increase the memory bandwidth is to replicate memory blocks. However this is obviously not

a viable solution given the limited amount of on-chip memory and the restricted bandwidth of off-chip memory access. To mitigate this issue, only the non-zero elements of the **K** matrix are saving a significant amount of memory due to sparsity of **K** matrix.

In the proposed architecture, parallelism in matrix by vector multiplication is achieved at several levels explained in the following.

First Level Parallelization (*PL1*): An interesting property of the equivalent stiffness matrix generated by FEM is that each three(3) rows/columns of $3i,\ 3i-1,\ 3i-2$ (for $i = 1, \cdots, n$), have the same indices of non-zero elements. This is due to the fact that each one of these three rows/columns are mapped to one node in the FEM model mesh. A 3D FEM stiffness matrix have a special structure that allows for three(3) values from three(3) rows belonging to the same node to be processed in parallel.

Second Level Parallelization (*PL2*): For further increase in parallelization, while multiplying each row of the matrix by the vector, the multiplication for several elements of that row by their corresponding values from the vector is done at the same time. This increase is determined by the memory bandwidth for reading non-zero elements of that row of the matrix and the vector. Given the memory resources of the FPGA device, the maximum number of columns per node that are processed concurrently is restricted to six(6). Fig. 5.1 depicts the connection of the multipliers and accumulation (MAC) units with the memory blocks containing the multiplicand vector and a portion of the sparse matrix. the MAC units are based on *PL1* and *PL2*.

Third Level Parallelization (*PL3*): As the last level, parallelism is improved by

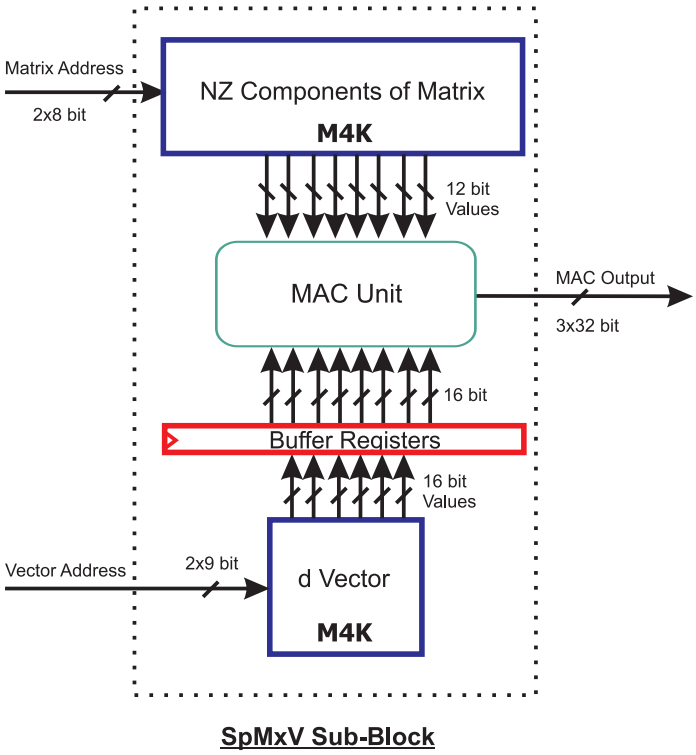

Figure 5.1: The connection of MAC units to memory blocks. Figure from [42]

dividing the matrix **K** into multiple partitions in a row wise manner, each containing several nodes as shown in Fig. 5.2. For each new partition 18 extra multipliers are required as shown in Fig. 5.3. The multiplicand vector is replicated per each partition as well.

Obviously a lager number of partitions leads to a higher degree of parallelization. In the proposed single-FPGA design on EP2S60 device, this number is limited to five(5) due to the available on-chip memory and multiplier units.

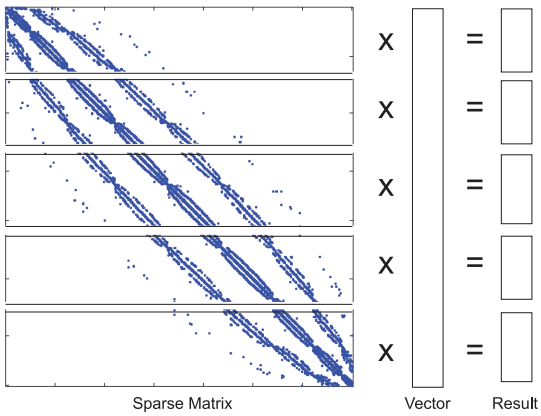

Figure 5.2: Matrix partitioning for increased parallelism. Figure from [42]

5.2 Multiple FPGA Design

In [1], three parallelization levels named $PL1$ to $PL3$ were utilized to fully employ the parallel processing capabilities of the FPGA. In this section a generalization of the hardware architecture to an N-FPGA topology will be introduced increasing parallelism to four levels.

Dividing the computations among multiple FPGAs results in one more parallelization level compared to the single-FPGA design. We refer to this as the first level of parallelization, $PL1$. The second level of parallelization in the multi-FPGA design, is equivalent to the third level of parallelization in single-FPGA design by [1]. In the multi-FPGA design $PL2$ divides the portion of matrix on each FPGA to L sub-partitions. Therefore, employing the first and second levels of parallelization using multiple FPGAs yields a total number of $N * L$ sub-partition, performing matrix by vector multiplication in parallel. Fig. 5.4 depicts how the matrix partitioning is done for $PL1$ and Fig. 5.5 illustrates the final partitions of the **K**, considering

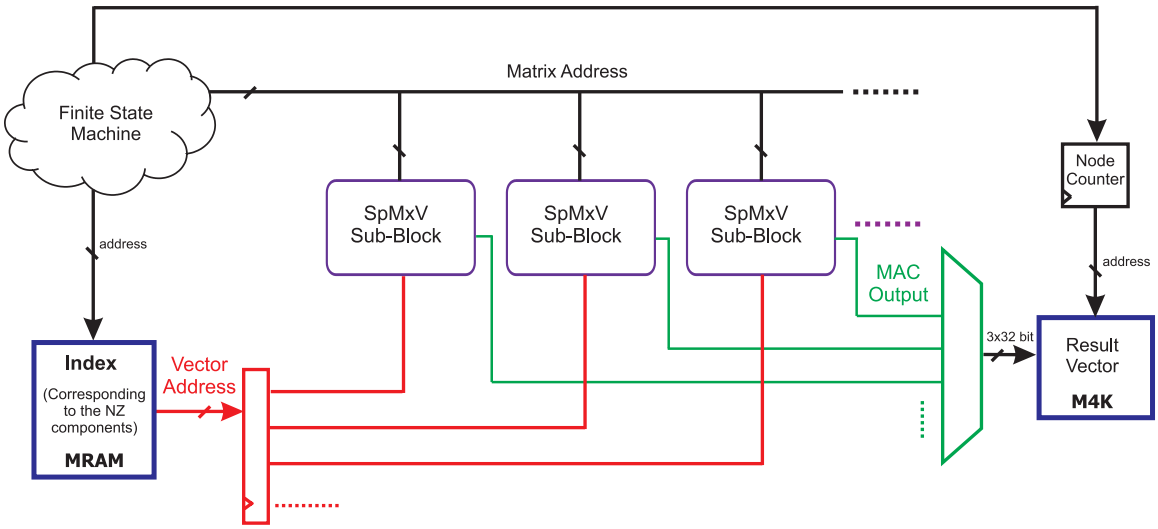

Figure 5.3: Third level of parallelization: top-view block diagram of the hardware-based sparse matrix by vector multiplication. Figure from [42]

both $PL1$ for dividing the matrix between multiple FPGAs and constituting sub-partitions on each FPGA for $PL2$.

In developing $PL1$ two criteria must be considered. First, the FPGAs should work as independently as possible with the least amount of communication among them. In addition there should be a minimum of data overlap among the FPGAs to optimize memory usage. This would ensure that the solution can be properly scaled using the limited memory available on each FPGA. To this end, a partitioning of the CG algorithm is proposed as shown in Table 5.1 in which superscripts i $1 \leq i \leq N$ denotes the ith sub-partition of the corresponding matrices and vectors, respectively. Moreover, the arrows between pairs of FPGAs represent steps involving communication, details of which will be discussed later. As can be seen in Table 5.1, the computations involved at each step of the algorithm are divided such that each FPGA only needs to store one out of N parts of the matrix, i.e. $\mathbf{K^i}$, $1 \leq i \leq N$. Among the vectors in the CG algorithm in Table 5.1, only $\mathbf{d}$ and the

FPGA #1	FPGA #2	...	FPGA #N
1) $\mathbf{x = init}$	$\mathbf{x = init}$	...	$\mathbf{x = init}$
2) $\mathbf{r^1 = b^1 - K^1 x}$	$\mathbf{r^2 = b^2 - K^2 x}$	...	$\mathbf{r^N = b^N - K^N x}$
3) $\mathbf{d^1 = r^1}$	$\mathbf{d^2 = r^2}$	...	$\mathbf{d^N = r^N}$
4) $send\ \mathbf{d^1}, recieve\ \mathbf{d^2}...\mathbf{d^N}$	$\rightleftharpoons send\ \mathbf{d^2}, recieve\ \mathbf{d^3}...\mathbf{d^1}$	$\rightleftharpoons$ ...	$\rightleftharpoons send\ \mathbf{d^N}, recieve\ \mathbf{d^{N-1}}...\mathbf{d^1}$
5) $cntr = 1$	$cntr = 1$	...	$cntr = 1$
6) $rr_1 = \mathbf{r^{1\prime} r^1}$	$rr_2 = \mathbf{r^{2\prime} r^2}$	...	$rr_N = \mathbf{r^{N\prime} r^N}$
7) $accumulate\ rr_i\ scalar$	$\rightleftharpoons accumulate\ rr_i\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons accumulate\ rr_i\ scalar$
8) $distribute\ rr = \sum_{i=1}^{N} rr_i\ scalar$	$\rightleftharpoons distribute\ rr\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons distribute\ rr\ scalar$
9) while $(cntr < \#itr)$	while $(cntr < \#itr)$	...	while $(cntr < \#itr)$
10) $Kd^1 = K^1 * d$	$Kd^2 = K^2 * d$	...	$Kd^N = K^N * d$
11) $dkd_1 = d^T * Kd^1$	$dkd_2 = d^T * Kd^2$	...	$dkd_N = d^T * Kd^N$
12) $accumulate\ dkd_i\ scalar$	$\rightleftharpoons accumulate\ dkd_i\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons accumulate\ dkd_i\ scalar$
13) $\alpha = zr\ /\ (dkd)$, where $dkd = \sum_{i=1}^{N} dkd_i$		...	
14) $distribute\ \alpha\ scalar$	$\rightleftharpoons distribute\ \alpha\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons distribute\ \alpha\ scalar$
15) $\mathbf{x^1 = x^1 + \alpha d^1}$	$\mathbf{x^2 = x^2 + \alpha d^2}$	...	$\mathbf{x^N = x^N + \alpha d^N}$
16) $\mathbf{rn^1 = r^1 - \alpha K^1 d}$	$\mathbf{rn^2 = r^2 - \alpha K^2 d}$	...	$\mathbf{rn^N = r^N - \alpha K^N d}$
17) $rrn_1 = \mathbf{rn^{1\prime} rn^1}$	$rrn_2 = \mathbf{rn^{2\prime} rn^2}$	...	$rrn_N = \mathbf{rn^{N\prime} rn^N}$
18) $accumulate\ rrn_i\ scalar$	$\rightleftharpoons accumulate\ rrn_i\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons accumulate\ rrn_i\ scalar$
19) $distribute\ rrn = \sum_{i=1}^{N} rrn_i\ scalar$	$\rightleftharpoons distribute\ rrn\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons distribute\ rrn\ scalar$
20) $\beta = rrn/rr$		...	
21) $distribute\ \beta\ scalar$	$\rightleftharpoons distribute\ \beta\ scalar$	$\rightleftharpoons$ ...	$\rightleftharpoons distribute\ \beta\ scalar$
22) $\mathbf{d^1 = rn^1 + \beta d^1}$	$\mathbf{d^2 = rn^2 + \beta d^2}$	...	$\mathbf{d^N = rn^N + \beta d^N}$
23) $send\ \mathbf{d^1}, recieve\ \mathbf{d^2}...\mathbf{d^N}$	$\rightleftharpoons send\ \mathbf{d^2}, recieve\ \mathbf{d^3}...\mathbf{d^1}$	$\rightleftharpoons$ ...	$\rightleftharpoons send\ \mathbf{d^N}, recieve\ \mathbf{d^{N-1}}...\mathbf{d^1}$
24) $\mathbf{r^1 = rn^1}$	$\mathbf{r^2 = rn^2}$	...	$\mathbf{r^N = rn^N}$
25) $rr = rrn$	$rr = rrn$	...	$rr = rrn$
26) $cntr = cntr + 1$	$cntr = cntr + 1$	...	$cntr = cntr + 1$
27) end	end	...	end

Table 5.1: The pseudo-code of CG and partitioning of the algorithm for implementation on N FPGAs. This method applies to all 3 Design evolutions.

initial guess for x have to be fully maintained on all FPGAs whereas all other local vectors are splitted into almost $\frac{1}{N}$ of their original size to minimize memory usage. It should be noted that all the steps except those for communication among the FPGAs are independent of each other and hence can be executed concurrently.

The final multi-FPGA design presented in this thesis has been achieved after going through three design evolutions. In the first design our objective was to extend the design proposed by Mafi et al. [1] foe implementation on a Procstar II FPGA board with two Stratix II EP2S60 Altera devices, doubling the total number of nodes in our FE-based simulation.

After accomplishing this, we made the second design for a quad-FPGA board with

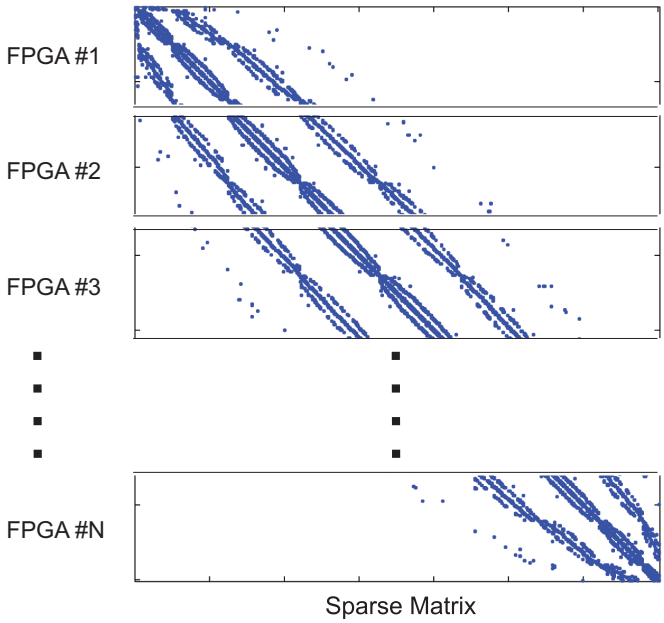

Figure 5.4: Portions of **K** matrix stored on each FPGA for performing first level of parallelization

four Altera Stratix III EP3SE110 devices. By employing these more powerful FPGAs we increased the number of nodes on each FPGA in this step. This required some changes to the architecture design in order to still meet timing requirements after increasing the number of nodes on each FPGA. However this design has limited potential for scalability on multiple FPGAs due to its increasing memory usage on each FPGA by an increase in the number of nodes.

Our third design resolves the issues emerged in the second design by developing a storage format for storage of **d** vector (SMVIS) and employing a new SpMxV unit. Each of these design evolutions are briefly introduced in this chapter and the final design will be reviewed in more details in Chapter 6.

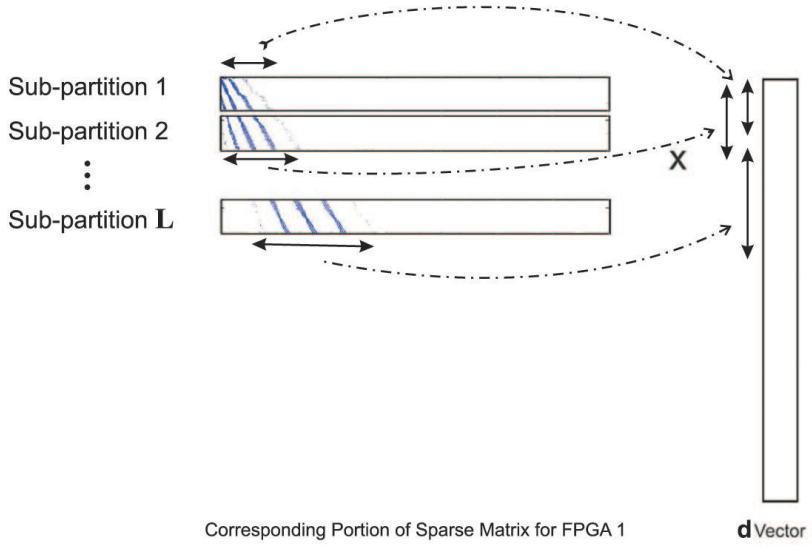

Figure 5.5: Partial storage of the vector **d** for FPGA#1; Note that the corresponding elements of the vector for each sub-partition is marked by arrows.

5.2.1 Design I

In [1], a custom fixed-point CG algorithm was implemented on a single Altera Stratix II device using a Gidel ProcStar II board. The board hosts two FPGA devices and in the first architecture design the objective is to utilize both of the FPGAs.

The first step for developing the multi FPGA architecture to find a method for the division of computations between the two FPGAs. For this purpose, the distributed CG algorithm for multiple-FPGAs in Table 5.1 is utilized with $N = 2$.

The first architecture uses almost the same structure for each FPGA as described in [1].

The First evolution of the multi-FPGA design was successfully implemented on two FPGAs. This two-FPGA design is capable of doubling the maximum number of nodes, handled by the single-FPGA design. The double-FPGA design, solves a

system with a maximum number of nodes of 1024 after 60 iterations of CG algorithm in almost the same 2ms time frame as needed for the single-FPGA design to perform 60 CG iterations for 512 nodes. The computation time for the dual-FPGA design is more or less the same as the single-FPGA design since the time needed for the communication of $\mathbf{d}$ vector in Step 23 of the Table 5.1, the other added steps in multi-FPGA scheme consume ignorable amount of time. Assuming that both FPGAs need the same amount of time to finish the matrix by vector multiplication,they reach to Step 23 of the Table 5.1 at the same time with no overhead time. In the double-FPGA design, the Steps 22 and 23 in Table 5.1 are combined for each FPGA by carefully considering the circuit timings. This allows for both update and communication of $\mathbf{d}$ vector on each FPGA to be carried out at the same amount of time needed for just updating $\mathbf{d}$ vector in [42].

5.2.2 Design II

In the second evolution of our multi-FPGA architecture, the goal is to extend the design to work on a four Altera Stratix III FPGAs, sitting on a Gidel ProcStar III board.

In this case, we have to both scale up the old design on each FPGA so it can handle more nodes per FPGA and also scale it to four FPGAs. First we will discuss the former.

5.2.3 Scaling up the Number of Nodes on Each FPGA

Scaling up the number of nodes on each FPGA requires scaling up the implemented design on each FPGA. The design is almost the same in the single-FPGA and double-FPGA architectures except for the added communication step in the double-FPGA design. First we will investigate the expansion of the old architecture to the new device, that is migrating from Altera Stratix II (EP2S60) to a Altera Stratix III (EP3SE100) for a single FPGA design.

The available resources on these two FPGAs are compared in Table 5.2.

Feature	EP2S60 FPGA Device	EP3SE110
M512 RAM Blocks (512 bits + Parity)	329	-
M4K RAM Blocks (4 Kbits + Parity)	255	-
MRAM Blocks (512 Kbits + Parity)	2	-
Embedded Multipliers (18×18)	144	896
M144K RAM(144 Kbits + Parity)	-	14
M9K RAM Blocks(9 Kbits+Parity)	-	639

Table 5.2: Comparison of Altera Stratix II EP2S60 and Altera EP3SE110 resources

In the design for Stratix III device, the M4K and MRAM RAM blocks in the old Stratix II device are substituted by M9k and M144k RAM blocks for the new FPGA respectively. Therefore, for obtaining an optimal architecture for the new device, the old architecture had to undergo some changes in memory architectures.

For scaling up the old design on each FPGA, in principle, two major changes are needed. First, because of the increase in the number of nodes on the FPGA, the memory architecture has to change in order to allow for storage of larger matrix

and vectors. Second, the matrix by vector multiplication, vector by vector multiplication and vector updater units have to change, as will be described later, such that the new FPGA performs the CG computations for more number of nodes without increasing the time frame. The rationale behind the first type of changes is obvious. The second type of changes are needed to curb the linear increase in the computation time on each FPGAs as the number of nodes increases. Another issue is that, in the single-FPGA design, the amount of resources needed in the single-FPGA design will increase linearly with the number of nodes, but the improvements in resources of the new FPGA (EP3SE110) are not linear comparing to the old FPGA (EP2S60). That is, the amount of DSP elements, embedded memories and LUTs do not increase by the same proportion. Therefore, we will not achieve an optimal architecture on the new devices with just scaling up elements of design. Moreover as can be seen in Table 5.3, in the new devices there are some structural changes in the embedded memory blocks and architectures as stated in Table 5.2 , which impose more changes on the design.

To scale up the design to the new FPGA considering all these aspects, a hardware architecture is proposed that can process a maximum of 1536 nodes on each FPGA with the same data widths as in [1], that is 12 bits for K matrix values, 16 bits for all vectors except Kd multiplication result vector which has a 32 bit representation.

As the second design is a transition step from the single-FPGA design to the Design III, we will only provide a summery of changes performed compared to the Design I and we will skip the details.

The first change required in the old design concerns its memory architecture. In the new design, vectors utilize three M9Ks instead of one M4K in the old design order to have the capacity to store values corresponding to 1536 nodes instead of 512 nodes. Moreover due to the larger number of sub-partitions in the new design and the increased capacity of M9Ks compared to M4Ks, the memory structure for storing non-zero elements of **K** matrix has to change. The other change in memory architecture of the new design is the different configuration of M144k memories for storing indices of the non-zero elements of **K** matrix compared to the configuration of their counterparts (MRAMs) in the old architecture.

After making proper changes to the memory architecture of the old design to fit the resources on the new FPGA, we also need to ensure that the computation timing constraint will not be violated.
The time required for an iteration of the CG algorithm is mainly taken by two types of operation, namely sparse matrix by vector multiplication and updating x, r and d vectors in the Steps 15, 16, 17 and 22 of the main algorithm Table 5.1. To avoid an increase in the matrix by vector multiplication time, number of sub-partitions in $PL2$ is linearly increased in accordance with increase in number of nodes. This will prevent increase in number of nodes in each sub-partition, So we can get the result of sparse matrix by vector multiplication without consuming any excess time. Hence, by tripling number of nodes in second design, the number of sub partitions should be also tripled, that is, it should increase from 5 to at least 15. Our second design has 16 sub-partitions, which is the maximum number of sub-partitions we could have due to hardware resources limitations.

The problem about the second major time consuming type of operations (vector updates) is that with an increase in number of nodes, vector updating times will increase linearly with increase in size of vectors. In the previous design because of the relatively short length of vectors (3*512), the required time for vector updates was not significant compared to the time consumed by sparse matrix by vector multiplication. However as the number of nodes grows the vector update time also increases, preventing the design from meeting real-time response requirement.

To address the increase in vector update times due to the longer vector lengthes, changes are made in memory architecture of vectors. To increase speed of vector updating we need more bandwidth for reading data from and writing data to vectors at each clock cycle. So for each vector instead of instantiating memory blocks with two read/write ports and 1536 words depth, we use a memory architecture as depicted in Fig. 6.6. In this architecture, we multiplex a number of (3 in this design) smallest dual port memory blocks available (for Stratix EPS3E110 this is 512x16 bits for 1 M9K). In each state of updating vectors we can de-multiplex the memory blocks of the vector and concurrently update these three memory blocks. This matter is discussed further in Section 6.2.2. With this approach the time needed for updating each vector will be limited to length of the $smallestmemoryblock * 2$ clock cycles regardless of the number of nodes involved. Here the constant 2 stands for one clock cycle for reading the old value and one for writing new value. In the Altera® Stratix III® EPS3E110, this time will be 512x2 clock cycles. The method

for limiting the vector update times to make it independent from number of nodes is also utilized the next architecture design which will be discussed in details in Chapter 6.

5.2.4 Scale the New Design to Multiple FPGAs

The extension of the computing architecture to N FPGAs in our approach, in its simplest form, requires $N \times L$ full copies of $\mathbf{d}$ vector. Meanwhile, extending the design into N FPGAs will increase the length of $\mathbf{d}$ vector by $\sim N$. This means that the total memory usage for all vectors in the CG algorithm grows linearly by N, except the required space for $\mathbf{d}$ and $\mathbf{x}$ vectors which increases by N^2.

As a result an N-FPGA configuration would not exactly increase the number of nodes by a factor of N when compared to a 1-FPGA architecture. To mitigate this problem, by proper graph partitioning and node numbering we should guarantee that for each partition the range of nonzero elements would be within 50% of the full length of the vector. With our node numbering technique and due to matrix sparsity this was always the case in our test matrices although for some mesh shapes it might not work. In this method we start numbering the nodes from a center node inside the mesh and the numbers of nodes increase by getting farther from that node and getting near the surface nodes. By this assumption on the matrix structure, each sub-partition we will only need less than half of $\mathbf{d}$ vector to generate the corresponding part of Kxd.

We applied this method on several meshes and all of them satisfied the 50% assumption. For instance for the sphere mesh used as an example in this thesis this

range is about 27%. Thus in our off-line computations for each sub-partition we find the range of **d** vector corresponding to the indices of nonzero elements and send the first and last addresses for this range to the FPGA. Fig. 5.5 illustrates the parts of **d** vector needed to be stored for each sub-partition of K matrix for FPGA#1 which has L partitions in totals.

The proposed approach decreases the memory usage growth of the vector **d** to $N^2/2$. Fig. 5.5 illustrates the parts of the vector **d** needed to be stored for each sub-partition of the matrix K for FPGA#1 which has L sub-partitions in total.

5.2.5 Design III

Both previous multiple-FPGA designs where mainly utilizing the core architecture proposed by [1] for single-FPGA design to perform the sparse by matrix multiplication. As described in Section 5.2.2, using this scheme for SpMxV will prevent the design from meeting timing requirements by increase in the number of FPGAs, for example it limits us to four FPGAs in case of using EP3SE110. In the third design, we propose a new SpMxV scheme which removes the limitation on increasing number of FPGAs. This new SpMxV engine will be described in detail in Chapter 6. In summary new SpMxV architecture:

1. Allows more sub-partitions on each FPGA by utilizing SMVIS (will be introduced in Chapter 6 data storage format for **d** vector.

2. Speeds up the SpMxV operation by utilizing a new method for storing $K * d$ multiplication results.

3. Lowers the multiplier usage for each sub-partition in comparison with the

old SpMxV unit employed in the single-FPGA design.

The 3^{rd} and last evolution of our multiple-FPGA design, in case of being provided by adequate bandwidth for inter-FPGA communications, will solve a system of sparse linear equations using the CG method for an arbitrary amount of nodes on sufficient number of FPGAs. It it worth mentioning that for expansion of the design to an arbitrary number of FPGAs, the bandwidth for data communication should linearly grow by an increase in the number of FPGAs. If the bandwidth for data communication among the FPGAs remains unchanged, the communication time for exchanging $\mathbf{d}$ vector among the FPGAs will linearly grow with increase in number of FPGAs. Considering limitations on data bandwidth for communications, we will be limited to a maximum of 44 FPGAs in case of utilizing Low Voltage Differential Signalling (LVDS) for serial communications among FPGAs as will be discussed later in Section 6.5.

Another issue in the previous designs, affecting timing is the method used for the storage of sparse matrix by vector results. To efficiently utilize the memory blocks when storing non-zeros of $\mathbf{K}$ matrix, the sub-partitions are chosen such that they all carry an average number of non-zeros. Therefore the number of nodes in the last sub-partition of the last FPGA is much more than other sub-partitions due to the more sparsity of that part of the matrix. In the previous designs, to simplify the storage of $\mathbf{Kd}$ multiplication result, all sub-partitions in an FPGA start the multiplication for the next row just after all of them have finished the multiplication and storage of their previous rows. This blocks all of the FPGAs until the multiplication for the last sub-partition of the last FPGA finishes.

In the new architecture, a whole new sparse by vector multiplication block is designed, solving the problem in the previous designs while making an order of magnitude increase in the speed of sparse matrix by vector operation by utilizing 42 sub-partitions instead of 16. Moreover, in the new architecture the data bitwidths have increased as depicted in Table 5.4 in order to increase the accuracy.
There has been also a new method proposed for storage of $\mathbf{d}$ vector in design three, eliminating the need for storing multiple copies of $\mathbf{d}$ vector. This architecture will be explained in details in Chapter 6. The extension of the computing architecture to N FPGAs in our approach as described in Table 5.1, in its simplest form, requires L full copies of $\mathbf{d}$ vector on each FPGA, where L is the number of sub-partitions on each FPGA. Meanwhile, extending the design into N FPGAs will increase the length of $\mathbf{d}$ vector by $\sim N$. This means that by an increase in number of FPGAs, the total memory required for all vectors in each FPGA will remain constant, except the required space for $\mathbf{d}$ vector which increases by N.

The proposed method for saving half of **d** vector for each sub-partition can mitigate the problem but will still prevents us from increasing the number of FPGAs more than a certain number depending on the type of FPGA we are using.

To tackle this problem an indexing method for **d** vector has been proposed which will be discussed in detail in Chapter 6 . By indexing d vector for each sub-partition of the matrix we will just have to store the values of **d** vector corresponding to non-zero elements of that sub partition. Thus d vector corresponding to each sub-partition uses a certain amount of memory resources (one M9K in this architecture) independent of the number of FPGAs(N).

Design#	Device	Max. Number of FPGAs	Max. number of nodes per FPGA	Number of sub-partitions per FPGA	Limited time for vector operations regardless of num. of nodes	d vectors for sub-partitions with half length of full d vector	Utilizing SMVIS for d vector storage	New enhanced SpMxV result storage scheme	New SpMxV unit decreasing num. of employed multipliers	Increased Data bitwidths
Single-FPGA	EP2S60	1	512	5	×	×	×	×	×	×
I	EP2S60	2	512	5	×	×	×	×	×	×
II	EP3SE110	4	1536	16	✓	✓	×	×	×	×
III	EP3SE110	44	1536	≤ 42	✓	×	✓	✓	✓	✓

Table 5.3: Comparison of important characteristics for different evolutions of the multiple-FPGA Architectures

To summarize the improvements achieved by different designs, we will have a comparison between important parameters and characteristics of these different designs in Table 5.3.

	Data bitwidths for Single-FPGA and Designs I and II	Data bitwidths for Design III
K	12	18
x	16	36
r	16	18
b	16	18
d	16	18
Kd	32	36

Table 5.4: bitwidths assigned for vectors and K matrix in the fixed-point implementation of the CG

Table 5.4 illustrates the data bitwidths used for fixed point representation of K matrix elements and different vectors in different designs. A comparison between the single-FPGA design and the final multi-FPGA design (Design III) in terms of timing is given in Table 5.5. The data provided in the table are based on both designs working for 468 nodes and at a clock frequency of 100MHz.

Operation	Required time by Design III implemented on 4 FPGAs	Required time by Single-FPGA design
sparse Matrix by vector multiplication	1236 ns	14249 ns
vector by vector operations for **r** and **x**	10273 ns	9429 ns
vector by vector operations for **d**	2264 ns	9409 ns
d vector communication	8289 ns	–
Total stall time for multi-FPGA design communications	3008 ns	–
Total time for one CG iteration	25070 ns	33749 ns
Total time for 30 CG iterations	0.7521 ms	1.0125 ms

Table 5.5: Timing comparison between the final multi-FPGA design (Design III) and single-FPGA design for one CG iteration.

In this table, the times given for multi-FPGA design are based on the timing of FPGA#1, thus the total stall time for multi-FPGA design represents the total time in one CG iteration when FPGA#1 is ready for communication while the FPGA in the other side has not reached that point.

It should be noted that as explained in Section 5.2.2, the vector operations, vector by vector multiplication and vector updates in multi-FPGAs design take a constant time for all vector sizes, equal to the time needed to update a vector of size 1536×1. Therefore, when the number of nodes on each FPGA is less than 512, this constant time will have negative effect on timing. To investigate this issue further, in the timing tests for Table 5.5, among two steps which perform vector

operations, one for updating **r** and **x** vectors and the second for updating d vector, we applied two different methods of updating vectors. The former vector by vector operation, Steps 15 and 16 of the algorithm in Table 5.1, utilizes the new architecture for vector operations with fixed time for different vector lengths and the latter, updating **d** vector in Step 22 of the same algorithm, utilizes the old method as in the Single-FPGA design with time depending on the vector length. If the second method was also used for first step of vector operations as well, the total time for 30 iterations of the CG algorithm would decrease to about 0.5 milliseconds for the Design III. In the four FPGA design, each FPGA has to perform the calculations for about 115 nodes.

The results of performance analysis for the new SpMxV unit in Table 5.5 are very encouraging results. The SpMxV operation using the new multi-FPGA scheme in Design III requires a tenth of the time used by single-FPGA design.

To summarize, after improvements made on the multiple-FPGA design in three steps, Design III can be implemented on an arbitrary number of FPGAs devices satisfying timing requirements for a real-time response as required in haptic applications. The increase in the number of FPGAs would be dependent on the available inter-FPGA communication bandwith. In the next chapter we will discuss this final architecture in detail.

Chapter 6

Hardware Architecture

In this chapter,multi-FPGA hardware architecture denoted as Design III in Chapter 5 will be discussed in detail. The CG algorithm is composed of three types of operations : vector by vector multiplication, vectors-updates and matrix by vector multiplication. The vector operations, vector by vector multiplication and vector-updates, in Steps 15,16 and 17 of the algorithm in Table 5.1 can be carried out simultaneously, and are not main of concern in terms of resource usage and complexity. It is the matrix by vector multiplication is the bottleneck in the algorithm both in timing and resource usage and must be made as fast as possible to meet the timing constraints although performing vector operations is also speeded up by as described in Chapter 5.

To increase the number of nodes and simulation accuracy, the size of the portion of K matrix stored on each FPGA should be as large as possible.Moreover for larger matrices a greater number of multipliers must be utilized to avoid increase in multiplication time. This implies the need for a high memory bandwidth to feed a large number of multipliers. The data should be stored in memory units efficiently to

maximize throughput and storage while avoiding data replication and storage of zero values of the sparse matrix. To summarize, the main challenges for designing an architecture meeting our application requirements are in:

- designing a data storage format for efficient use of memory resources to increase the size of the FE mesh
- utilizing as many multipliers as possible in parallel for SpMxV
- designing a memory architecture with maximum data access bandwidth
- designing an architecture that would be scalable both on each FPGA device and to multiple FPGAs
- designing the communication scheme for required inter-FPGA data exchanges.

6.1 Analysis of Hardware Limitations

Although the proposed architecture has been designed to work on all FPGA types, its implementation has been optimized for use on Altera EP3SE110 and Gidel ProcStar III board shown in Fig. 6.2. This platform imposed some limitations on the design as will be discussed shortly.

The resources on the FPGAs provided by Altera [58] are summarized in Table 6.1. It is worth noting that M9Ks, which are the most significant portion of the embedded memory blocks on each chip, are the new generation of embedded memories, substituting M4Ks in former generation of Altera devices. Each of these

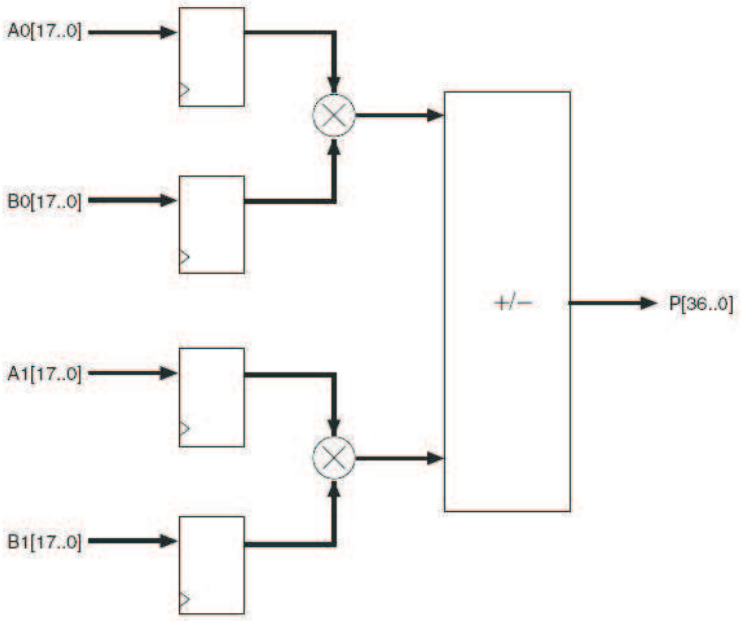

Figure 6.1: Basic two-multiplier adder building block

blocks, contains 9Kbit of data which can be most efficiently used in multiples of 9-bit data words. This would be compatible with 9-bit DSP units.Each Stratix III EP3SE110 contains 112 DSP blocks [58]. Each of these 112 DSP blocks is composed of two half DSP blocks and each of these half DSP blocks consists of two fundamental building blocks as depicted in Fig. 6.1. Unlike Stratix and Stratix II devices in which fundamental building block consists of one 18 * 18 bit multiplier, the DSP block in Stratix III consists of two pairs of 18 * 18 bit multipliers followed by an adder with 37 bit output result [58], as illustrated in Fig. 6.1. Therefore by using independent 18 * 18 bit multipliers one can obtain up to 448 multipliers. All 896 18 * 18 multipliers can be fully utilized only in a $mult\_add$ architecture depicted in Fig. 6.1.

Moreover the overall speed of the architecture would be determined by the FPGA's logics, if faster logics were utilized or if bigger chips were available for

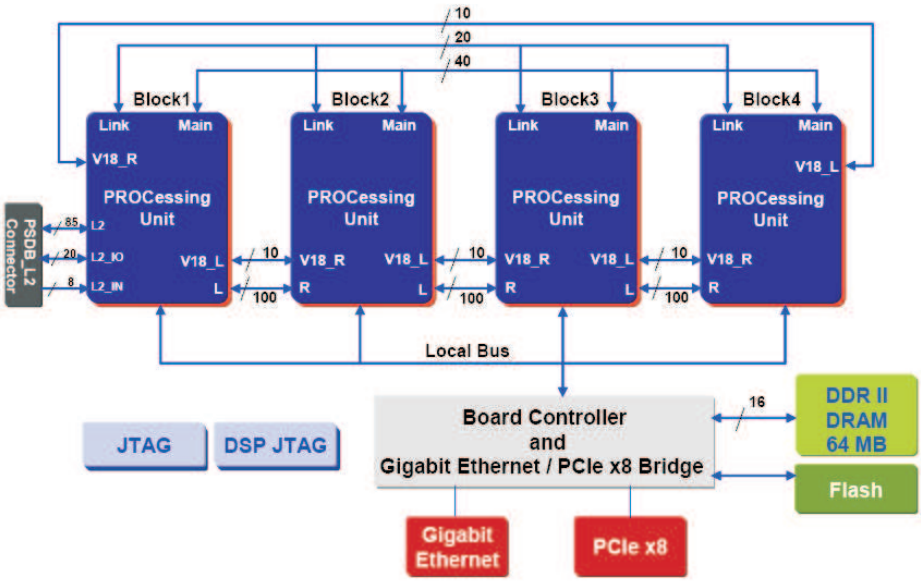

Figure 6.2: ProcStar III system block diagram [4]

implementation, the speed of computations performed in this architecture would increase.

Another limitation on our design imposed by the Gidel board concerns inter-FPGA communications.

6.2 Data Storage

After going through many design iterations finally the following memory architecture was chosen. This architecture meets our requirements in terms of data throughput, capacity for handling large matrices and also ease of scalability. Referring to the CG algorithm in Section 4.2, the following data must be stored on each FPGA:

- K matrix

ALMs(Adaptive Logic Modules)	43K
LEs (Logic Elements)	107.5K
M9K Blocks (9Kbit dual port memories)	639
M144 Blocks(144Kbit dual port memories)	16
Mlab Blocks(SRAM memory array blocks)	2150
DSP Blocks	112
18*18 Multipliers	448
Four Multiplier Adder Mode	896
PLLs	8

Table 6.1: EPS3E110 device overview

- **d** vector
- **b** vector
- **x** vector
- **r** vector
- $K * d$ result vector (**Kd** vector)

We will start by storing K matrix here and will go through storage of other vectors in order. To better utilize of memory resources, only the non-zero values of the K matrix are stored in a format described later in Section 6.2.1. As mentioned earlier in Chapter 5, **d** vector is stored for different sub-partitions, utilizing its own storage method. In this method in each sub-partition, only elements of **d** vector which correspond to non-zero values of the sub-partition are stored. This storage method will avoid storage of a full **d** vector for each sub-partition.

6.2.1 Data Storage Formats

Among the vectors and other data needed to be stored on FPGAs in the implementation of the CG algorithm, K matrix occupies the biggest ratio of the embedded memory. But fortunately the K matrix arising form the FEM is highly sparse with ratio of non-zeros of about 3% by our meshing method and using tetrahedral finite elements for meshes with more than 400 nodes [42]. Therefore, for preventing extra usage of the memory and time for storing and processing zeros we would apply a method for only storing the non-zero elements of K matrix. Before describing our data storage approach, first we will briefly review some of popular methods for storage of sparse matrices. These include the Compressed Row, Compressed Column, and Coordinate Storage Schemes (CRS, CCS, CSS) [59]. Each of these formats has its own advantage in utilizing specific property of sparsity in matrix, resulting in different degrees of space efficiency [60].

CRS Format

The CRS format represents a matrix by three vectors, Val, Col and Rowptr. It starts from the top left of the matrix and goes row by row, storing the non-zero values in Val vector, the corresponding column index for each non-zero value in Col vector in the same order and the number of the first non-zero value of each row in Rowptr.

CCS Format

The CCS format is similar to CRS, except that instead of going row by row, it begins from the top left corner of the matrix and goes column by column. Therefore,

2	0	3	0
1	1	4	0
0	0	5	0

Figure 6.3: A sample matrix used to demonstrate data storage formats

each sparse matrix will be presented by three vectors of Val, Row and Colptr. This method stores the same data for a matrix as CRS format would store for the transpose of the matrix.

CSS Format

This method is similar to the CRS method but instead of Rowptr it keeps the row indices corresponding to all non-zero values stored in Val in vector Row.

To demonstrate how each of these methods work, they are applied to the matrix shown in Fig. 6.3. The results are as follows.

- CRS format:

  Val:[2,3,1,1,4,5]

  Col:[0,2,0,1,2,2]

  Rowptr:[0,2,5]

- CCS format:

  Val:[2,1,1,3,4,5]

Row:[0,1,1,0,1,2]

Colptr:[0,2,3]

- CSS format:

  Val:[2,3,1,1,4,5]

  Col:[0,2,0,1,2,2]

  Row:[0,0,1,1,1,2]

The CSS format because of saving indices for both row and column needs more memory resources. The size of this extra memory becomes significant as the size of the sparse matrix grows. The CRS and CCS formats require the same amount of memory to store data. However for Sparse Matrix by Vector multiplication, the CRS format is more appropriate as it delivers better memory access for data stored in each sub-partition of the matrix. That is, for performing SpMxV multiplication considering second level of parallelization in our design introduced in Chapter 5, we divide the matrix into some sub-partitions and then the multiplication of matrix by vector for each of these sub-partitions is performed independently. In the CRS format, to read values from different sub-partitions of the matrix at the same time we just have to keep the Val vector and its corresponding Col vector in a separate memory. This would be feasible in the CCS format.

Storage Format for K Matrix

The method we use for storing K matrix is a modified CRS format, that is, we represent the matrix by two vectors Val and Col which are the same as their counterparts in CRS format. The only difference is that instead of using the Rowptr vector a zero value is stored at the end of non-zeros of each row.Therefore during the multiplications we will just have to read the non-zero values from Val in the same order that they have been stored and multiply them by the values read from **d** vector with the indices read from Col vector.

Storage Format For d vector

Performing matrix by vector multiplication for a full (non-sparse) matrix for each row of the matrix requires the data of the full vector. As already explained in Section 5.2, to increase the multiplication speed, in this architecture we perform the multiplication for rows of different sub-partitions of the matrix at the same time (42 rows in the current implementation).

Due to the difference in patterns of non-zeros in each of rows, a different portion of **d** vector would be needed for the multiplication. This requires storing multiple copies of **d** vector equal to the number of sub-partitions of the matrix such that each sub-partition drives the address line of its dedicated **d** vector by the values it reads from the Col vector as illustrated in Fig. 6.4.
Such approach will waste a huge amount of memory for storing values of **d** vector for each sub-partition which will never be accessed in sparse matrix. In particular, saving a full copy of **d** vector for each sub-partition results in usage of a total

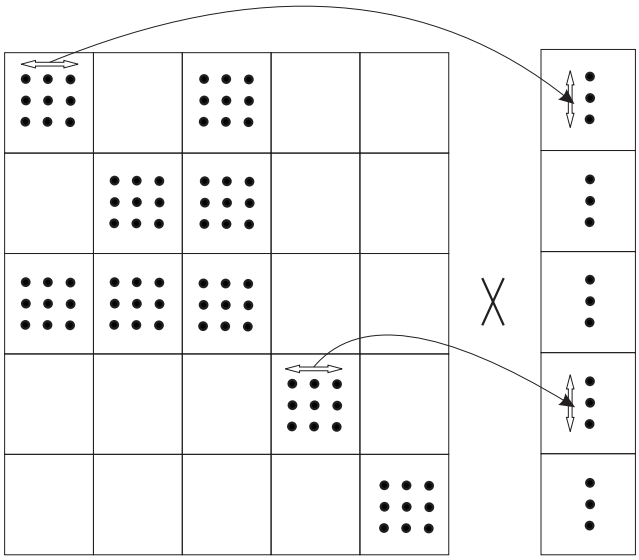

Figure 6.4: Performing sparse matrix by vector multiplication for rows 0 and 9 concurrently demands reading data from different locations of **d** vector

number of M9K memories for **d** vector (d-M9ks) given by:

$$number of d - M9Ks = 3 * 3 * number \\ of sub - partitions * number of FPGAs \tag{6.1}$$

This prevents scaling number of FPGAs by using more and more M9ks as the number of FPGAs increases.

To avoid storing multiple full copies of **d** vector, a custom storage format, Sparse Matrix Vector Indexing Scheme (SMVIS) is introduced in this thesis. In this format the **d** vector corresponding to each subpartition is represented by 2 vectors, Val and Ind. These two vectors are formed by the following algorithm:

We start from the left most column of the sparse matrix which is going to be multiplied by a full vector and go column by column.During this procedure, a non-zero value exists in a column, the number of the column will be stored in Ind

vector and the value of the corresponding value of **d** vector will be stored in the Val vector. This search will continue until the last column of the matrix.
The following example shows how this method works for the matrix and vector depicted in Fig. 6.5.

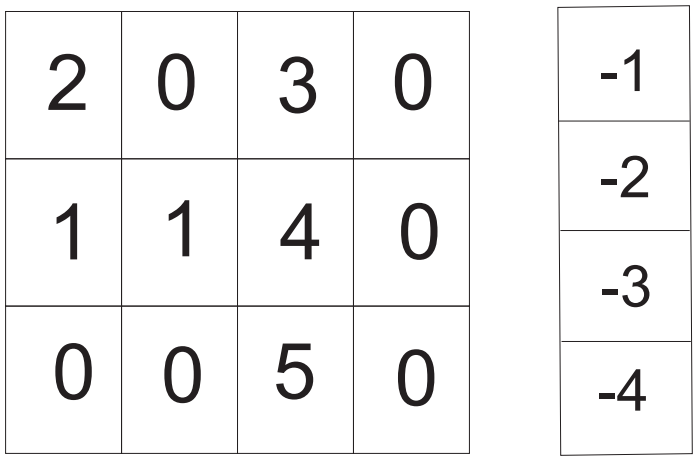

Figure 6.5: The sample matrix and vector used to demonstrate SMVIS format

- Val:[-1,-2,-3]
  Ind:[0,1,2]

Using SMVIS format for each sub-partition, we only have to store the elements of **d** vector which correspond to non-zero elements of K matrix in that sub-partition. In our implementation of the CG on 4 FPGAs each having 42 sub-partitions and each handling 1536 nodes (K matrix of size 18000*18000), SMVIS reduces M9K usage on each FPGA for storing **d** vector from $42*4*3*3+4*3*3 = 1548$ to $42 + 42 * 3 + 4 * 3 * 3 = 204$ M9ks. It is worth mentioning that each EPS3E110 has a total number of 639 M9K embedded memory units.

6.2.2 Memory Architecture

In order to solve the system of equations $\mathbf{K} * \mathbf{x} = \mathbf{b}$ using the CG algorithm, the following vectors have to be stored in embedded memory blocks of the FPGA: **b**, **x**, **r**, **Kd** and Val vectors for the CRS representation of K matrix (NZ_values), Ind vector for CRS representation of K matrix ($NZ\_indices$), Val and Ind vectors for SMVIS representation of d vector for each sub-partition (**d** and $d\_indices$), and a full copy of d vector (d_full).

As explained in Chapter 5, the maximum number of nodes handled on each FPGA in this design is 1536. As our mesh is defined in a 3D domain, each node of the mesh will have three corresponding values in vector **b** for x, y and z coordinates and a 3*3 corresponding block in **K** matrix. Therefore, all vectors on each FPGA would have $1536 * 3$ elements for x, y and z coordinates of 1536 nodes except $NZ\_Indices$ and $d\_indices$.

In order to increase the memory bandwidth for performing SpMxV and vector operations (vector by vector multiplication and vector updates), the vector for each dimension is stored in a separate memory block like the one in Fig. 6.6. In other words, each of these vectors is stored in three memory blocks, for the x, y and z components respectively.

6.2.3 Memory Architecture for Storing Vectors

Vectors **b**, **r** have similar memory structures depicted in Fig. 6.6.

Each of these vectors utilizes three memory blocks, for x, y and z coordinates,

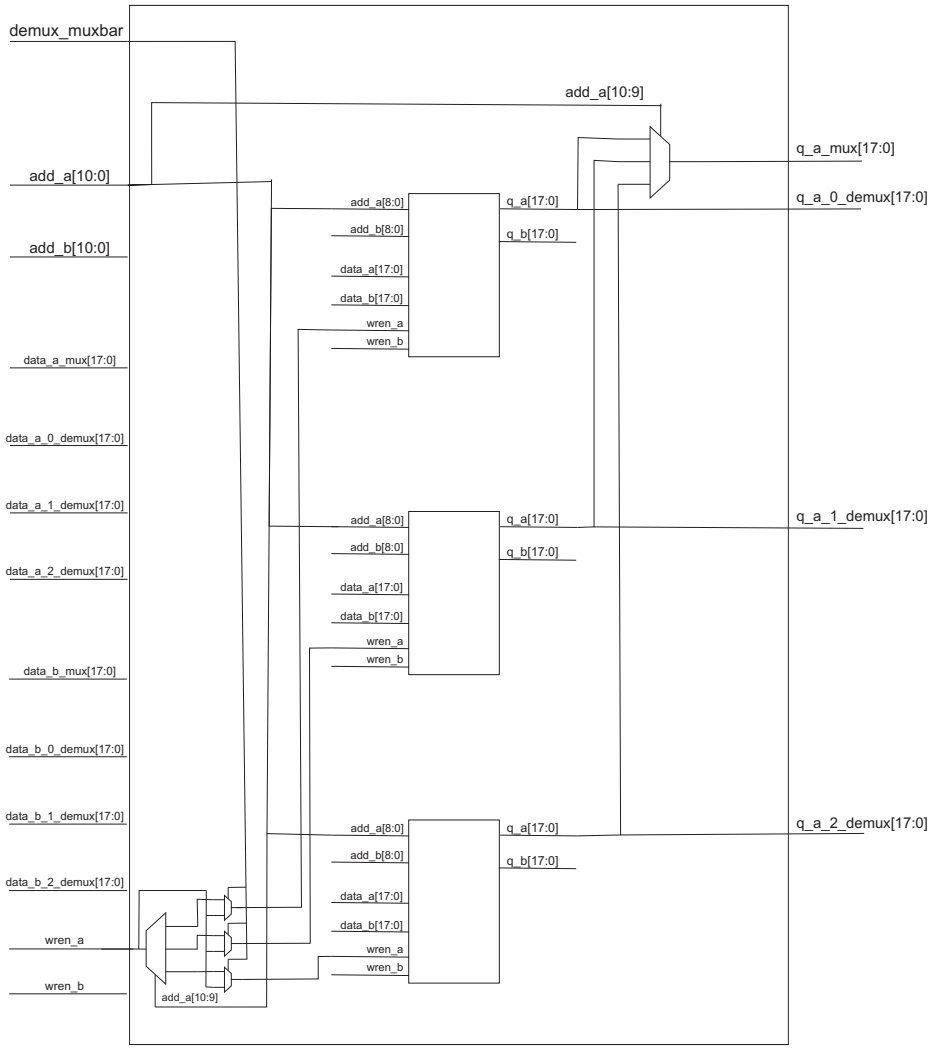

Figure 6.6: Memory architecture for one dimension (out of x, y or z) of vectors, wiring for port a (inputs and outputs with ”_a” term in their names) has been depicted in this figure and port b has similar wirings. Ports with names including ”data_” are input ports and those with term ”q_” are output ports

where a memory block consists of three M9ks, each configured as 512 words of 18-bits. The three M9ks have been multiplexed such that they work as one memory block with 1536 words depth. As illustrated in Fig. 6.6, this memory block also has demultiplexed inputs and outputs, denoted names including term "_demux" such that while performing vector updates, the read/write operation is performed concurrently for these three memory blocks, reducing the time needed to update the vectors. depth. As illustrated in Fig. 6.6. In this figure, the inputs and outputs with the term "_mux" included in their names denote these multiplexed inputs and outputs.
Vectors **x** and **Kd** have the same architecture as **r** and **b** vectors unless theirs data width is 36 bits instead of 18 bits depicted in Fig. 6.6.

Moreover, on each FPGA, we have to store a **d** vector for each sub-partition using SMVIS format. Using the SMVIS format for vector storage for in SpMxV multiplication, the length of vector will always be equal or less than the number of non-zeros in the matrix. Hence after allocating three M9ks for storing non-zeros of each sub-partitions as described in Section 6.2.4, allocating three M9ks for **d** vector corresponding to each sub-partition would be reasonable. Taking into account that this vector will also have three elements for x, y, and z coordinates, we will need 3*42 M9ks for storage of **d** vector.

Another vector that we have to store on each FPGA is a full copy of **d** vector (d_full). This vector is needed for calculating $d^T * kd$ in Step 11 of the CG algorithm in Table 5.1. The full copy of **d** vector on each FPGA is also required for vector updates in Steps 15, 16, 17 and 22 of the algorithm. Vector d_full will have a similar structure to that of b and r vectors with the difference that it utilizes 12 M9ks

(number of FPGAs*3) in order to store the union of d vectors corresponding to all FPGAs.

Finally, vector $NZ\_indices$ whose architecture will be later described in Section 6.2.5, and employ one M9k with 512 words depth and 18-bit data width for storing $d\_indeces$.

Table 6.2, summarizes the memory allocation for all vectors needed to be stored for the CG implementation on each FPGA.

Vector	M9K Usage	M144 Usage
b vector	3*3=9	0
x vector	3*3*2=18	0
r vector	3*3=9	0
kd vector	3*3*2=18	0
NZ_values vector	42*3*3=378	0
$NZ\_Indices$ vector	0	42/3=14
d vector	42*3=126	0
$d\_Indices$ vector	42	0
d_full vector	4*3*3=36	0
Total memory usage	636	14

Table 6.2: A summary of memory usage by different vectors on each FPGA for the multi-FPGA architecture with a maximum capacity of 1536 nodes on each FPGA

6.2.4 Storing K Matrix: Non-zero Values

K matrix is stored using a modified CRS format. In our scheme two vectors are used to represent K matrix, NZ_values vector which is stored in NZ-M9ks and

$NZ\_indices$ vector, which is stored in 14 m144ks. NZ_values vector for each sub-partition require nine M9ks. A connection between nodes numbered i and j in the 3D mesh results in a 3*3 block of non-zeros in the K matrix, i.e. the elements shared among rows $3i-2$, $3i-1$ and $3i$ and columns $3j-2$, $3j-1$ and $3j$. The **K** matrix structure will be further discussed in Section 6.3.2.

We employ this structural property of K matrix to increase the bandwidth for reading values from K matrix as well as speed of the SpMxV in the following way:

Since the non-zero values of the K matrix appear in 3*3 blocks as depicted in Fig. 6.7, all non-zero elements would have indices in the form of (k,l) where $(3*i-2 \leqslant k \leqslant 3*i)$, $(3*j-2 \leqslant l \leqslant 3*j)$ and $(1 \leqslant i,j \leqslant n)$ resulting in 9 different combinations of indices.

Therefore, for each sub-partition we will utilize nine M9ks, corresponding to these

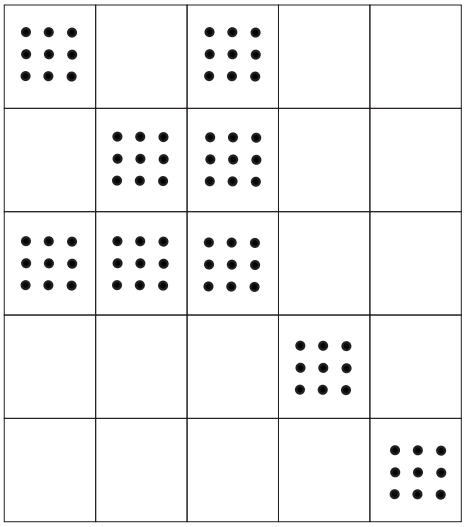

Figure 6.7: 3*3 non-zero blocks for a matrix representing a mesh with 6 nodes

nine index combinations. In this memory architecture as illustrated in Fig 6.8, each of the three consecutive rows ($3i-2$, $3i-1$ and $3i$, $(1 \leqslant i \leqslant n)$) would have a

bundle of three M9ks for storing its non-zero values.

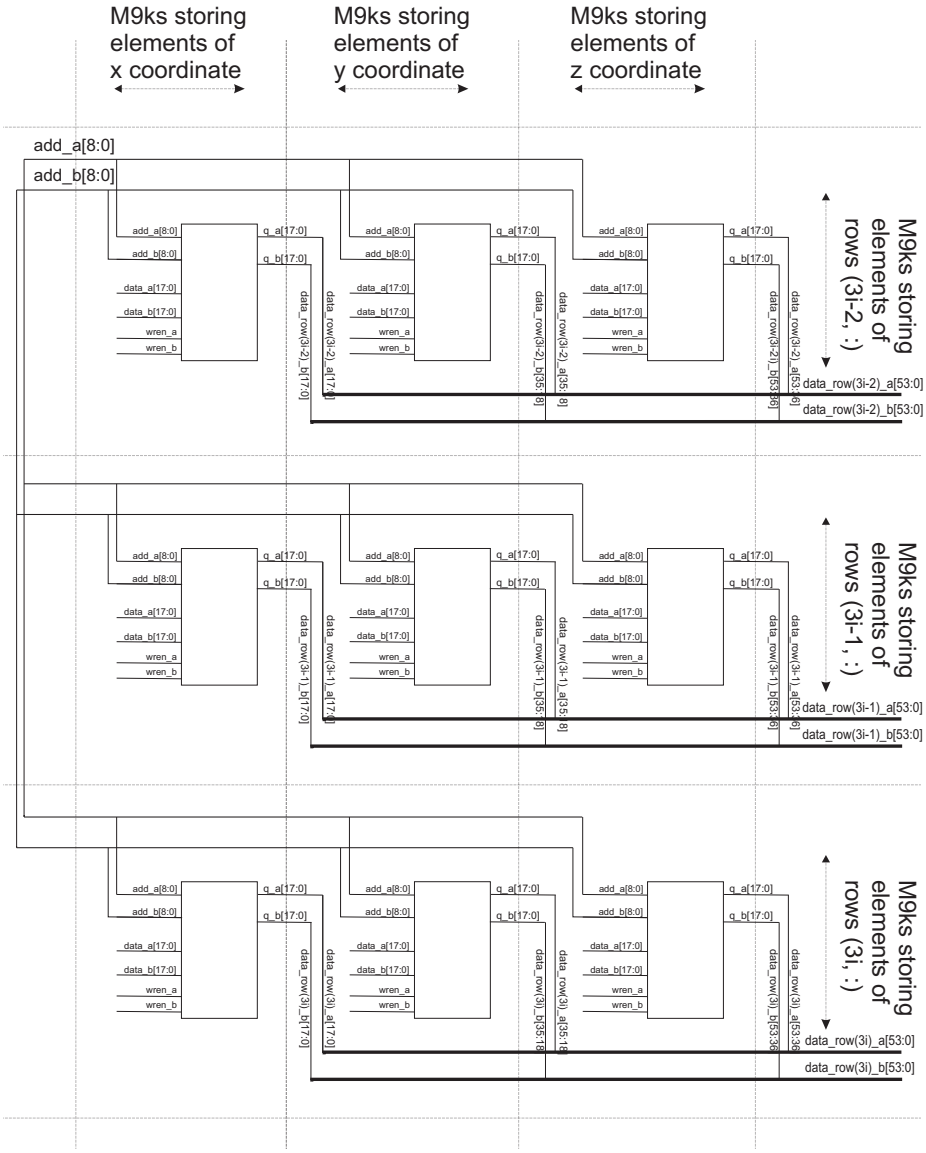

Figure 6.8: Memory architecture for storing NZ_values vectors of each subpartition

In addition, we perform the SpMxV for each three consecutive rows $3i-2$, $3i-1$ and $3i$ ($1 \leqslant i \leqslant n$) concurrently. Therefore three sets of xyz non-zero M9ks are needed to store non-zeros for three rows of each sub-partition, which would be

three M9ks.

6.2.5 Storing K Matrix: Non-zero Indices

As described in Section 6.2.1, to store K matrix we need another vector called Col, which is used to drive the address lines of M9ks storing **d** vector while performing the SpMxV. Recalling that we are utilizing another storage format for **d** vector, the elements of d_full vector are stored in **d** vector of each sub partition in a particular order depending on the location of non-zero values of that subpartition.

Hence, for addressing the proper element of d vector to be multiplied by a non-zero value of the sparse matrix, the values stored in $nz\_indices$ of M144k Rams are not really the values stored in Col vector of the CRS format, instead, they are the mapped version of those values to make them correspond to the right address of the M9K storing **d** vector. For example when we have a non-zero value in Column 5 of the matrix, instead of storing 5 as the corresponding index for that non-zero in $NZ\_indices$ vector, by using $d\_indices$ vector, we find the address of M9k storing **d** vector which stores the value of d_full vector corresponding to that non-zero value and store it in $NZ\_indices$ vector. This mapping is done off-line after performing CRS and SMVIS on K matrix.
After obtaining the values for $NZ\_indices$ of all sub-partitions, we store them in M144k Rams using the following architecture.

In the general form of this architecture with L number of sub-partitions, the number of M144ks used to store $NZ\_indices$ equals $\frac{L}{3}$. In particular, there are 14 M144ks in our implementation with 42 sub-partitions on each FPGA.

The reason for this is getting enough bandwidth from M144ks in the current design with EPS3E110 FPGA devices, which is described in more details as follows.

As will be explained in Section 6.3.2, while performing SpMxV, at each clock cycle we need to read two indices for **d** vector of each sub-partition. Therefore we need to read $L * 2$ indices in each clock cycle to feed the address lines for **d** vectors of all L sub-partitions. Thus in the case of using a two-port memory block for each sub-partition, a total of L M144ks. On the other hand, our calculations show that the maximum number of sub-partitions on a design that fits on a Stratix EP3SE110 is 42. This requires 42 M144ks, while on each device we have a total of 14 M144ks. Therefore, to have enough bandwidth to read sufficient amount of data in each clock cycle, by utilizing 14 M9K Rams, we store three index values for three sub-partitions in each address of one M144k. To accomplish this, the following memory architecture for M144ks is proposed.

The i^{th} M144k keeps the Col values for three sub-partitions, $3i + 2$, $3i + 1$ and $3i$ as depicted in Fig. 6.9. Thus while performing the SpMxV we can read $2*3*14 = 2*42$ values from M144ks and hence $3 * 2 * 42 = 6 * 42$ values from d M9ks for 42 sub-partitions or 6 values for each sub-partition.

The proposed memory architecture enables a fast and memory efficient implementation the CG algorithm. Using the proposed memory architecture, each Stratix III EPS3E110 can process up to 1536 (3*512) nodes, when each row of the sparse matrix can have up to an average number of 39 non-zeros in each row. This is how we obtained this number of non-zeros(39): 1536 nodes are processed on each FPGA, so each partition has to process an average number of $1536/42$ 37 nodes. According to the memory allocation for NZ-M9ks, the non-zero values of

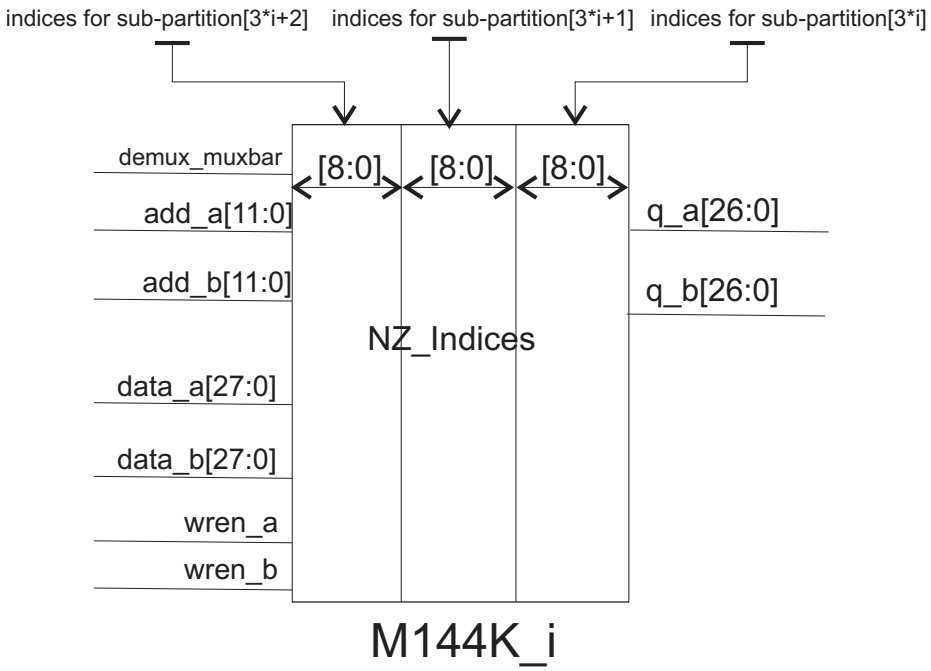

Figure 6.9: Memory architecture for $nz\_indices$ storage in M144k rams. Ports including the term "data_" in their names are input ports and those with "q_" are output ports

these 37 nodes has to be stored on three M9ks which results in the capacity for storing $3 * 512/37$ 39 non-zeros per row of the **K** matrix.

6.3 Implementing CG Algorithm On Multiple FPGAs

The implementation of the CG on each FPGA follows the steps in the algorithm in Table 5.1, except in Steps 10 and 11, as well as Steps 15, 16, and 17. Due to no data dependency among many of the steps of the algorithm, they can be performed concurrently on all FPGAs. In this section the hardware architecture designed for performing these steps will be described in details.

6.3.1 Sparse Matrix by Vector Multiplication

The sparse by matrix multiplication unit is the most resource and time consuming block in the implementation of the CG algorithm. To minimize the memory usage by this block we employ the storage schemes in store d vector and K matrix. We also propose a highly parallelized scheme to increase the throughput of the SpMxV unit in order to reduce the time needed for completing the operation.

6.3.2 A Highly Parallelized Scheme for SpMxV

In the proposed architecture, the SpMxV kernel exploits four different levels of parallelization. The first and second levels of parallelization (PL1 and PL2), described in Chapter 5, are created by employing multiple FPGAs and dividing the corresponding portion of K matrix on each FPGA to some sub-partitions. It is worth noting that in this design the number of sub-partition in $PL2$ is variable and is limited by the available memory and multiplier resources on device. For example for Implementation on EPS3E110, number of sub-partitions on each device can be any value ≤ 42.
Further parallelism can be achieved by carefully investigating the data structure of K matrix as follows.

- Third level of parallelization (PL3): K represents the stiffness matrix of the 3D FE mesh of the soft object to be modeled. Connections between each pair of nodes would result in a $3 * 3$ block of non-zero elements in the matrix as illustrated in Fig. 6.7.

Consequently in SpMxV operation in line of 10 of the CG algorithm in Table 5.1, these three rows will fetch the same values of **d** vector. Therefore we can do the multiplication for these three rows in parallel resulting in the third level of parallelization (PL3). Moreover, this special structure of K matrix decreases memory usage by enabling us to store non-zero indices for only one out of each three consecutive rows of K matrix.

- Fourth level of parallelization (*PL4*): Finally, we can increase the parallelization further by performing the multiplication for 6 non-zero values of each row by **d** vector at the same time. The number 6 has been chosen due to a memory bandwidth limitation and is the maximum number of non-zero elements from each row and from **d** vector that can be read at one clock cycle.

Taking these into account and to use as many DSP blocks in parallel as possible, the data path for matrix by vector multiplication is designed as illustrated in Fig. 6.10.

$K * d$ data path

In the proposed memory architecture, during the SpMxV, at each clock cycle for each row which is undergoing multiplication by **d** vector, a maximum of six non-zero values from **K** matrix and six corresponding values from **d** vector can be read. Hence considering the $PL4$, each row requires six $18 * 18$ multipliers at each clock cycle for performing multiplication for these six pair of values from the row and the **d** vector. Also, as already mentioned in Section 6.1, to utilize all available 896 $18 * 18$ multipliers, they should be used in a $mult\_add$ configuration as depicted in Fig. 6.1.

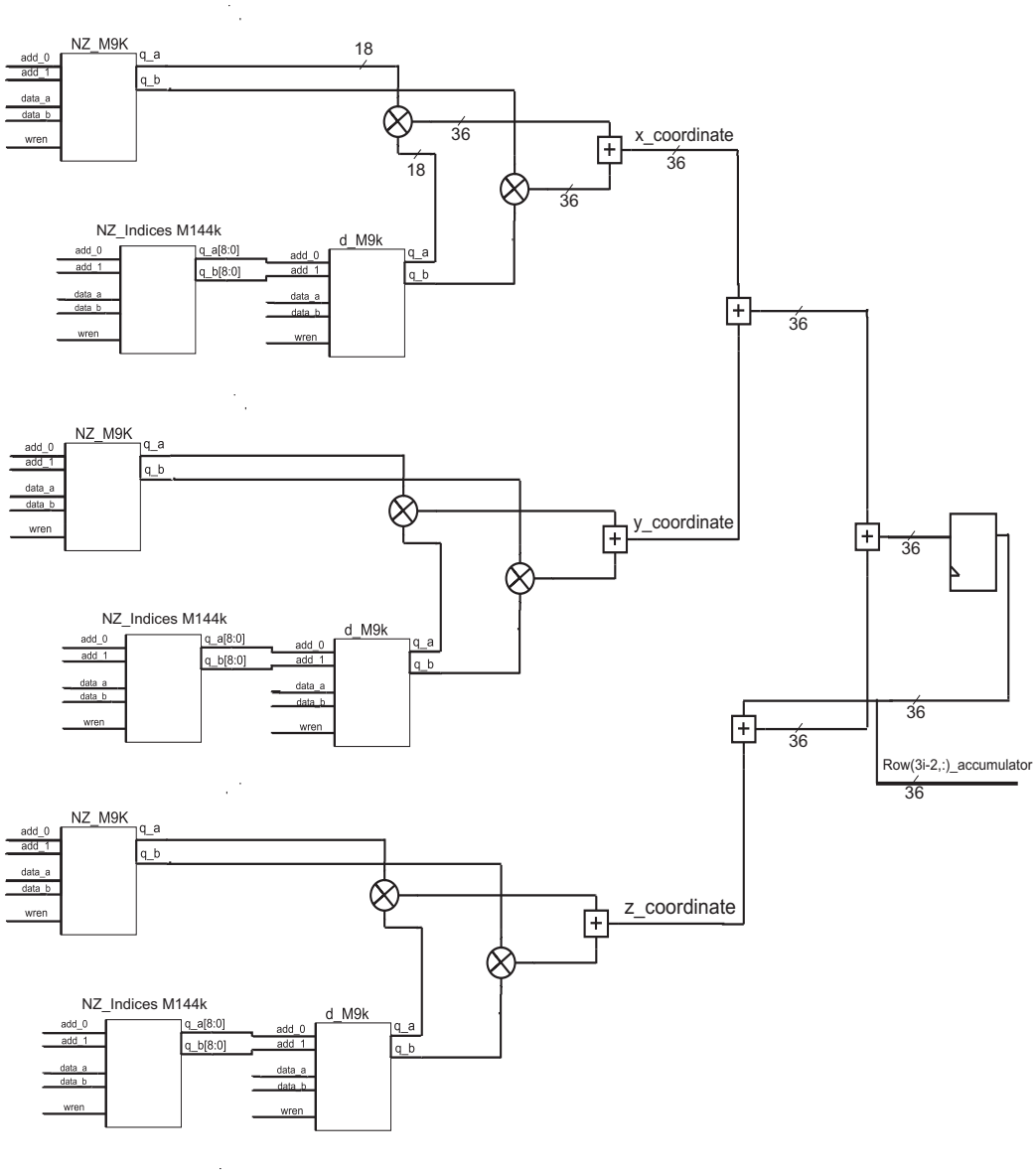

Figure 6.10: Data path for K*d multiplication for row (3i-2,:), the same architecture exists for rows (3i-1,:) and (3i,:) for each sub-partition. In this figure, d_M9k refers to M9ks storing **d** vector values for sub-partitions.

Saving K*d Results

The K*d multiplication is designed such that at each clock cycle, the multiplication for 42 rows of 42 sub partitions is done concurrently. For saving results of each row by vector multiplication two scenarios could be considered. One is that all sub-partitions start the multiplication for one row together and when all of them have finished, the whole process is stopped until the results of all 42 sub-partitions are stored. The multiplication for the next row can resume afterwards. In the second scenario is that each sub-partition acts independently while performing multiplication and the results of each sub-partition would be buffered and stored during the K*d multiplication process. The second scenario is more complicated and needs more hardware resources (more LUTs and registers for implementing shift register and buffering data). However it increases the number of nodes that can be processed on the device for the following reason.

For an efficient use of NZ M9ks, matrix partitioning is performed such that non-zero values are evenly distributed in $N * 42$ sub-partitions where N is the number of FPGAs. Depending on the pattern of non-zeros in the K matrix the number of rows of the matrix in different sub-partitions will be different. For example, with our meshing algorithm applied to a sphere, the pattern of the non-zeros in the K matrix is as illustrated in Fig. 6.11. It can be seen that the sparsity of the last sub-partition is much more than those in the middle part of the matrix. For example the number of rows for the last partition of last FPGA is about 12 times more than the average number of nodes in other sub-partitions for a the stiffness matrix of a sphere with 569 nodes used in our tests. (this increases with an increase in the number of nodes) greater than the maximum number of rows in all other

partitions. Hence using the first method for saving results would increase the time of the whole process in three ways,

1. The time needed for the multiplication of i^{th} row of each partition would be equal to the time needed for the multiplication of the i^{th} row of the sub-partition with maximum number of nonzero element.

2. After finishing multiplication for a row we need to stop the multiplication process and save the 42 results generated by 42 sub- partitions.

3. Due to inequality in the number of rows in different sub-partitions, the most sparse partition of the matrix will have less non-zeros in each row and more number of rows. Therefore it will be halted by other partitions after finishing the multiplication for each of its rows and when other partitions are done the whole process would be blocked until this partition finishes multiplication for all of its rows.

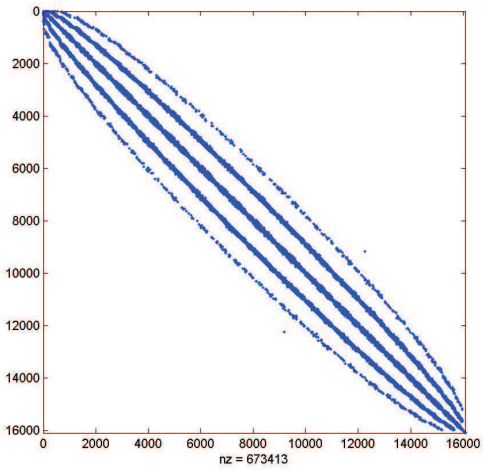

Figure 6.11: The K matrix arising from performing FE analysis on an sphere in our tests

According to this reasoning the second method for saving multiplication results is employed in this thesis. In this scenario, a 42 bit register $(write\_req)$ keeps track of all partitions which have finished multiplication of one row and another module at each clock cycle goes through this list and finds the next partition for which the results should be stored. Then a third block writes the results on XYZ M9k (XYZ M9ks store the **K*d** multiplication results).

6.3.3 Vector by Vector Multiplication

An increase in the number of nodes on each FPGA also increases the length of vectors on each of them, elongating the time needed for vector by vector multiplications linearly by the number of nodes on each FPGA. This can slow down the iterations of the CG and eventually violate the real-time response requirement.
In order to improve the scalability of the design, vectors are stored such that depending on an input named *demux/mux bar*, as illustrated in Fig. 6.6, they can be utilized both as one dual port memory block with 1536 words depth or three dual port M9ks with 512 words depth. The latter gives us more bandwidth for read/write operations on the vectors. In vector by vector multiplication modules, thesememory units are used in unpacked mode. In this case vector by vector multiplication will take 2*512 clock cycles regardless of length of vector.
The memory architecture for vectors in our implementation with vector length of 1536 will save $4*512$ clock cycles in vector vector multiplication time. This is rather significant considering that about $17 * 512$ clock cycles are needed to complete one iteration of the CG iteration.

6.3.4 Inter-FPGA Communication

The communication among the FPGAs is arranged in a ring configuration shown in Fig. 6.12. This configuration was chosen in order to obtain the maximum bandwidth possible for inter-FPGA communications, considering the physical connections between FPGAs on the Gidel ProcStar III board as depicted in Fig. 6.2. This parallel communication scheme utilizes a bandwidth, given in bits/clock cycle by Equation 6.2 if 18-bit data words were used to represent $\mathbf{d}$ vector.

$$N \times (3 \times 18 + [\log_2(\text{number of nodes})]) \tag{6.2}$$

In this formulation $3*18$ stands for the bits needed for sending three elements of $\mathbf{d}$ vector for three coordinates and $\log_2$(number of nodes is the bandwidth needed of sending index for these three values.

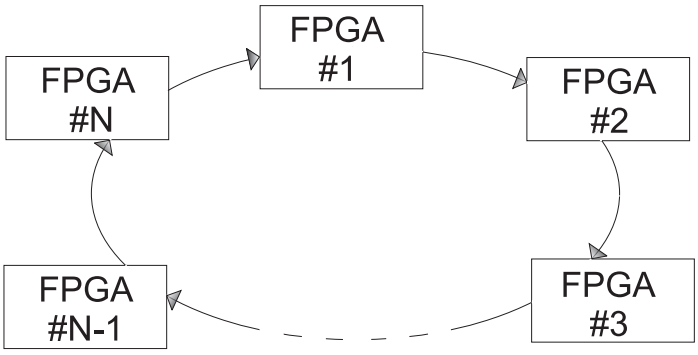

Figure 6.12: The communication scheme in multi-FPGA architecture. Arrows show the direction of the data flow.

There are essentially three types of data exchange in the distributed CG algorithm depicted in Table 5.1. The partitions of the vector $\mathbf{d}$ are updated on each FPGA in Steps 4 and of 23 of the algorithm in Table 5.1 according to the data flow

	FPGA #1	FPGA #2	...	FPGA #N
1)	$send\ \mathbf{d^1}, recieve\ \mathbf{d^N}$	$\longrightarrow send\ \mathbf{d^2}, recieve\ \mathbf{d^1}$	$\longrightarrow ... \longrightarrow send\ \mathbf{d^N}, recieve\ \mathbf{d^{N-1}}$	
2)	$send\ \mathbf{d^N}, recieve\ \mathbf{d^{N-1}}$	$\longrightarrow send\ \mathbf{d^1}, recieve\ \mathbf{d^N}$	$\longrightarrow ... \longrightarrow send\ \mathbf{d^{N-1}}, recieve\ \mathbf{d^{N-2}}$	
⋮				
N-1)	$send\ \mathbf{d^3}, recieve\ \mathbf{d^2}$	$\longrightarrow send\ \mathbf{d^4}, recieve\ \mathbf{d^3}$	$\longrightarrow ... \longrightarrow send\ \mathbf{d^2}, recieve\ \mathbf{d^1}$	

Table 6.3: Algorithm used for communicating updated $\mathbf{d}$ vector.

	FPGA #1	FPGA #2	FPGA #3	...	FPGA #N
1)	–	$send\ scalar_2 \longrightarrow$	$receive\ scalar_2$	...	–
2)	–	–	$send\ (scalar_2 + scalar_3) \longrightarrow$	...	–
⋮					
N-2)	–	–	–	$... \longrightarrow receive\ \sum_{i=2}^{N-1} scalar_i$	
N-1)	$receive\ \sum_{i=2}^{N} scalar_i$	–	–	...	$send\ \sum_{i=2}^{N} scalar_i$

Table 6.4: Algorithm used for accumulating a scalar partially calculated on FPGAs and send it to FPGA#1.

presented in Table 6.3. Since the matrix partitions per FPGA are balanced in terms of non-zero components, it takes almost the same time for all the FPGAs to reach to the state of updating vector $\mathbf{d}$ in line 23 of the algorithm. The second type of data communication occurs in Steps 7, 12 and 18 of the algorithm where the partially calculated results of the inner vector by vector multiplications are exchanged among the FPGAs using the scheme depicted in Table 6.4. Finally in Steps 8, 14 and 19, the scalar values rrn, α and β which are calculated in FPGA#1 are transmitted to all other FPGAs as illustrated in Table 6.4. The resource cost and computation and communication time for these scalar values are relatively negligible.

In the proposed scheme, as shown in Fig. 6.13, each FPGA requires only a portion of vector $\mathbf{d}$ that is associated with nonzero elements of its corresponding part of matrix $\mathbf{K}$. This can reduce the communication time for $\mathbf{d}$ by transmitting only that portion of it in each FPGA.

	FPGA #1		FPGA #2		...		FPGA #N
1)	*send scalar*	$\longrightarrow$	*receive scalar*		...		–
2)	–		*send scalar*	$\longrightarrow$	...		–
⋮							
N-1)	–		–		...	$\longrightarrow$	*receive scalar*

Table 6.5: Algorithm used for distributing a scalar from FPGA#1 to all other FPGAs

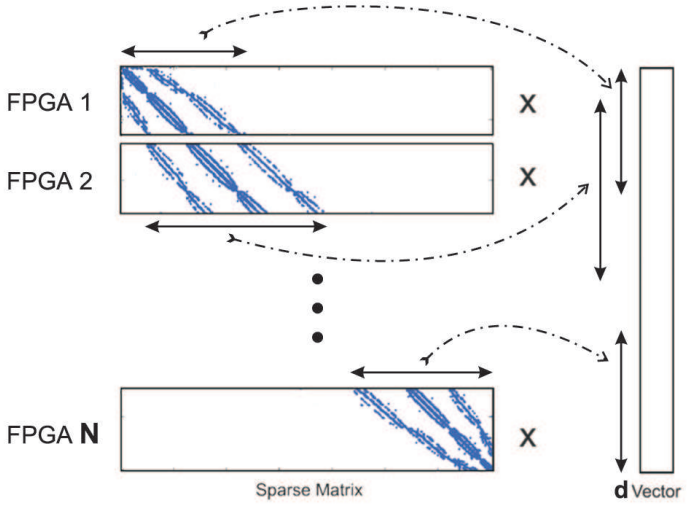

Figure 6.13: Portions of **d** vector needed for each FPGA for performing **Kxd** multiplication **d** vector

In the current design, due to ProcStar III board limitations, for communications between FPGAs, we are using a parallel communication with a bandwidth of 14Gbps for each FPGA.

6.4 Resource Usage

The resource usage of each FPGA in the current design is 82% of logics, 90% of the memory bits and 95% of the DSP blocks on device.Fig. 6.14 depicts the compilation result generated by Quartus II software for FPGA #1.

Flow Status	Successful - Fri Aug 14 18:25:56 2009
Quartus II Version	8.0 Build 215 05/29/2008 SJ Full Version
Revision Name	fpga1
Top-level Entity Name	fpga1
Family	Stratix III
Device	EP3SE110F1152C2
Timing Models	Preliminary
Met timing requirements	No
Logic utilization	85 %
Combinational ALUTs	68,577 / 85,200 (80 %)
Memory ALUTs	0 / 42,600 (0 %)
Dedicated logic registers	19,767 / 85,200 (23 %)
Total registers	19972
Total pins	736 / 744 (99 %)
Total virtual pins	0
Total block memory bits	7,402,864 / 8,248,320 (90 %)
DSP block 18-bit elements	852 / 896 (95 %)
Total PLLs	2 / 8 (25 %)
Total DLLs	0 / 4 (0 %)

Figure 6.14: Compilation report for an FPGA in our current design

The list of significant resources used by the design is as given in Table 6.6 and Table 6.7. Table 6.6 lists design units with major DSP usage and number of $18 * 18$ multipliers employed by each of them. As expected, the most of the multipliers are dedicated for performing SpMxV. It is worth mentioning that the multipliers used for the SpMxV and those employed for calculating $d^T * Kd$ work together in parallel while performing Steps 10 and 11 of CG algorithm in Table 5.1. Table 6.7, shows the usage of embedded memory blocks by different vectors in the design.

module	18*18 multipliers
K*d	756=6*3*42
$d^T * K * d$	12=3*4
vector*vector	36=2*3*3*2
Total number of multipliers	804

Table 6.6: 18*18 bit multiplier blocks used on each FPGA

	number of M9ks	number of M144ks
NZ_values	378=3*3*42	0
d vectors	126=3*42	0
d_full	36=4*3*3	0
$d\_Indices$	42	0
b vector	9=3*3	0
x vector	9=3*3	0
r vector	18=2*3*3	0
Kd vector	18=2*3*3	0
$NZ\_Indices$	0	14
Total memory usage	636	14

Table 6.7: Embedded memory usage by different vectors on each FPGA

6.5 Timing Analysis

The time required for performing one iteration of the CG iteration is mainly spent on three steps of the algorithm: K*d multiplication, Vector By Vector Multiplication, and **d** Vector Communication. In the following we will discuss each of these steps. The calculated times here are for the design with four FPGAs, each handling up to 1536 nodes and with 42 sub-partitions.

K*d multiplication

The time needed for K*d multiplication depends on the size and sparsity of the matrix. However, employs 42 sub-partitions the current architecture which is bounded by not the rate **Kd** vector elements become ready, but by the rate at which the results are written to **Kd** vector. Therefore, the time needed for performing K*d multiplication is almost equal to the time needed for writing 1536 values to **Kd** vector which is 1536 clock cycles.

Vector By Vector Multiplication

As discussed in Section 6.3.3, the time required for two vector by vector multiplication steps in Table 5.1 is 2*2*512 clock cycles. The first is for the vector by vector multiplication in Steps 15, 16 and 17 of the algorithm which are carried out simultaneously and the second is for Step 22.

d Vector Communication

The communication time for scalar value dkd, α, rrn and β are negligible as sending each scalar takes three clock cycles including the time needed for hand shaking. So we will just consider **d** vector communication time.

For our current implementation on ProcStar III board, in a worst case scenario when the design is working for 4*1536 nodes this time will be 1536 clock cycles for each of three communication steps in Table 6.3. Therefore, the total time needed for communication among the FPGAs would be 3*1536=9*512 clock cycles at each CG iteration. This can be compared to 1536(for SpMxV)+2*2*512(for

vec*vec)=7*512 clock cycles for all other tasks in one CG iteration.The time needed for communications will increase almost linearly with increase in the number of FPGAs. For example the total time for one CG iteration while using 10 FPGAs would be 9*1536(comm.time)+7*512 clock cycles. Therefore, the communication time will be the limiting factor for increasing the number of FPGAs in this implementation.

In general with the current bandwidth for parallel communication among FPGAs , the total number of clock cycles for performing one CG iteration will be (N-1)*1536+7*512 clock cycles. Here N is the total number of FPGAs.

To avoid increase in communication time by increase in the number of FPGAs, the communication bandwidth should increase linearly by the number of FPGAs. There are two options for communications, parallel and serial. The parallel communication bandwidth is limited by the number of I/O pins available on the FPGA which depends on the FPGA and its packaging. The Stratix III EP3SE110 FPGA device in 1152 pin package has 780 pins for inter-FPGA data communication [58]. Hence, we can have a maximum of 22 FPGAs in case of utilizing all 780 IO pins of the FPGA for communications. The other method for data exchange among FPGAs is serial communication. This can be attained by exploiting high speed serial communication capability on Stratix III devices.These devices provide up to 132 full duplex 1.25 Gbps true LVDS channels (132 Tx + 132 Rx) on the row I/O banks, which yields 165 Gbps bandwidth for communications.

Hence, in principle the design can be expanded on 44 FPGAs without increasing in the time required for performing CG calculations comparing to our current 4 FPGA design. It should be noted that the high-speed serial data transmission is

Step	Number of clock cycles
SpMxV	3*512
Vector operations	2*2*512
d vector communication	3*3*512
Total clock cycles for 1 CG iteration	8192

Table 6.8: Number of clock cycles required by most time consuming steps of the CG iteration operating for 6000 nodes.

not supported by Gidel Procstar III board.

Based on the implementation using parallel communication, for a maximum of 6000 nodes (18000*18000 K matrix), we will require a total time of $3*512(forSpMxV)+2*2*512(forvec*vecmultiplications)+3*3*512(forcommunications) = 16*512 = 8192$ clock cycles for each iteration of CG method. And as will be described the maximum frequency (f_{max}) of the circuit is 100 MHz, this will result in 0.08192 ms for each CG iteration.

In Table 6.8, the number of clock cycles for important steps of the CG algorithm for our implementation on four Altera Stratix III EP3SE110 devices are given. In this table, vector operation refers to vector updates and vector by vector multiplications performed in Steps 15, 16, 17 and 22 of the CG algorithm in Table 5.1.

The Critical Path and Maximum Clock Rate f_{max}

For the current design the f_{max} calculated by Quartus II software classic time analyzer is 100 MHz. This maximum frequency is determined by $\frac{1}{max(t_{pd})}$ where the $max(t_{pd})$ is the maximum propagation delay time for all pathes in the FPGA. For

this design the critical path (the path with the maximum propagation delay) is the path for K*d multiplication which is depicted in Fig. 6.10. This path contains three combinational circuits including an 18*18 multiplier, and three signed number adders. One of these adders performs 18-bit by 18-bit addition of results from multiplication of two consecutive non-zero values in x coordinate (or y), the second signed adder is utilized for addition of two 36-bit values from x and y coordinates, finally, a third signed adder sums up the result of the second adder, with the summation result of z-coordinate and "row(3i-2,:)-accumulator" as depicted in Fig. 6.10.

The hardware architecture proposed in this chapter has been successfully implemented and tested on a Gidel ProcStar III board hosting four Stratix II EP3SE110 devices. However, for processing more number of nodes, the design can be scaled up to be implemented on multiple FPGA devices or on newer devices with more hardware resources. In case of utilizing more powerful FPGAs, number of sub-partitions on each FPGA should be scaled up to avoid elongation in SpMxV operation time. This can be performed by increasing the number of sub-partitions which is defined as a constant in our HDL code.
In the following chapter the experimental results and performance of this architecture to accelerate CG computations is studied. To provide more practical results, our hardware-based CG solver has been integrated into a hardware-in-the-loop haptic simulator. Chapter 7 will present the set up for this haptic simulator for simulating interactions with a deformable object.

Chapter 7

Experimental Results

This chapter presents the experimental results for the implementation of the CG algorithm on a Gidel ProcStar III board, with four Altera Stratix III EP3SE110 FPGA devices. The performance of the new multi-FPGA architecture is compared with the single-FPGA design in [1] and some other conventional processors. The experimental setup for hardware-in-the-loop platform for real time haptic simulation of soft-object deformation will be also presented in this chapter.

7.1 Performance Evaluation

The proposed multi-FPGA hardware-based accelerator for the CG algorithm has been implemented on a Gidel ProcStar III development board with $N = 4$ Altera Stratix III EP3SE110 FPGA devices. The system can find the solution to the system of equations arising from a 3D FE mesh of up to 6000 nodes at an update rate of 400Hz. The current quad-FPGA implementation with $PL2 = 42$ utilizes 756 $18 * 18$ multipliers per FPGA for SpMxV multiplication. In addition 96 $18 * 18$

multipliers are used in parallel for vector-vector and scalar-vector multiplications in the implementation of the CG algorithm. This leads to an SpMxV kernel that operates at a rate of 302.4 Giga Operations Per Second (GOPS). In Table 7.1, the performance of the proposed architecture has been compared with three different microprocessors used in the experiments conducted by Goumas et al. [61] and also the single-FPGA design proposed by Mafi et.al in [1]. The last column of the table provides a comparison between the average results in the fourth column to the speed of computation in our proposed architecture. As illustrated by the comparison results, our new multi-FPGA architecture yields a high speed-up over the microprocessors. This is achieved due to the massive parallelism of computations using customized fixed-point implementation.

The improvement in the rate of operation compared to the single-FPGA design in [1] is due to the utilization of multiple FPGAs as well as the new SpMxV unit and the SMVIS format for **d** vector storage. These changes in the multiple-FPGA design will allowing a greater number of multipliers concurrently compared to the single-FPGA design.

Processor	Clock Speed	L2 Cache	Average operations per second	xSlower
Woodcrest	2.6 GHz	4MB	495.53 MFLOPS	x232
Netburst	2.8 GHz	1MB	297.88 MFLOPS	x387
Operton	1.8 GHz	1MB	273.72 MFLOPS	x421
Single-FPGA design by [1]	100 MHz	-	18 GOPS	x16.8
quad-FPGA architecture	100 MHz	-	302 GOPS	x1

Table 7.1: Comparative performance of SpMxV kernel on general-purpose CPUs [61], the single-FPGA design proposed in [1] and the proposed multi-FPGA hardware accelerator.

In the following, the experimental results form implementation of the final multi-FPGA architecture (Design III) are presented. For performing these tests,

the Gidel Procstar III board has been installed on PCI Express local bus of a PC, Therefore exchanging data with the FPGAs are performed via the PCI.

To establish numerical stability of our Fixed-point CG implementation, the Monte Carlo simulations has been performed using a moderately sized matrix $1707 * 1707$ with condition number of 2873. The simulation has been carried out for 30 runs of the algorithm with random initial x vector. The results of average relative error in **x** vector for different number of iterations in the CG algorithm is depicted in Fig. 7.1. According to these results, after performing 30 iterations of the CG algorithm the decrease in $||error||$ will not be significant by increase in number of iterations. The reason is that as the error becomes smaller, the elements of the **r** vector in the CG algorithm will become smaller, increasing the effect of quantization errors. Therefore 30 is a good choice for the number of iterations for the CG solver in terms of achieved accuracy and computational cost.

Fig. 7.3 shows the result of solving $K*x = b$ for K matrices with sizes $1707*1707$ (for 569 nodes), $7455 * 7455$ (for 2485 nodes) and $16095 * 16095$ (for 5365 nodes) respectively. Fig. 7.2 depicts the mesh associated with the third matrix with 5365 nodes. In these tests the initial value of **x** vector has been zero. Table 7.2 compares these three tests in terms of K matrix properties and accuracy of results.

Number of nodes in mesh	Size of K matrix	Condition number of K matrix	$\frac{\|\|error\|\|}{\|\|x\|\|}$
569	1707*1707	2.873*10^3	0.055
2485	7455*7455	1.972*10^5	0.064
5635	16095*16095	1.22*10^6	0.073

Table 7.2: A comparison between K matric properties and error in results for three tests of the CG solver

.

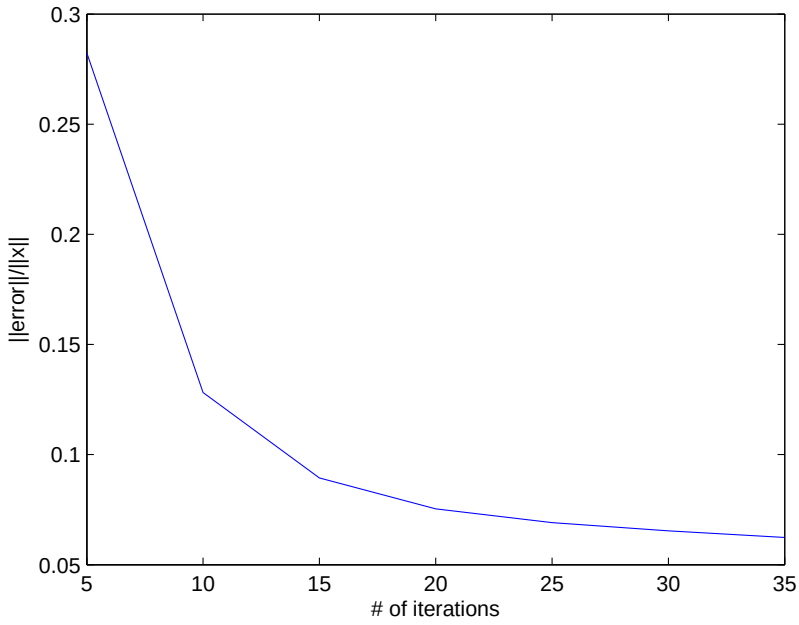

Figure 7.1: Changes in the norm of error in FPGA result as function of number of iterations

As expected, increasing the number of nodes will result in less accuracy of results. The reason is that by increasing the number of nodes, the condition number of the **K** matrix will grow, defecting the convergence of the CG algorithm. In the tests performed with three different matrix sizes, for the $1707 * 1707$ matrix, condition number is 2873, in the second case with the size of matrix equal to $7455 * 7455$, it equals $1.972 * 10^5$ and finally for 5365 nodes in the mesh, the condition number of stiffness matrix is $1.2218 * 10^6$. This issue can be mitigated by employing preconditioned CG algorithm (PCG) instead of CG, which will result in better convergence rate and therefore more accurate results specially for larger **K** matrices. A brief overview of PCG algorithm, the pseudo code for performing PCG using jacobi preconditioning, and its difference with CG in terms of implementation has been presented in Chapter 4. Furthermore, a detailed discussion on numerical stability

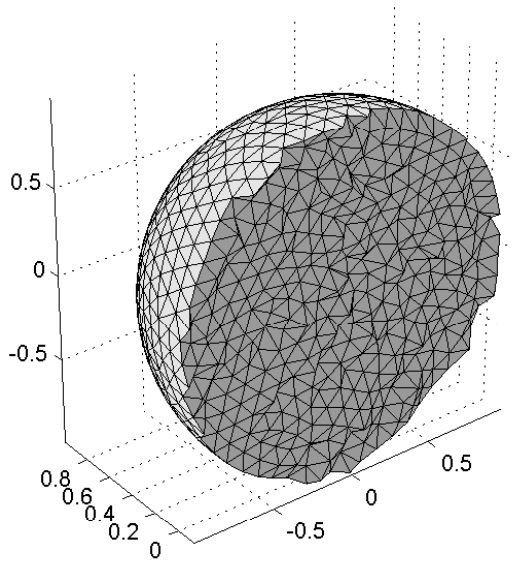

Figure 7.2: A transverse section of the spherical mesh associated with the largest matrix with 5365 nodes in our tests

and convergence of Preconditioned Conjugate Gradient algorithm(PCG) is offered in [48]. improving our iterative solver core from CG to PCG will be a matter of future work.

7.2 Hardware-in-the-Loop Haptic Simulation Platform

The proposed implementation of CG algorithm has been employed as the computational engine for modeling soft objects using FEM. The simulation platform used in our experiments is illustrated in Fig. 7.4. This haptic-enabled simulator consists of a custom 3DOF haptic interface, a Quanser QPA linear current amplifier, Quanser Q4 hardware-in-the-loop data acquisition board, a Gidel ProcStar III

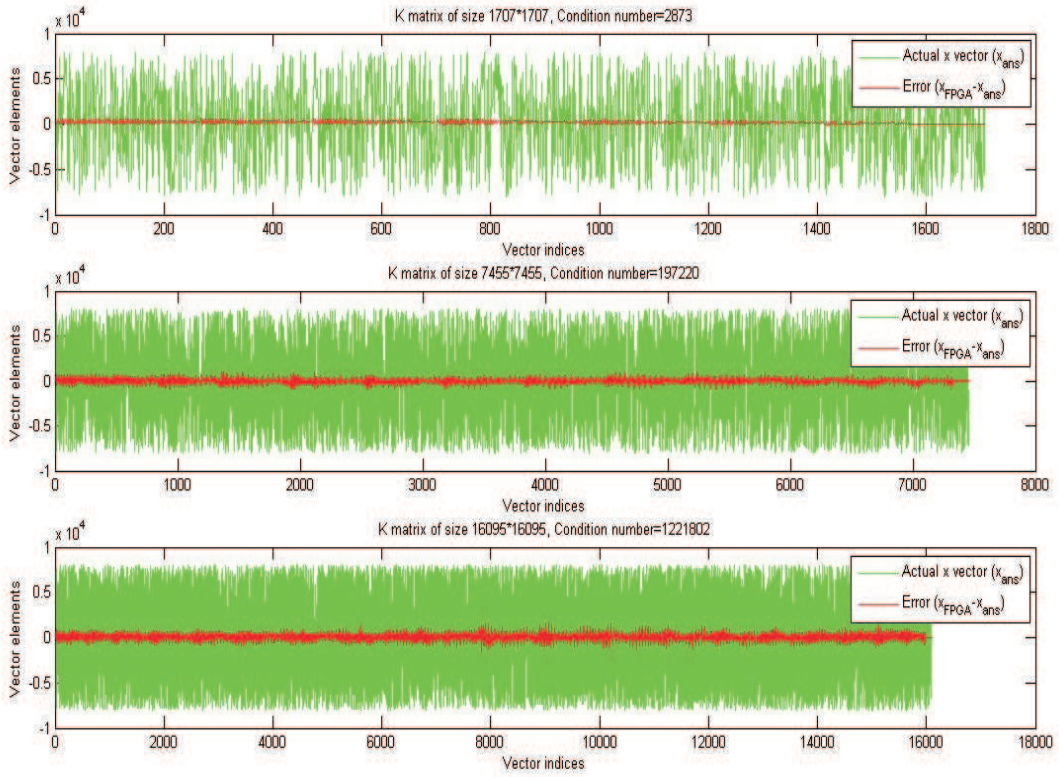

Figure 7.3: Error in FPGA solution compared to the real **x** vector for three different matrix sizes

development board with four Altera Stratix III EP3SE110 FPGA devices, and a 3.0GHz Pentium R(D) with 4.0 GB RAM using a Matrox Millennium PCIe P650 graphics card. The FPGA board communicates with the PC through the PCI bus interface to receive the vector **b**. The matrix **K** is assumed to be constant (except for changes due to a variable contact node for which changes in **K** matrix are performed inside the FPGA) and hence is loaded once at the beginning of the simulation process.

In this simulation when the operator holds the robot end-effector in his hand, this end-effector can be moved in the 3D space freely while it is out of the region where

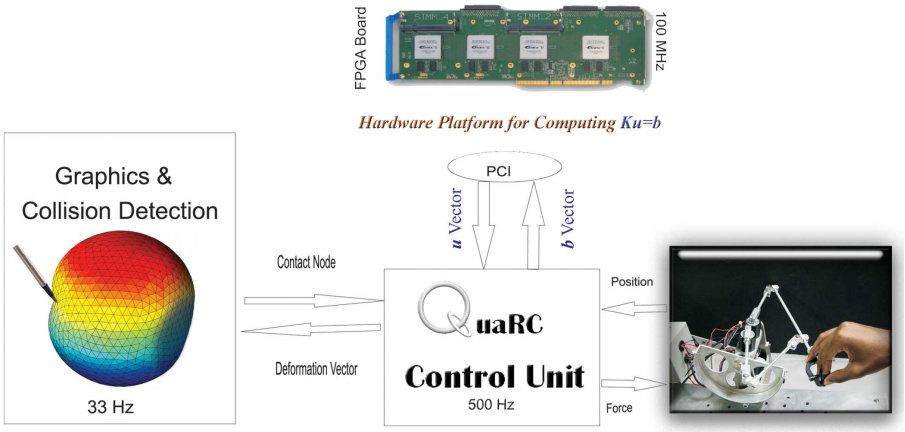

Figure 7.4: The block diagram of haptic-enabled simulator with hardware-based accelerator.

our virtual soft object is located. The position of the robot's end-effector is continuously sent to the PC with a frequency of 1KHz. Once the end effector enters the region defined as the internal space for the soft object, a collision detector unit will trigger the process of calculations of deformations of the object and amount of force feedback. At this point, the **b** vector and the number of the contact node would be sent to the FPGA via the PCI bus. Afterwards, the calculated **x** vector by the FPGA will be passed to the graphics unit for generating the new deformed shape of the object and the force vector will be sent to the robot.

The control process runs at a rate of 400Hz under Quanser's Windows real-time extension QuaRC and is the communication hub amongst the haptic interface, the hardware-based accelerator, and the graphics and collision detection unit.

OpenGL is used for graphics rendering and collision detection which are performed on the GPU of the graphics card at a rate of 33Hz. The results of haptic exploration experiments with both static and dynamic linear elastic FE models have been very encouraging. The system can provide users with a realistic and reliable

haptic and deformation feedback for objects with different mechanical properties.

Chapter 8

Conclusions and Future Work

Real-time FE-based soft tissue modeling is computationally demanding. Performing huge amount of computations involved in FE-based modeling requires a highly parallelized architecture for meeting timing requirements, especially in applications requiring high resolution, large FE meshes such as those in medical training systems in medical applications. This accentuates the need for parallel processing when none of the available processors are capable of handling FE-meshes with high enough resolution.

In this thesis, a highly parallelized multiple-FPGA implementation of the CG algorithm was proposed for solving the system of equations arising from FE models of soft-object deformation. The current implementation of this design on four Stratix III EP3SE110 devices achieves up to a peak of 302 Giga Operations Per Second (GOPS). The proposed hardware architecture employs customized fixed-point operations in order to maximize the parallelization for the computations. The hardware architecture is fairly independent of the FE mesh structure and can be scaled for deployment on multiple-FPGA devices with various capacities. The

current communication scheme for inter-FPGA communications, connects them in a ring configuration with a custom parallel communication design. This communication scheme provides fastest possible data exchange rate among FPGAs while hiring all available bandwidth on the board.

Increasing the number of nodes in FE mesh is possible by employing a sufficient number of FPGAs. Each of the FPGAs can handle up to a certain number of nodes depending on the FPGA devices in use. For the FPGA employed in our implementation, Stratix III EP3SE110, this is 1536 nodes per FPGA. Scalability of this design on multiple FPGA devices will have a negligible effect on the utilized hardware resources of each FPGA. This was achieved by exploiting the sparsity of the FE-based equations through new data storage methods for the matrix and vectors in the CG algorithm. The only limitation on the number of FPGAs in this design is the data bandwidth for inter-FPGA communications. Provided a non-adequate communication bandwidth, the CG solver will still work but it can lead to more time required for communications.

The implementation of the proposed architecture on a quad-FPGA system has enabled real-time haptic/deformation of simulation a 3D linear elastic FE model with 6000 at an update rate of 400Hz.

The future work in short term will involve developing our CG-based solver to work under PCG algorithm to improve the numerical accuracy of the solver. Employing domain decomposition methods for performing multi processing with less communications cost can be another subject of future work. This would require investigating on the issues arisen by domain decomposition methods, like

the convergence time due to more CG iterations required and updating the FE-mesh. In addition, haptic-enabled simulation of cutting, needle insertion, soft-object to soft-object contact, as well as nonlinear deformation behavior using the proposed hardware-based accelerator can be pursued in future.

Bibliography

[1] R. Mafi, S. Sirouspour, B. Mahdavikhah, and et al., "Hardware-based parallel computing for real-time haptic rendering of deformable objects," in *INTUITION 2008 Conference, Turin, Italy*, 2008.

[2] *Computer Vision Group, University of Leeds, Retrieved 20 May 2004 from* $http$: $//www.comp.leeds.ac.uk/vision/medical\_VEs.html$. Gidel, Jan,2008.

[3] S. F. Gibson, "3d chainmail: A fast algorithm for deforming volumetric objects," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pp. 149–154, 1997.

[4] *ProcStar III Data book Version 1.0, Gidel Ltd., Jan 2008*.

[5] K. Salisbury, F. Conti, and F. Barbagli, "Haptic rendering: Introductory concepts," *IEEE Computer Graphics and Applications*, vol. 24, no. 2, pp. 24–32, 2004.

[6] M. Srinivasan and C. Basdogan, "Haptics in virtual environments: Taxonomy, research status, and challenges," *Computers and Graphics*, vol. 21, no. 4, pp. 393–404, 1997.

[7] S. D. Laycock and A. M. Day, "A survey of haptic rendering techniques," *Computer Graphics Forum*, vol. 26, no. 1, pp. 50–65, 2007.

[8] O. S. Lanru Jing, *Fundamentals of Discrete Element Methods for Rock Engineering: Theory and Applications*. Elsevier, August 2007.

[9] J. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," in *Technical report, School of Computer Science, Carnegie Mellon University*, 1994.

[10] Y. Zhuang, *Real-time simulation of physically realistic global deformations*. PhD thesis, 2000. Chair-John Canny.

[11] U. Meier, O. Lpez, C. Monserrat, M. Juan, and M. Alcaiz, "Real-time deformable models for surgery simulation: A survey," *Computer Methods and Programs in Biomedicine*, vol. 77, no. 3, pp. 183–197, 2005.

[12] A. Nealen, M. Muller, R. Keiser, E. Boxerman, and M. Carlson, "Physically based deformable models in computer graphics," *Computer Graphics Forum*, vol. 25, no. 4, pp. 809–836, 2006.

[13] A. S.Misra and K.T.Ramesh., "Force feedback is noticeably different for linear versus nonlinear elastic tissue models," *Proceedings of the Second Joint Euro-Haptics*, pp. 519–524, 2007.

[14] I. peterlik, *Haptic Interaction with Deformable Objects*. PhD thesis, 2006. PhD Thesis Proposal.

[15] S. Gibson and B. Mirtich, "A survey of deformable modeling in computer graphics," in *Technical Report No. TR-97-19, Mitsubishi Electric Research Lab., Cambridge, MA*, 1997.

[16] D.C.Popescu and M.Compton., "A model for efficient and accurate interaction with elastic objects in haptic virtual environments," *GRAPHITE 03: Proceedings of the 1st internat. conf. on Computer graphics and interactive techniques*, pp. 245–250, 2003.

[17] H.-W. Nienhuys and A. F. van der Stappen, "Combining finite element deformation with cutting for surgery simulations," *EUROGRAPHICS 2000*, 2000.

[18] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 205–214, 1987.

[19] K. Waters and D. Terzopoulos, "A physical model of facial tissue and muscle articulation," in *Proceedings of the First Conference on Visualization in Biomedical Computing*, pp. 77–82, 1990.

[20] J. C. Platt and A. H. Barr, "Constraint methods for flexible models," *Computer Graphics (SIGGRAPH88)*, vol. 22, no. 4, pp. 279–288, 1988.

[21] H. Delingette, S. Cotin, and N. Ayache, "A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation," *Visual Computer*, vol. 16, no. 8, pp. 437–452, 2000.

[22] H. D. Stphane Cotin and N. Ayache., "Efficient linear elastic models of soft tissues for real-time surgery simulation," in *Technical report, INRIA*, 1998.

[23] Y. Lee, D. Terzopoulos, and K. Walters, "Realistic modeling for facial animation," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pp. 55–62, 1995.

[24] J. K. H. P. K. Z. P. K. Bhat, C. Twigg and S. Seitz., "Estimating cloth simulation parameters from video," *ACM SIGGRAPH/ Eurographics Symposium on Computer Animation*, pp. 37–51, 2003.

[25] G. S. G. Bianchi, B. Solenthaler and M. Harders., "Simultaneous topology and stiffness identification for mass-spring models based on fem reference deformations," *MICCAI 2*, pp. 293–301, 2004.

[26] C. Basdogan and M. Srinivasan, "Haptic rendering in virtual environments," *K. Stanney (Ed): Virtual Environments HandBook, Lawrence Erlbaum Associates*, pp. 117–134, 2002.

[27] D. James and D. K. Pai, "A unified treatment of elastostatic and rigid contact simulation for real time haptics," *Haptics-e, the Electronic Journal of Haptics Research*, vol. 2, no. 1, 2001.

[28] M. Bro-Nielsen, "Finite element modeling in surgery simulation," *Proceedings of the IEEE*, vol. 86, pp. 490–503, March 1998.

[29] Y. Fung., *A First Course in Continuum Mechanics*. Prentice-Hall, Englewood Cliffs, N.J., 1994.

[30] C. K. P. Boresi, A. P., *Elasticity in Engineering Mechanics*. Wiley, 2000.

[31] J. Raamachandran, *Boundary and Finite Elements*. Narosa Publishing House, New Dehli., 2000.

[32] K. J. Bathe, *Finite Element Procedures*. Prentice Hall, 1996.

[33] S. P. Jian Zhang and J. Dill, "Haptic subdivision: an approach to defining level-of-detail in haptic rendering," in *Proceedings of the 10th Symp. On Haptic Interfaces For Virtual Envir. and Teleoperator Systs.*, 2002.

[34] M. C. Cavusoglu and F. Tendick, "Multirate simulation for high fidelity haptic interaction with deformable objects in virtual environments," in *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pp. 2458–2465, 2000.

[35] Y. Zhuang and J. Canny, "Haptic interaction with global deformations," in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 2428–2433, 2000.

[36] M. deLorimier, *Floating-Point Sparse Matrix-Vector Multiply for FPGAs*. California Institute of Technology Pasadena, California, 2005. Master's Thesis.

[37] X. Wu, T. G. Goktekin, and F. Tendick *Lecture Notes in Computer Science*, vol. 4791, pp. 124–133, 2004.

[38] Z. A. Taylor, M. Cheng, and S. Ourselin, "Real-time nonlinear finite element analysis for surgical simulation using graphics processing units," *Medical Image Computing and Computer-Assisted Intervention*, vol. 4791, pp. 701–708, 2007.

[39] G. C. L. Buatois and B. Lvy, "Concurrent number cruncher an efficient sparse linear solver on the gpu," in *High Performance Computation Conference - HPCC'07, Houston : United States (2007)*, 2007.

[40] D. G. R. STRZODKA and S. TUREK, "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in fem simulations," in *INTERNATIONAL JOURNAL OF PARALLEL, EMERGENT AND DISTRIBUTED SYSTEMS*, 2006.

[41] E. S. D. G. W. J. G. Yousef Elkurdi, David Fernandez ., "Fpga architecture and implementation of sparse matrixvector multiplication for the finite element method," in *Computer Physics Communications(178)*, pp. 558–570, 2008.

[42] R. Mafi, *Hardware-based Parallel Computing for Real-time Simulation of Soft-object Deformation*. DEPARTMENT OF ELECTRICAL and COMPUTER ENGINEERING AND THE SCHOOL OF GRADUATE STUDIES OF MCMASTER UNIVERSITY, 2008. Master's Thesis.

[43] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 13th International Symposium on Field-Programmable Gate Arrays*, pp. 63–74, 2005.

[44] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *Proceedings of 15th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 349–352, 2007.

[45] P. B. Barry Smith and W. Gropp, *Domain Decomposition*. Cambridge University Press, 2004.

[46] X. Wu and F. Tendick, "Multigrid integration for interactive deformable body simulation," in *S. Cotin and D. Metaxas (Eds.): ISMS 2004, LNCS 3078, ©Springer-Verlag Berlin Heidelberg 2004*, pp. 92–104.

[47] U. Meier and R. Eigenmann, "Parallelization and performance of conjugate gradient algorithms on the cedar hierarchial-memory multiprocessor," in *Assossiations for computing machinery (ACM) Issue 7 Vol.26*, pp. 178–188, 1991.

[48] B. M. B. M. K. E. A. K. Ramin Mafiy, Shahin Sirouspoury and N. Nicoliciy, "A parallel computing platform for real-time haptic interaction with deformable bodies," in *IEEE Transactions on Haptics, Submitted*, 2009.

[49] O. C. Zienkiewicz and R. L. Taylor, *The Finite Element Method, Vol. 1, 4th Edition*. McGraw-Hill Book Company, 1989.

[50] R.W.Clough, "The finite element in plane stress analysis," in *Proc.2nd ASCE Conf. on Electronic Computation, Pittsburg, Pa.*, 1960.

[51] R.W.Clough, *The Finite Element method in structural machanics, Chapter 7 of stress analysis*. Wiley, 1965.

[52] F. L. Stasa, *Applied Finite Element Analysis for Engineers*. Holt, Rinehart, and Winston, 1985.

[53] L.Collatz, *The numerical treatment of differential equations*. Springer-Verlag, 1966.

[54] S. H. Crandall, *Engineering Analalysis*. McGraw-Hill book Company, New York, N.Y., 1965.

[55] G. Karniadakis and R. K. II, *Parallel Scientific Computing in C++ And Mpi: A Seamless Approach to Parallel Algorithms and Their Implementation*. Cambridge University Press, 2003.

[56] Y. Saad, *Iterative Methods for Sparse Linear Systems, 2nd Edition*. SIAM, 2003.

[57] K. H. Huebner, D. L. Dewhirst, D. E. Smith, and T. G. Byrom, *The Finite Element Method for Engineers, 4th Edition*. Wiley, 2001.

[58] *Stratix III Device Handbook, Volume 1, Altera Corporation, July 2008*.

[59] J. I. f. C. S. K. L. Wong, "Iterative solvers for system of linear equations," 1997.

[60] G. G. F. Smailbegovic and T. D. U. S. Vassiliadi, Computer Engineering Laboratory, "Sparse matrix storage format," *In Proceedings of the* 16^{th} *annual workshop on circuits, systems and signal processing*, pp. 445–448, Nov,2005.

[61] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the performance of sparse matrix-vector multiplication," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pp. 283–292, 2008.