McMaster University
Department of Electrical & Computer Engineering

GPU-based Parallel Computing for Nonlinear Finite Element Deformation Analysis

GPU-BASED PARALLEL COMPUTING FOR NONLINEAR FINITE ELEMENT DEFORMATION ANALYSIS

BY

RAMIN MAFI, M.A.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

DOCTOR OF PHILOSOPHY

Master of Applied Science (2013) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	GPU-based Parallel Computing for Nonlinear Finite Element Deformation Analysis
AUTHOR:	Ramin Mafi M.A.Sc., Electrical Engineering McMaster University, Hamilton, Canada B.Sc., Biomedical Engineering B.Sc., Electrical Engineering Amirkabir University, Tehran, Iran
SUPERVISOR:	Dr. Shahin Sirouspour
NUMBER OF PAGES:	xv, 156

to my parents,

for their endless love,

support and encouragement

ABSTRACT

Computer-based surgical simulation and non-rigid medical image registration in image-guided interventions are examples of applications that would benefit from real-time deformation simulation of soft tissues. The physics of deformation for biological soft-tissue is best described by nonlinear continuum mechanics-based models which then can be discretized by the Finite Element Method (FEM) for a numerical solution. Computational complexity of nonlinear FEM-based models has limited their use in applications requiring real-time or fast response. However, the data-parallel nature and intense arithmetic operations in nonlinear FEM models are suitable for massive parallelization of the computations, in order to meet the response time requirements in such applications. Modern GPU architectures with large number of computing cores and a high memory bandwidth offer a cost effective hardware platform in compact form factor for carrying out these types of computations.

This thesis is concerned with computational aspects of complex nonlinear deformation analysis problems with an emphasis on the speed of response using a parallel computing philosophy. It proposes a novel Graphic Processing Unit (GPU)-based implementation of the total Lagrangian FEM using implicit time integration for dynamic nonlinear deformation analysis. This is a general formulation

of the deformation analysis. It is valid for large deformations and strains and can account for material nonlinearities. A penalty method is used to satisfy the physical boundary constraints due to contact between deformable objects, and a local mesh refinement data structure and algorithm is proposed to produce a finer mesh at the contact boundaries.

The proposed GPU-based solution addresses computational challenges in constructing nonlinear FEM matrices, as well as solving the linear system of equations resulting from implicit time integration. The proposed set of optimized GPU kernels for computing the FEM matrices achieves more than 100 giga floating-point operations per second (GFLOPS) and up to 28× speed-up on a GTX 470 GPU device compared to a sequential implementation on an Intel core i7-3770 CPU. Two different designs based on the element-by-element and conventional Preconditioned Conjugate Gradients (PCG) algorithms are presented and compared for solving the FEM equations arising in deformation analysis. The use of a novel vector assembly kernel and memory optimization strategies result in a performance gain of up to 25 GFLOPS in the PCG computations.

In summary, this thesis presents a fast, accurate and scalable solution for simulation of soft-tissue deformation. The speed-up achieved with the proposed parallel implementations of the algorithms will be instrumental in the development of advanced surgical simulators and medical image registration methods involving soft-tissue deformation.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Shahin Sirouspour for his invaluable suggestions, support and encouragement throughout my graduate program at McMaster university. In addition, I give my special thanks to Dr. Alexandru Patriciu for his immense knowledge on GPU computing and insightful comments and suggestions throughout my Ph.D. research program. I also would like to thank my other committee members, Dr. Nicola Nicolici and Dr. Aleksandar Jeremic for their interest in my work.

My deepest gratitude goes to my parents, grandmother, sister and brother for their unwavering love and encouragement throughout my life. I'd like to acknowledge my late uncle, Vahab Montazery, whom without his encouragement and support, my ambition to pursue graduate studies abroad could hardly be realized. Although no longer with us, he is always in our hearts and thoughts. I'd also like to give my sincere thanks to my uncle Esfandiar Mafi and his kind family for their love, caring and occasional visits.

I was fortunate to know many interesting people and make great friends during my graduate program in Canada. I would like to acknowledge *all* my friends, specially Fabiola De Vierna, who taught me to discover the joy of exploring and trying new things. Peyman Setoodeh for all the great moments and fun movie

nights, Sadegh Dadash whom we shared lots of joy and sweat of volleyball, gym workout and chess games together! Alireza Azami for the ping-pong challenges, Mahboob Bolandi for the music band and Ali Khalatpour for the foosball games!

Additionally, thanks to all my lab-mates at McMaster Telerobotics, Haptics and Computational Vision Laboratory: Bahram Marami, Hamed Mousazadeh, Pawel Malysz, Behzad Mahdavikhah, Behzad Iranpanah, Sajad Salmanipour and Saman Rahnamaei for the stimulating discussions and making a great working environment. I greatly appreciate the partial funding of my research provided by Ontario Graduate Scholarships (OGS) and Natural Sciences and Engineering Research Council of Canada (NSERC).

NOTATION

τ and t	current and next time-steps respectively
$\mathbf{u}$ or $\mathbf{U}$	displacement vector
$\boldsymbol{F}$	deformation gradient tensor
J	determinant of the deformation gradient
$\boldsymbol{I}$	Identity tensor
$\boldsymbol{E}$	Green-Lagrange or Green strain tensor
$\boldsymbol{\varepsilon}$	Engineering strain tensor
$\boldsymbol{\eta}$	quadratic term of the Green strain tensor
$\boldsymbol{\sigma}$	Cauchy stress tensor
$\boldsymbol{S}$	second Piola-Kirchhoff stress tensor
$\boldsymbol{C}$	elastic modulus (material) tensor
$\boldsymbol{C_r}$	right Cauchy-Green deformation tensor
I_C, II_C, III_C	first, second and third invariants of right Cauchy-Green tensor
Ψ	strain energy density function
λ and μ	*Lamé* constants
e	right superscript e denotes the quantity is defined over a finite element
$\mathbf{K_L}$ and $\mathbf{K_N}$	linear and non-linear components of the stiffness matrix
$\mathbf{K}_\alpha$	contribution of the penalty-based contact model in the stiffness matrix
$\mathbf{F}$	vector of nodal internal forces due to internal stresses
$\mathbf{R}$	vector of external forces
$\mathbf{R_c}$	vector of contact forces
$\mathbf{B_L}$ and $\mathbf{B_N}$	linear and nonlinear strain-displacement matrices
$\mathbf{p}$, $\mathbf{t}$ and $\mathbf{pt}$	nodal position, elemental and nodal-elemental matrices of the mesh data
$\biguplus$	assembly operator of the finite element vectors/matrices
g	gap between a contact pair
f_c	contact force magnitude
α	penalty parameter

Contents

List of Figures

CHAPTER 1

INTRODUCTION

Advanced surgical simulation systems and intraoperative image registration algorithms are examples of applications that require fast/real-time analysis of non-rigid body deformation. Virtual reality (VR)-based simulators are emerging as a promising alternative to the conventional means of training in the medical field [1, 2]. They allow surgeons to practice on virtual patients as they would operate on real patients with realistic sensory feedback.

Research studies on the role of human errors in the medical field [3, 4] point to a need for sufficient theoretical knowledge as well as proficient fine manipulation skills for reducing such errors. Computer-based simulation offers a safe and reliable environment for medical trainees to learn, repeatedly practice and improve their manual skills without placing patients at risk of harm. Computer-based simulation allows proper training of rare clinical cases by replicating those scenarios.

In this approach, the trainee's progress can be quantitatively evaluated and the task difficulty level can be adjusted accordingly. Fig. 1.1 summarizes some factors contributing to the growing interest in computer-based medical simulation techniques [5].

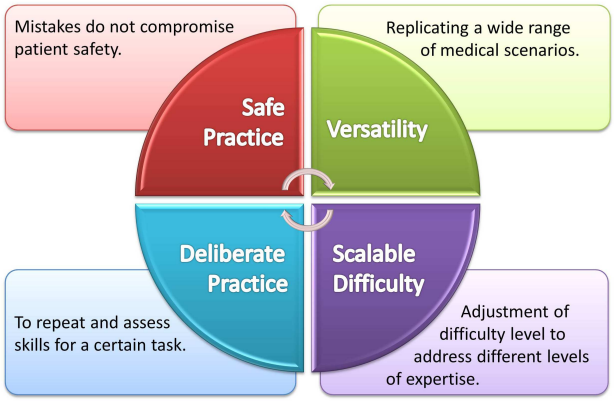


Figure 1.1: Different factors affecting interest in computer-based medical simulation.

The ability to compute soft-tissue deformation in real-time is an indispensable component in a computer-based surgical simulation. There has been considerable research in developing high-fidelity deformation models where fidelity refers to the extent of similarity of appearance and behavior of the model to the real system [6]. One main challenge in this regard is to find a good balance between the computation speed and accuracy of the model.

Fast deformation analysis using a physical model of soft tissue can also be useful in the development of deformable medical image registration algorithms for interventional and diagnostic purposes [7]. Deformable registration consists of deriving a nonlinear local transformation to minimize the difference between two images [8]. The main idea is to register low-resolution intraoperative images to

high-resolution preoperative images in order to guide the surgeon through the procedure. This technique is in particular very useful in minimally invasive surgeries where only a small incision is made and the surgeon may not rely on direct visualization of the surgical site.

Non-rigid registration in image-guided surgery is challenging. A physical deformable model can be useful as a constraint for performing non-rigid registration. To account for the organ motion and deformation, the boundary displacement of the deformed body is used as an input to the deformable model [9]. One example of use of deformable models in image-guided surgery is in assessing brain shift in minimally invasive neurosurgery [10, 11]. In this procedure, the brain is subject to deformation due to changes in boundary constraints. Another example is in percutaneous therapy, where pre-operative intervention plans often have to be monitored and revised in real-time to account for soft-tissue deformation due to needle insertion as well as organ movement and respiratory motion. Independent of the imaging modality used, intra-operative images can be registered to a pre-operative surgical plan in order to provide the operator with real-time feedback as the procedure unfolds [12].

In summary, in computer-based surgical simulations and image-guided surgeries of soft tissues, it is essential to have a realistic and reliable mechanical model to simulate interactions with deformable organs in real-time.

1.1 Problem Statement

Realistic modeling of deformable interactions can be computationally expensive. An ideal solution for applications involving fast or real-time deformation analysis

should address both accuracy and speed to the required level. The goal in this thesis is to step toward such a solution by proposing an efficient and highly parallel computation scheme based on Graphic Processing Unit (GPU) devices.

In real-time or fast applications, strict constrains on the computation time make deformation analysis a daunting task. Interactive simulation systems require a refresh rate of around 30 Hz for smooth graphics rendering. Additionally, if force feedback (haptics) is employed, the update rate should be in the range of 100-1000 Hz [12]. Violating these timing constraints could degrade the simulation quality and even cause physical instability when haptic feedback is involved.

In general, deformable interaction analysis demands a common framework to address a number of issues involving (a) realistic graphics rendering, (b) handling collisions, (c) accurate physical modeling, (d) deformable contact simulation and (e) numerical solution of a large system of equations. Fig. 1.2 presents a block diagram of these computing tasks. Note that an optional haptic device can enable bidirectional interaction between the user and the deformable tissue model. This thesis is concerned with the tasks (c) to (e), which involve massive computing operations on large sets of data. A more detailed discussion of each of these problems follows.

1.1.1 Accurate Physical Modeling

The response of the deformable model to external loads is characterized by *stiffness* property. There has been significant research in modeling the stiffness of non-rigid objects [13]. Heuristic mass-spring models [1] are easy to develop but not accurate [14]. On the other hand, methods based on computational continuum

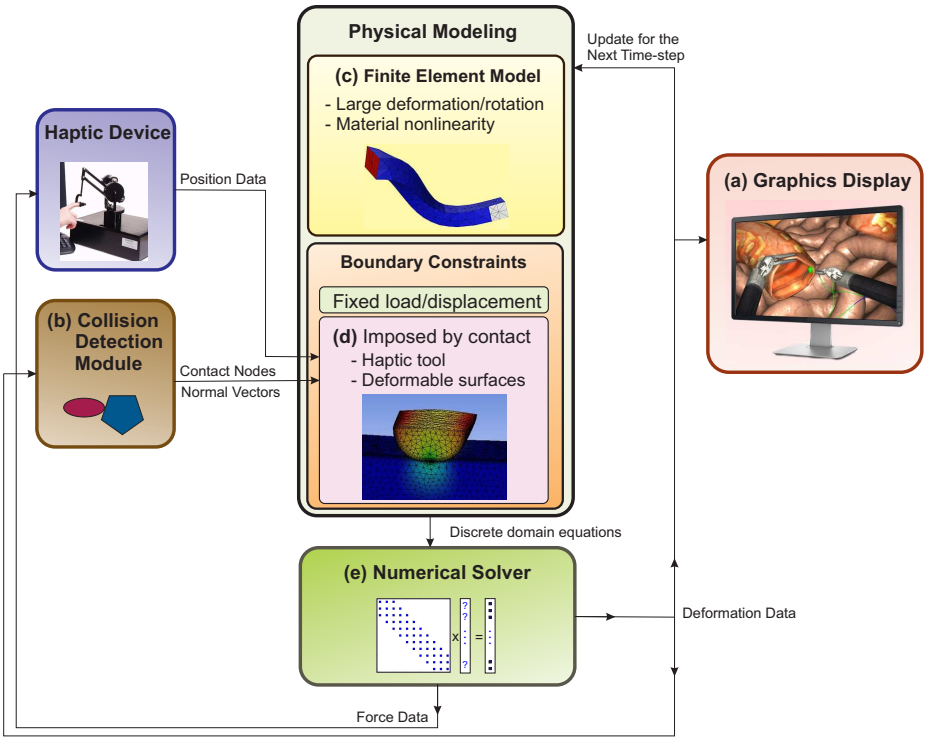


Figure 1.2: Different computing tasks involved in interactive simulation of deformable objects.

mechanics such as the finite element method (FEM) have gained popularity since they can potentially produce accurate results [15]. FEM can specify stiffness of the model using few material parameters as opposed to the mass-spring model, which requires adjusting a large number of spring constants [16]. However FEM is computationally expensive, making its use in real-time applications challenging.

Stiffness of a deformable object depends on the model geometry, material property and boundary constraints. In a linear model, changes in these factors are considered to be small throughout the deformation analysis and the stiffness remains unchanged. However, this is not a valid assumption in many real-life examples such as in surgical simulation involving biological soft tissue. In such cases, a

nonlinear finite element analysis should be considered. Nonlinearity adds more complexity to the finite element analysis since in general, actual geometry, boundary conditions and material properties are unknown a priori and an incremental procedure should be followed to find the solution [17]. Generally there are two types of nonlinearity in the area of solid mechanics, namely *geometric* and *material* nonlinearity [18].

- Geometric nonlinearity is attributed to large strains, or small strains but large rotation. Examples of such nonlinearity can be found in modeling of metal forming or rubber-like materials exposed to large strains [19]. The latter is a common case in soft tissue modeling in surgical simulation. Geometric nonlinearity also includes deformation dependant load or displacement boundary constraints.

- Material nonlinearity occurs when the stiffness varies due to the changes in material property. This is often expressed by a nonlinear constitutive equation relating stress and strain. Examples of such nonlinearity include but not limited to hyperelasticity, plasticity and viscoelasticity. The latter is time-dependent.

Fig. 1.3 displays a general procedure for deriving the stiffness model of a deformable object and different sources of nonlinearity in the model. Large deformation/rotation is considered in the definition of a nonlinear strain term and material nonlinearity is included in the stress-strain relationship. There is also boundary condition nonlinearity imposed by contact interaction, which will be discussed in Section 1.1.2.

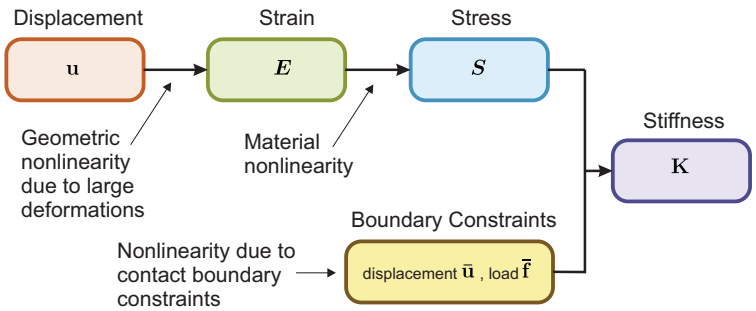


Figure 1.3: Different sources of nonlinearity.

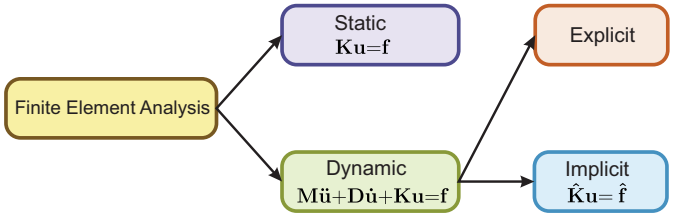


Figure 1.4: Static and dynamic FEM.

Linear or nonlinear finite element equations can be derived for steady-state or transient analysis. In structural mechanics, the former is referred to as *static* analysis and is time independent. The latter is known as *dynamic* analysis and is time dependent. In dynamic analysis, inertial effects and velocity-dependent damping forces are also considered. In this case, spatial discretization is followed by temporal discretization using *explicit* or *implicit* integration methods [20]. Implicit integration is computationally costly, since it involves solving a large linear system of equations per time-step, similar to the one in static analysis; however it allows for larger time increments without causing numerical instability. Explicit integration significantly reduces the computations per time step as it does not involve numerical solution of a large system of equations. However it suffers from stability issues and may require prohibitively small time steps to satisfy the numerical

stability criteria. The diagram in Fig. 1.4 shows that both implicit dynamic analysis and static analysis lead to a set of linear equations for finding the unknown object deformation.

1.1.2 Deformable Contact Simulation

Contact analysis is required for any mechanical system involving interaction between different bodies. Boundary constraint imposed by contact should prevent physical interpenetration between surfaces in contact. There are different approaches to apply contact boundary constraints on the problem. Simplified systems of interactive deformation analysis consider only single-point contacts, for example in simulation of a surgical tool tip in contact with a deformable tissue. However, for accurate results in a more general case, contact between deformable surfaces should be considered. Interaction of deformable organs with each other in a surgical simulation is an example of a case requiring such contact modeling.

Contact can be between two (or more) objects at equilibrium state, known as static contact, or in transient state, known as dynamic contact. The latter is more complex to model. Friction at contact surfaces increases the complexity of the contact behavior. Contact analysis between two or multiple objects is an inherently nonlinear problem, since the contact region and displacement boundary conditions imposed by contact are unknown before solving for the deformation [21]. Since contact nonlinearity is associated with the geometry at the surface area, it can be considered as one form of geometric nonlinearity [18].

1.1.3 Numerical Solution of Linear System of Equations

The linear system of equations arising from the static or implicit dynamic FEM, depending on the size of the model, can be quite large. Solving this system of equations is the most time consuming task in the simulation loop for interactive deformable objects in Fig. 1.2. The system of equations can be expressed in matrix form $\mathbf{Ax} = \mathbf{b}$, where matrix $\mathbf{A}$ has a sparse symmetric structure. There are different algorithms to solve $\mathbf{Ax} = \mathbf{b}$. These algorithms are grouped into *direct* and *iterative* methods. In solving a large sparse system of equations, iterative methods can offer better savings in terms of memory usage and computing time compared to direct methods [22]. Preconditioned Conjugate Gradient (PCG) [23] is one of the most popular iterative methods. Owing to its robust numerical behavior and relatively low computational complexity, PCG is widely used in solving the FEM discrete system of equations [24].

...

In summary, nonlinear finite element analysis using implicit integration of deformation in applications requiring fast/real-time response is very challenging. In past, two general approaches have been pursued to solve this complex computational problem. In the first approach, the models are greatly simplified to reduce the computation time, but at the expense of reduced accuracy [25, 26]. Another approach relies on achieving high performance computing through parallel execution of instructions on computer clusters or special-purpose many-core accelerators. Nodal and elemental computations in FEM are mostly data independent and hence are well-suited for parallel implementation.

1.2 Parallel Computing

In the last decade, the conventional approach of speeding up CPUs by increasing the processor clock speed has faced some barriers [27]. Physical limits on power consumption, heat dissipation and current leakage [27, 28] have capped the CPU clock frequency. A new trend has emerged to achieve increased computing power through expansion to multiple processing cores rather than focusing on single-threaded architectural optimizations. Beside multi-core CPUs, many-core accelerators have gained increasing popularity in high-performance computing. Today, programmable GPUs, FPGAs (Field Programmable Gate Arrays) and recently released Intel MIC (Many Integrated Core architecture) offer viable solutions for compute-intensive data-parallel problems [29, 30]. In a heterogeneous computing paradigm, serial portions of an algorithm can run on a CPU, while compute-intensive tasks are delegated to a many-core co-processor. Heterogeneous computing is a powerful approach to utilize the massive parallel computing power of available resources to reduce solution time [31].

Parallel computing requires extra effort for modifying algorithms so can be executed concurrently. Some challenges in parallel computing include learning relevant programming paradigms, optimal usage of memory bandwidth, achieving good load balancing between the computing cores, minimizing data communication, and managing synchronization and possible data race conditions. Additionally, this approach may not be applicable to all types of computational problems due to data dependencies. A computational problem in which the same operation is performed concurrently on a large set of data is known as being data-parallel and potentially can fit well on many-core accelerators. Fortunately, the majority of

computations involved in physics-based deformation analysis are of data-parallel nature.

High performance computing solutions based on computer clusters for real-time deformation simulations are limited due to the cost and complexities associated with distributed memory [32]. FPGAs are highly flexible in customizing the computing hardware architecture to the problem, and through large number of processing elements and fast on-board memory, can achieve very high computing performance. However, developing an FPGA-based solution for the computations of the general nonlinear deformation model is very challenging [33]. In recent years, GPUs have evolved from special-purpose computing devices into general purpose computing platforms. Thanks to their massive computing power, transparent scalability, relative ease of programming, accessibility, low cost and a small form factor, GPUs have gained popularity in high-performance computing [31]. Intel MIC is a recent competitor to general-purpose GPUs in the area of high-performance computing [34]. Intel MIC can be programmed with OpenMP [35], Intel Threading Building Blocks (Intel TBB) [36] or Intel Cilk Plus [37]. The new intel co-processor shows comparable performance to high-end GPUs in some parallel applications; however it is not as ubiquitous as GPUs.

A critical challenge for achieving high degrees of efficient parallelism in co-processors is to employ an optimal strategy for using memory hierarchies. Modern GPU architectures provide a large memory bandwidth to the highest level of memory hierarchy, known as global memory. But access to the global memory has a very high latency (400-600 clock cycles) [38]. Additionally, in an improper memory access pattern, the amount of data that can be transferred to the computing

units at each cycle can be fairly limited, hence reducing the degree of parallelism. Therefore, a key point for achieving good performance is to hide latency by employing large number of active threads and utilizing maximum possible effective memory bandwidth.

1.3 Thesis Contributions

The FEM analysis of continuum mechanics based models of deformation is known to generally produce accurate and reliable results in modeling of soft-object deformation. However, the use of such models in real-time applications is inhibited by their complexities, specially when considering nonlinearity and massive computations required for obtaining and solving the resulting system of equations. Fig. 1.5 illustrates the computing blocks in finite element formulation and the iterative conjugate gradients method for solving the resulting system of equations. This thesis investigates methods for efficient parallel execution of the loops and vectorized computations on GPU in nonlinear FEM deformation analysis .

Fig. 1.5 is based on an element-by-element CG algorithm [39] where matrix by vector multiplication is performed directly on elemental FEM matrices followed by vector assembly. An alternative approach is to first assemble the elemental matrices into a global sparse matrix, and then perform the matrix by vector multiplication in parallel. Both of these approaches will be studied and compared in Chapters 6 and 8.

In this thesis, highly parallel GPU-based methods are proposed for computing nonlinear FEM matrices and solving the resulting system of equations. The deformation model is based on the general total Lagrangian nonlinear FEM formulation

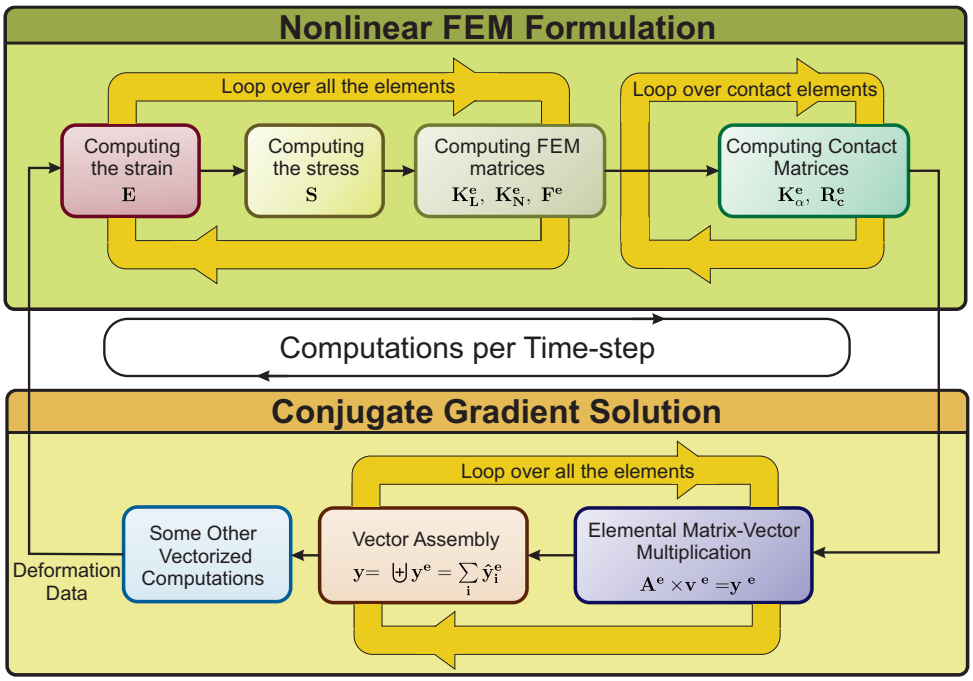


Figure 1.5: Computing diagram consisting of nonlinear FEM calculations followed by CG iterative method.

presented in [40]. This formulation considers large displacements, large strains and material non-linearities, without using the so-called co-rotational approximation technique. For dynamic analysis, the *Newmark* implicit time integration [20] is utilized. The proposed GPU-based solution addresses the real-time computing challenges in both areas of nonlinear FEM matrices construction and solving the system of equations resulting from implicit time integration. It exceeds 100 GFLOPS in the computation of nonlinear FEM matrices and 25 GFLOPS in the iterative equation solver on a single GTX 470 GPU device. These are very promising results for applications requiring fast rendering of soft-tissue deformation.

The main objective of the proposed computational methods is to achieve *speed*, *accuracy* and *scalability* in FEM-based deformation analysis. Fig. 1.6 shows how the work in this thesis accomplishes each of these objectives. The results of this

research are of great significance to applications requiring fast computation of the deformation response of soft objects, including in surgical training and medical image registration applications.

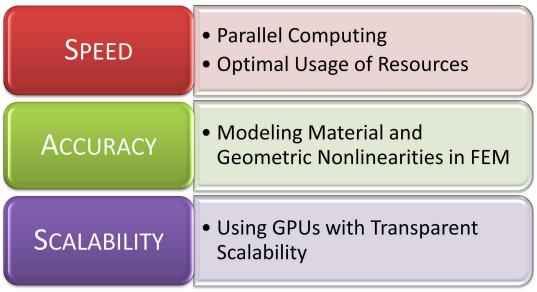


Figure 1.6: Objectives and methodology.

A key novelty of the proposed computational methods for GPU-based nonlinear finite element analysis of deformable objects is in their ability to efficiently utilize memory bandwidth to continuously supply data operands to a large number of computing units within modern GPU architectures. The computation of finite element matrices is *embarrassingly* parallel and can be scaled up based on available GPU resources and beyond, to multiple-GPUs. The iterative PCG solver involves matrix-vector operation which can be performed either with local elemental matrices or a global sparse matrix. An efficient GPU implementations of both of these approaches will be presented in this thesis.

In summary, the main contributions of this thesis are:

- A novel highly parallel and optimized GPU-based method for computing the finite element matrices in nonlinear two and three dimensional deformation models.

- Addressing the main sources of nonlinearity including material, geometric due to large rotations/deformations and contact interaction under an implicit integration scheme, fully realized on GPU.
- A data structure and marking scheme to allow integration of a basic mesh refinement method at contact elements.
- A novel vector assembly method which uses GPU shared memory to increase the computation performance.
- Presenting and comparing two different GPU implementations of the preconditioned conjugate gradients method for solving large systems of equations.
- Performance analysis of the GPU kernels.

1.4 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2 reviews the literature on different physical models and computing approaches for deformable interaction analysis. The focus of this review is on parallel computing methods with biomedical applications. The previous work related to deriving finite element matrices, contact formulation, assembly and solving the linear system of equations on multi-core and many-core processors is presented in this chapter.

Chapter 3 presents some relevant theoretical background on the continuum mechanics. First, a mathematical description of the deformation is presented. Then, strain and stress measures are derived. Constitutive equations relating stress and strain of a material are discussed next. At the end of this chapter, the principal

of virtual displacement is reviewed as a basis for establishing the finite element formulation.

Chapter 4 explores a total Lagrangian formulation of the finite element method in presence of large deformations and material nonlinearity. Different steps of the FEM formulation including continuum differential equilibrium equation, incremental decomposition, linearization and discretization using shape functions are briefly reviewed in this chapter.

The first part of Chapter 5 is dedicated to the computational contact modeling where a mathematical description of the discrete interaction surfaces and physical constraints due to contact are presented. Then a penalty method for imposing the physical contact boundary constraints is discussed. The second part of Chapter 5 presents a local mesh refinement method at the contact boundaries for enhanced accuracy. Details of the mesh data structure and marking stage for the parallel GPU implementation are discussed at the end of this chapter.

Chapter 6 is concerned with the numerical solution of the linear system of equations resulting from the FEM analysis. A number of efficient numerical algorithms and preconditioning techniques are discussed and compared. At the end, two different implementations of the conjugate gradients method for GPU-based computing are analysed.

Chapter 7 describes GPU-based massively parallel computations involved in the finite element deformation analysis. An introduction to GPU computing and basic concepts in CUDA architectures is followed by discussion of optimization strategies of the GPU kernels. Details of computing FEM matrices, assembly and Jacobi preconditioned CG on GPU are discussed in this chapter.

Chapter 8 presents the results of several numerical experiments and assesses the performance of different GPU kernels for FEM-based deformable interaction modeling. Conventional and element-by-element PCG are compared and contrasted in this chapter for GPU-based computing.

Chapter 9 concludes the thesis with a summary and a critical discussion of the modeling and computing methods presented in this thesis. Several possible avenues of future work are also briefly discussed.

1.5 Related Publications

Journal Paper

1- R. Mafi and S. Siroupour, "GPU-based Acceleration of Computations in Nonlinear Finite Element Deformation Analysis", *International Journal for Numerical Methods in Biomedical Engineering* (in press).

2- R. Mafi and S. Sirourpour, "GPU Acceleration of Hyperelastic Finite Element Method with Large Deformations and Contact Simulation", in preparation for submission to *Journal of Parallel and Distributed Computing*.

3- B. Mahdavikhah, R. Mafi, S. Sirouspour and N. Nicolici, "A Multiple FPGA Parallel Computing Architecture for Real-time Simulation of Soft-object Deformation", *ACM Transactions on Embedded Computing Systems* (in press).

Refereed Conference Paper

1- B. Mahdavikhah, R. Mafi, S. Sirouspour and N. Nicolici, "Haptic Rendering of Deformable Objects using a Multiple FPGA Parallel Computing Architecture", Proc. of *FPGA 2010 Conf.*, Monterey, California, February 2010.

CHAPTER 2

LITERATURE REVIEW

Advances in computing technology has given rise to new applications of medical training simulators and computer-assisted surgical tools. This chapter reviews different approaches in modeling and computation of soft-object deformation, with a particular emphasis on biomedical applications.

Accurate numerical analysis of non-rigid objects can be computationally demanding. Generally, this may consist of spatial and temporal discretization, evaluation of the stiffness in the discrete domains considering nonlinear attributes and numerical solution for deformation considering the boundary conditions. There are numerous methods in the literature intended for reducing the computational complexity of deformable models in order to achieve a fast simulation response time [25, 41–43]. However, most of these methods have limited accuracy and are

only valid for a small domain of applications. For example, disregarding nonlinear effects and using a linear model is valid only for applications where the object undergoes small deformation and adheres to linear elastic material behavior. This is not often the case in simulation of surgical procedures. Using a coarse spatial discretization of a model domain would reduce its computations at the expense of its fidelity and accuracy.

High performance computing powered by modern many-core processors has created an opportunity to employ realistic deformation models in applications requiring fast/real-time response. What will follow is a brief review of different deformable models, contact models, computations in deformation analysis and methods for solving large systems of differential equations in the literature with a particular interest in parallel computing.

2.1 Deformable Models

Deformation modeling has found an ever-increasing significance in different applications ranging from graphic animation to medical image registration, haptics and surgical simulation [44]. Depending on whether the deformation models are physics-based or not, they can be generally classified in different categories. Some early examples of deformable models such as spline [45, 46], free form deformation [47] and chain-mail algorithm [48] lack rigorous foundation in physics of the problem. These models are mainly applied in computer graphics and have limited application in surgical simulation since they often produce unrealistic response [1]. In the following, mass-spring system and finite element method and some other physics-based models are briefly reviewed. A more comprehensive survey of the

subject can be found in [13,44,49,50].

2.1.1 Mass-Spring Systems

Mass-spring systems gained popularity in earlier research thanks to their relative simplicity for real-time implementation [51]. In this approach, the model domain is considered as a set of mass points interconnected via springs. Using the second law of Newton, the differential equations describing the dynamics of each mass point can be derived. The main computing steps involved in a mass-spring model are [52]:

(a) computing the net force applied per mass point. The force vectors are found by applying the Hooke's law over spring links connected per mass point.

(b) time integration of the mass points. This can be done using explicit or implicit time integration methods [53,54] on the differential equation per mass point.

(c) updating the positions of the mass points.

In a mass-spring system, the continuous equations of the motion are discretized. This non-continuum mechanics basis results in degraded accuracy compared with that achievable with the finite element method where the continuity is preserved by using interpolation functions. Large mass-spring models may impede rapid global propagation of deformation, resulting in local deformations [50]. In general, mass-spring systems cannot realistically model volume incompressibility in three-dimensional models. To address this problem, some extra ad hoc force constraints may be employed [44,55].

Another problem associated with mass-spring systems is the lack of a clear physical relation for defining the spring constants. Some previous studies tried

to identify these parameters with reference to results achieved by more accurate methods based on continuum mechanics [56, 57]. Bianchi et al. [56] used a genetic algorithm to identify the topology and constant parameters of three-dimensional mass-spring grid with reference to the corresponding FE-based model. In a different approach, Lloyd et al. [57] proposed an analytical expression for the spring parameters by equating the stiffness matrix derived by FEM and the linearized mass-spring model differential equations.

2.1.2 Finite Element Method

The finite element method is predominantly applied in numerical analysis of continuum mechanics based models, and is flexible in modeling complex geometries [20, 58]. In this approach, the object is partitioned into smaller elementary shapes, known as finite elements. These elements are connected to each other via nodes at their boundaries. The governing differential equations of equilibrium are applied per element. The solution is approximated by a weighted sum of *interpolation functions*, also known as *shape functions*. This process yields differential equations involving the position of the nodal points of finite elements [20]. FEM formulation will be discussed in more details in Chapter 4.

Linear FEM has been employed in several studies for modeling soft biological tissues [25, 41, 59]. This type of model is limited in accuracy but its simplicity and computational efficiency compared with that of nonlinear formulations have been the main factors driving its popularity [49]. Delingette et al. [41] and Bro-Nielsen and Cotin [25] proposed some methods based on pre-computing the inverse of

stiffness matrix derived by linear finite element in order to achieve real-time performance. The main limitation of such methods is the inability to handle a changing model stiffness. Varying boundary constraints, e.g. due to physical contact, geometric and material nonlinearity, and topological changes, e.g. due to cutting, can lead to a change in the stiffness. DiMaio and Salcudean [59] used a linear elastic material model in 2D FEM for needle insertion simulation and validated their model using experimental results. They achieved fast interactive simulation rate using a condensation technique to find the force and displacement along the needle.

Different forms of nonlinear FEM with biomedical applications have been addressed in [14, 60–63]. Zhuang and Canny [14] worked on geometric nonlinear FE models. They used linear elastic material model and graded-mesh for increasing the computation speed. The model was approximated by mass lumping in order to enable use of explicit time integration. Employing explicit time integration significantly simplifies the matrix update and eliminates the need for solving the system of linear equations [64]. However, explicit time integration methods are only conditionally stable and may require prohibitively small time steps to maintain their numerical stability.

Dick et al. [62] and Courtecuisse et al. [63] described geometric nonlinearity by use of co-rotated strain formulation [65]. They used a linear elastic material model for implicit dynamic analysis. The co-rotational approximation technique offers accurate approximations in the presence of large rotations, however it fails to accurately model geometric nonlinearity due to large strains [65].

Wu et al. [60] considered both material and geometric nonlinearities for FE-based modeling of triangular meshes. They relied on mass-lumping assumption and explicit integration method to reduce the solution time of the dynamic deformation analysis. Moreover, an adaptive meshing method was employed to reduce the number of mesh elements. In this approach, regions of the model undergoing small stress gradients were computed through a coarse mesh and a finer mesh was employed at contact regions with significant deformation.

Miller et al. [64] and Taylor et al. [61] used the total Lagrangian FEM with explicit integration for dynamic analysis of 3D models. Geometric nonlinearity in the presence of large deformations was accounted for by using nonlinear Green-Lagrange strain tensor, and a Neo-Hookean hyperelastic was employed as the nonlinear material model.

2.1.3 Other Methods

Mesh generation is a time consuming task in FE-based models. Applications such as cutting simulation are challenging due to real-time update requirement of the model mesh. Another problem associated with mesh-based computational techniques is the possibility of mesh elements distortion due to large deformations. Distorted elements can significantly degrade the numerical analysis accuracy. To overcome these problems, some meshless techniques including meshless FEM have been proposed [66–68].

De et al. [69] introduced a mesh-free computing scheme for real-time large deformation analysis known as Point Collocation-based Method of Finite Spheres (PCMFS). In this approach, the continuum mechanics differential equations for

linear elasticity are solved by approximate functions defined on a set of particles scattered in the computing domain. Lim and De [70] extended PCMFS method to consider for geometric nonlinearity in surgical simulations with haptic feedback. They adopted a multi-resolution scheme to enhance the computing results.

Boundary element method (BEM) [71,72] is another physics-based method used in deformation analysis. In the BEM, the governing differential equations of equilibrium are evaluated by integration at boundaries of the computing domain. Consequently, the problem size for a volumetric deformable model is reduced to discrete nodes at its surface mesh. However, unlike in the FEM or mass-spring systems, the linear system of equations derived by the discrete model is dense and asymmetric. This would limit efficient application of iterative solvers in solving the system of equations. Additionally, deformation behavior of objects with non-homogenuous material can not be modeled using the BEM [60]. Wang et al. [71] used the BEM for modeling prodding, pinching and cutting deformable objects.

2.2 Contact Models for Deformable Objects Modeling

Modeling contact interaction of deformable objects with other rigid or deformable surfaces is an important element of a surgical simulation system. Many examples of real-time interaction with deformable models are based on single-point contact [60,73–76]. Such methods cannot properly model the general physical interaction where the contact force depends on the geometry of the contact surface rather than a single point [77]. Mahvash and Hayward [77] took this approach one

step further by allowing contact of rigid tool with arbitrary shape with deformable body at multiple points. Tangential forces due to friction were considered for contact simulation. The method used in [77] was based on massive pre-computation of different combinations of tool-body contact points followed by interpolation to express contact forces at arbitrary points based on pre-computed nodes for real-time simulation. Methods based on pre-computation, in general cannot be used in presence of material or geometric nonlinearity where the superposition principle does not apply.

Contact analysis between two or several deformable objects at multiple points is more challenging compared to single-point contact, since the deformation and contact forces are both unknown. Depending on the method of applying the contact constraints on the discretized finite element model, there exist different formulations for contact analysis.

The *penalty method* is one of the common approaches in computational contact analysis [78–80]. In this method, the contact force is defined based on the measure of penetration between contact surfaces via a penalty factor. While robust non-penetrating contact interactions require large values for penalty factor, physical stability of contact simulation imposes an upper limit on this value [81]. Compared to other methods, penalty-based contact formulation requires fewer computations and is easier to implement.

The *Lagrange multiplier method* is another popular approach for modeling contact constraints. In this approach, the contact constraints are enforced with no approximation, however the size of the system of equations increases depending on the number of contact constraints. Cotin et al. [82] employed a simple form of

Lagrange multipliers to impose the deformation constraint introduced by haptic tool/deformable body interaction. This model considered multiple-point interactions at the mesh nodes.

Duriez et al. [83] proposed a Lagrange multiplier-based solution using Signorini's law for real-time simulation of interacting deformable objects. Signorini contact model [84] states a complimentary relation between the gap and stress due to contact between two objects; if there is a gap between two objects, $g \geq 0$, the contact stress is zero, $\sigma = 0$, and if the contact stress is non-zero, $\sigma \geq 0$, the gap between the objects is zero, $g = 0$. A formulation based on Signorini's law leads to linear complementary problem (LCP) [85]. The LCP formulation in [83] included inverse of stiffness matrix (i.e. compliance matrix). Therefore real-time computation of contact problems which requires updating the compliance matrix (i.e. due to material or geometric nonlinearity) with large number of contact points, can be prohibitively expensive. In order to compute the tangential forces in contact, Coulomb's friction law was used in [83].

Courtecuisse et al. [63] employed a similar contact formulation in soft-tissue deformation to that presented in [83]. To address the challenges in updating the compliance matrix in the presence of large deformation, a compliance warping technique was employed where the inverse of stiffness matrix in deformed configuration was approximated by the inverse of undeformed stiffness matrix and rotation matrices derived by co-rotational FEM [65]. However, co-rotational FEM approximates the stiffness matrix in the presence of geometric nonlinearity and does not compute the stiffness changes due to material nonlinearity.

A summary of computational contact mechanics can be found in [86–88].

2.3 Parallel Computing in Deformable Models

The majority of the recent research studies for developing real-time simulation tools in deformation analysis are based on parallel computing. The natural motivating factor is to gain the maximum computing power of the modern processors in order to meet the performance criteria in real-time applications. The following two sections review some of the recent work in parallel computation of the popular physics-based models, i.e. mass-spring and FEM-based models.

2.3.1 Parallel Implementation of the Mass-Spring Systems

Mass-spring model simulation can benefit significantly from GPU computing due to potentially large number of mass points and the inherent data independency for the computations per mass point in steps (a), (b) and (c) mentioned in Sec. 2.1.1. References [52] and [89] presented two of the earliest studies on GPU-based modeling of mass-spring systems. Both references used explicit Verlet [90] time integration and OpenGL API to program the GPUs. At the time, GPU programming APIs for general purpose and scientific computing such as CUDA [91] and OpenCL [92] had not been developed yet. To perform step (a) in mass-spring computations, one may compute the forces per spring link and *scatter* these force values over the mass points in the mesh. One other approach is to *gather* the forces from adjacent links and add them together per mass point. Georgii and Westermann [52] implemented these two methods and concluded the scatter method yields better performance compared to the gather method in their GPU implementation.

Leon et al. [93] presented a comparative study of mass-spring simulation on serial CPU, multi-thread CPU and CUDA-based GPU. A penalty solver algorithm

was employed to solve the resulting differential equations. The data structure presented in [93] aimed at efficient use of shared and global memory of GPU. A speed gain of 20 times was reported on 8800 GT Nvidia GPU versus a 2.2 GHz Intel Quad core processor.

2.3.2 Parallel Computing for the Finite Element Method

Parallel computation of finite element analysis on multi-processors has been well developed both in theory and practice over the past two decades [94, 95]. Many of today's FEM software packages such as Abaqus [96], ANSYS [97], RS^3 [98] and COMSOL Multiphysics [99] perform the matrix computing, assembly and solution steps in parallel on distributed or shared memory processors. Rao [100] reported three different formulations of parallel computation of implicit dynamic nonlinear FEM based on domain decomposition. In this work, Message Passing Interface (MPI) [101] was used to provide a parallel implementation on multiple multi-core processors. The results indicated close to linear scalability with respect to the number of processors up to 32. Paz et al. [102] presented the details of a hybrid parallel computation of different stages of FEM. They used MPI for communication between a computer cluster nodes and OpenMP [35] for parallel computing on each multi-core node.

Linear finite element model is employed in some real-time applications to reduce the computing load. In [12], we proposed an FPGA-based micro-architecture for parallel solution of equations derived by a linear elastic finite element model. This architecture was successfully employed in real-time haptic interaction with deformable objects in static and implicit dynamic analysis.

GPU-based implementations of nonlinear finite element models have been presented in a number of previous papers [61, 62, 103]. Taylor et al. [61] used the total Lagrangian explicit dynamic finite element method on tetrahedral meshes. In [61], a 16-fold speed up compared with a CPU implementation was reported. Multi-point contact constraints were not supported in this study.

Joldes et al. [103] proposed a GPU-based nonlinear finite element method for explicit dynamic analysis with application to neurosurgery. This work considered Neo-Hookean material, different element types and deformable/rigid contacts, attaining a speed-up of more than 20 times compared with a CPU implementation.

Dick et al. [62] presented an approach based on the NVIDIA GPU architectures for real-time simulation of deformable objects. A linear elastic material model on a regular hexahedral grid was used. A regularly structured grid allowed using multigrid solver [104] with fast convergence behvaior. Since all the finite elements had a similar shape, the stiffness matrix needed to be computed only once, greatly reducing memory footprint and computing time.

2.4 Solving Linear System of Equations

Numerical solution of the system of equations in static or implicit dynamic analysis is often the most computationally intensive step in the process of finite element analysis of interacting deformable objects. To solve the system of equations in FEM, a set of local stiffness matrices computed per element are usually assembled into one global sparse matrix. Matrix assembly requires large number of memory accesses and relatively few arithmetic operations. Therefore, in order to achieve high performance, an optimized data access pattern and efficient use of cache are

necessary. Several strategies for parallel implementation of matrix assembly on CPU and GPU are presented in [105–107]. In an alternative approach referred to as *element-by-element*, it is possible to forgo matrix assembly and directly use the elemental matrices [108, 109].

Methods for solving the linear system of equations derived by the FEM are generally grouped into *direct* [110] and *iterative* [111] algorithms. Compared to direct methods, iterative algorithms are, in general, more efficient in terms of memory usage for solving a large sparse system of equations and offer better computing parallelism. The preconditioned conjugate gradients method is an iterative algorithm that has been widely used in real-time applications of the finite element method. In [12], we proposed a Jacobi-conditioned conjugate gradient solver based on FPGA for solving the sparse system of linear equations arising from the FEM. This micro-architecture employed hundreds of processing elements running in parallel with access to a customized memory architecture to speed up the computations by a factor of 150-250 compared to conventional CPUs at the time. Concurrent use of 360 processing cores running at 100 MHz and a memory structure designed to provide a high bandwidth on an Altera Stratix III EP3SE110 FPGA board enabled 72 giga fixed-point operations per second. The non-zero components of each row in the sparse matrix were padded to have the same length for parallel processing. Mahdavikhah et al. [112] extended the proposed micro-architecture for solving the preconditioned CG in [12] to multiple FPGAs.

Courtecuisse et al. [63] employed a GPU-based element-by-element CG algorithm to solve the system of equations derived by backward Euler implicit integration in FE-based dynamic analysis of deformable objects. The authors of [63]

reported 15 to 35-fold speed-up on a GeForce GTX 280 compared to a sequential implementation on an Intel Core 2 Quad processor running at 3.0 GHz. However, there was no comparison made with a conventional CG method with an assembled sparse matrix input argument.

Weber et al. [107] presented a new data structure for sparse matrix storage on GPU to assemble elemental matrices and efficiently access data to perform the CG algorithm with Jacobi preconditioning. The proposed sparse matrix structure in [107] allowed for a higher GPU memory bandwidth compared to standard storage formats such as Compressed Sparse Row (CSR) available on CUDA sparse matrix library, achieving about 25-40 single-precision GFLOPS for sparse matrix by vector multiplication on a GeForce GTX 470.

Cevahir et al. [113] proposed a distributed memory implementation of the CG algorithm on a GPU-extended cluster. In [114, 115] other implementations of CG method on multiple GPU were presented. A multiple GPU-based implementation of element-by-element Jacobi-conditioned CG was reported in [116].

Dick et al. [62] proposed a numerical multigrid solver entirely realized on GPU using CUDA API. They employed a matrix-free formulation in which the stiffness matrices per element are *not* assembled into a global matrix. This allows for a direct mapping of the computations per element onto GPU implementation. Their parallel simulation on a GTX 480 graphic card achieved a performance gain of 27 times compared to a single-threaded implementation on a 3.2 GHz Intel Xeon X5560 processor.

Helfenstein and Koko [117] presented a preconditioned CG algorithm using approximate inverse matrix computed based on Symmetric Successive Over-Relaxation

(SSOR) preconditioning. They adopted CSR storage format for the sparse matrix, achieving up to 10 times speed gain on NVIDIA Tesla T10 GPU versus Intel Xeon Quad-Core 2.66 GHz. A comprehensive review of parallel implementation of different numerical solvers for linear systems of equations can be found in [118, 119].

CHAPTER 3

PHYSICS OF DEFORMATION BASED ON CONTINUUM MECHANICS

Continuum mechanics provides a physically accurate framework for modeling deformation of biological soft tissue subject to force or displacement boundary constraints. In this theory, the object is treated as a continuum mass rather than discrete particles. Despite the fact that materials are made of atoms, this assumption is highly accurate in scales much larger than atomic scale. In this chapter, the following subjects from the continuum mechanics will be reviewed [18]:

- Description of deformation and strain
- Force in continuum mechanics and stress measures
- Constitutive equations for linear-elastic and hyper-elastic materials
- Principle of virtual displacement

3.1 Deformation Description

Consider states of an object undergoing deformation at two different time instants, where point ${}^0\mathbf{x}$ in the initial configuration is mapped to point ${}^\tau\mathbf{x}$ in the current configuration. Fig. 3.1 demonstrates this transformation which can be represented with a displacement vector ${}^\tau\mathbf{u}$. The Cartesian coordinate system in which the motions are measured is assumed to be stationary and is represented by the X_1, X_2 and X_3 axes.

$$ {}^\tau\mathbf{x} = {}^0\mathbf{x} + {}^\tau\mathbf{u} \tag{3.1}$$

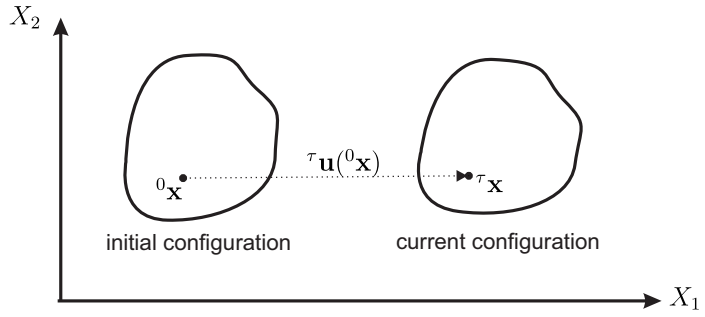


Figure 3.1: Displacement of a particle in two different states.

The components of vectors ${}^0\mathbf{x}$, ${}^\tau\mathbf{x}$ and ${}^\tau\mathbf{u}$ are denoted by a right subscript i, ${}^0x_i$, ${}^\tau x_i$ and ${}^\tau u_i$ where $i = 1,2,3$.

There are two different approaches in describing the mechanical quantities in continuum mechanics analysis: one is based on undeformed configuration, known as *material* or *Lagrangian* description and the other is with reference to deformed state, known as *spatial* or *Eulerian* description [18]. Lagrangian description concerns with the behavior at point ${}^0\mathbf{x}$ in the material, whereas Eulerian description is focused at point ${}^\tau\mathbf{x}$ which can be occupied with different material particles at different times.

Eulerian description is mainly employed in fluid analysis in which an initial configuration as steady-state flow does not exist [120]. In solid mechanics, Lagrangian description is primarily employed as the constitutive equations are defined based on material description [18]. The Lagrangian and Eulerian descriptions are related to each other through mapping of current and initial positions in Eq.(3.1). Since the focus of this thesis is on studying solid deformable objects, Lagrangian description is employed.

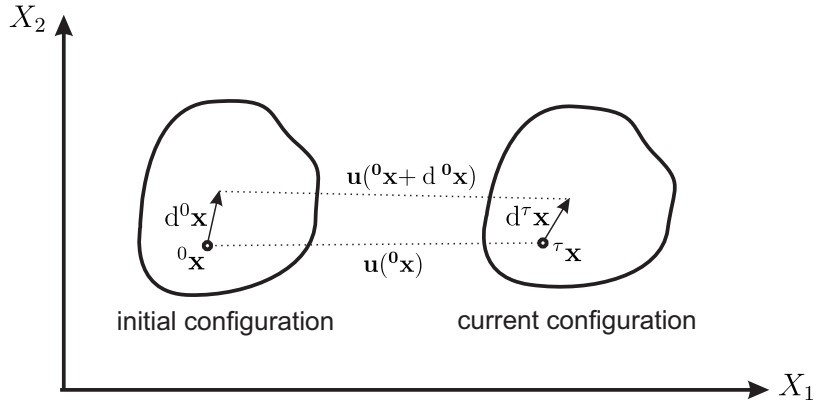


Figure 3.2: Deformation of a line segment in two different states.

Now consider the element line segment $\mathrm{d}^0\mathbf{x}$ in the initial configuration in Fig. 3.2, which is deformed into $\mathrm{d}^\tau\mathbf{x}$ in the current configuration. These two line segments are related to each other through the *deformation gradient* tensor $\boldsymbol{F}$ as

$$\mathrm{d}^\tau\mathbf{x} = \boldsymbol{F}\mathrm{d}^0\mathbf{x} \quad \text{where} \quad \boldsymbol{F} = \begin{pmatrix} \frac{\partial^\tau x_1}{\partial^0 x_1} & \frac{\partial^\tau x_1}{\partial^0 x_2} & \frac{\partial^\tau x_1}{\partial^0 x_3} \\ \frac{\partial^\tau x_2}{\partial^0 x_1} & \frac{\partial^\tau x_2}{\partial^0 x_2} & \frac{\partial^\tau x_2}{\partial^0 x_3} \\ \frac{\partial^\tau x_3}{\partial^0 x_1} & \frac{\partial^\tau x_3}{\partial^0 x_2} & \frac{\partial^\tau x_3}{\partial^0 x_3} \end{pmatrix} \tag{3.2}$$

Using tensor index notation, the deformation gradient can be represented as

$$F_{ij} = \partial^{\tau} x_i / \partial^{0} x_j \quad \text{for} \quad i, j = 1, 2, 3 \tag{3.3}$$

$\boldsymbol{F}$ is non-singular [120]. It is straightforward to show that the determinant of the deformation gradient tensor, $det(\boldsymbol{F})$, gives the scale factor by which an infinitesimal volume changes from initial to the current configuration [18]. In Fig. 3.3, the current volume $\mathrm{d}v$ spanned by line segments $\mathrm{d}^{\tau}\mathbf{x}$, $\mathrm{d}^{\tau}\mathbf{y}$ and $\mathrm{d}^{\tau}\mathbf{z}$ is related to the initial volume $\mathrm{d}V$ as

$$\mathrm{d}v = J\mathrm{d}V \tag{3.4}$$

Notation J is used to denote $det(\boldsymbol{F})$.

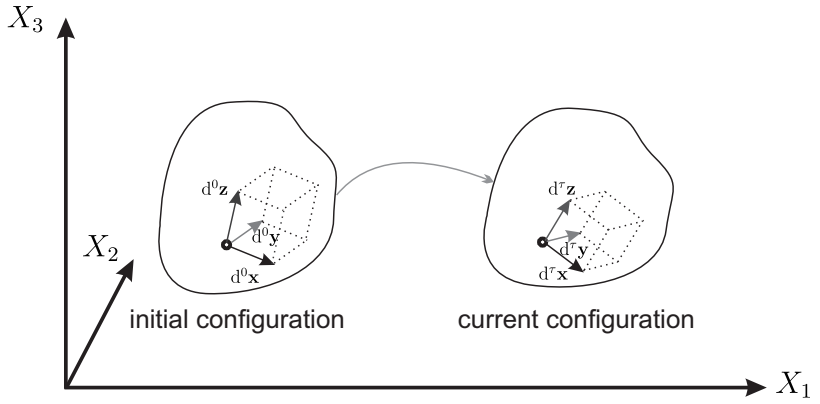


Figure 3.3: Change of infinitesimal volume.

The surface of a segment is represented by a vector with a magnitude equal to the area and a direction along the normal of the surface. Eq.(3.4) can be used to map the surface area from undeformed to deformed state. To this end, one may consider an arbitrary line segment $\mathrm{d}\mathbf{H}$ forming volume $\mathrm{d}V$ with the surface area $\mathrm{d}\mathbf{S}$ (see Fig. 3.4). $\mathrm{d}\mathbf{S}$ in the initial configuration is mapped to $\mathrm{d}\mathbf{s}$ in the current

configuration. Line segment d**H** is related to d**h** in the deformed configuration through the deformation gradient tensor, i.e. d**h** = ***F***d**H**. As shown in Fig. 3.4, volume dV on the left side is mapped to volume dv on the right using Eq.(3.4).

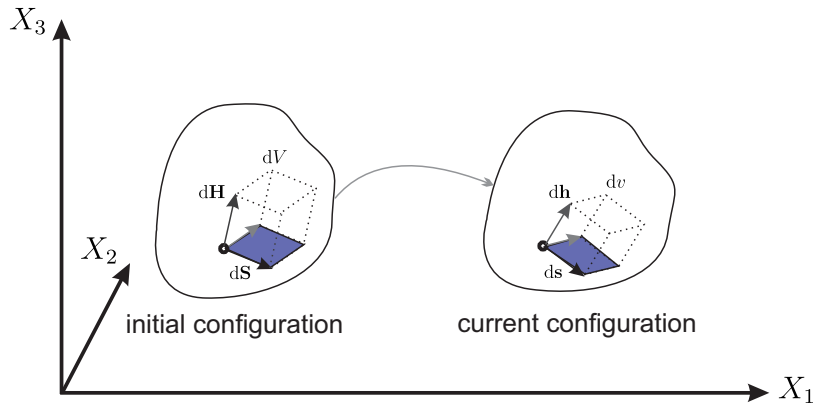


Figure 3.4: Change of infinitesimal surface area.

$$\mathrm{d}v = J\mathrm{d}V \quad \Rightarrow \tag{3.5a}$$

$$\mathrm{d}\mathbf{s}.\mathrm{d}\mathbf{h} = J\mathrm{d}\mathbf{S}.\mathrm{d}\mathbf{H} \tag{3.5b}$$

$$\mathrm{d}\mathbf{s}.\boldsymbol{F}\mathrm{d}\mathbf{H} = J\mathrm{d}\mathbf{S}.\mathrm{d}\mathbf{H} \tag{3.5c}$$

Since (3.5c) holds for any d**H**, it can be concluded that,

$$\mathrm{d}\mathbf{s} = \boldsymbol{F}^{-\mathrm{T}} J\mathrm{d}\mathbf{S} \tag{3.6}$$

The volume and surface mappings discussed above provide some good insight into understanding different measures of stress that will be introduced in Section 3.3.

3.2 Strain Measures

One measure of strain can be given by considering the elongation of a line segment. When the change in square of length of $\mathrm{d}^{\tau}\mathbf{x}$ is expressed in terms of the initial line segment $\mathrm{d}^{0}\mathbf{x}$ using the deformation gradient, the *Green* or *Lagrangian* strain tensor, $\boldsymbol{E}$, can be defined as

$$\mathrm{d}^{\tau}\mathbf{x}.\mathrm{d}^{\tau}\mathbf{x} - \mathrm{d}^{0}\mathbf{x}.\mathrm{d}^{0}\mathbf{x} = \mathrm{d}^{0}\mathbf{x}^{\mathrm{T}}\left(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} - \boldsymbol{I}\right)\mathrm{d}^{0}\mathbf{x} = \mathrm{d}^{0}\mathbf{x}^{\mathrm{T}}.2\boldsymbol{E}\mathrm{d}^{0}\mathbf{x} \tag{3.7a}$$

$$\boldsymbol{E} \triangleq \frac{1}{2}(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F} - \boldsymbol{I}) \tag{3.7b}$$

where $\boldsymbol{I}$ is the identity tensor. If the change of element segment scalar product is expressed in terms of current segment $\mathrm{d}^{\tau}\mathbf{x}$, another expression for strain would be derived [18]. This is denoted as *Almansi* strain tensor, $\boldsymbol{e}$, and is given by

$$\mathrm{d}^{\tau}\mathbf{x}.\mathrm{d}^{\tau}\mathbf{x} - \mathrm{d}^{0}\mathbf{x}.\mathrm{d}^{0}\mathbf{x} = {}^{\tau}\mathbf{x}^{\mathrm{T}}\left(\boldsymbol{I} - \boldsymbol{F}^{-\mathrm{T}}\boldsymbol{F}^{-1}\right)^{\tau}\mathbf{x} = \mathrm{d}^{\tau}\mathbf{x}^{\mathrm{T}}.2\boldsymbol{e}\mathrm{d}^{\tau}\mathbf{x} \tag{3.8a}$$

$$\boldsymbol{e} \triangleq \frac{1}{2}\left(\boldsymbol{I} - \boldsymbol{F}^{-\mathrm{T}}\boldsymbol{F}^{-1}\right) \tag{3.8b}$$

The Green strain is used to formulate deformation in the rest of this thesis. An interested reader is referred to two reference books in the field [18,121], for further discussion of the Almansi strain.

From (3.2), it can be observed that

$$\mathrm{d}^{\tau}\mathbf{x}.\mathrm{d}^{\tau}\mathbf{x} = \mathrm{d}^{0}\mathbf{x}^{\mathrm{T}}\left(\boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}\right)\mathrm{d}^{0}\mathbf{x} \tag{3.9}$$

The symmetric *right Cauchy-Green* deformation tensor is defined as $\boldsymbol{C_r} \triangleq \boldsymbol{F}^{\mathrm{T}}\boldsymbol{F}$. $\boldsymbol{C_r} = \boldsymbol{I}$ represents the case of pure rigid-body displacement or rotation with no

change in the length of any element segment. Tensor invariants of $\boldsymbol{C_r}$ have proven useful in defining constitutive equations discussed in Section 3.4.2. The Green strain tensor $\boldsymbol{E}$ in (3.7b) can be expressed in terms of the right Cauchy-Green tensor.

$$\boldsymbol{E} = \frac{1}{2}(\boldsymbol{C_r} - \boldsymbol{I}) \tag{3.10}$$

It would be instructive to express the Green strain tensor in terms of derivatives of the deformation vector. However, before proceeding it is necessary to lay out the notation which will be employed. Corresponding components in the coordinate system are denoted by right sub-index which can be numeric or represented by i, j, k, or l, varying from 1 to 3. For example ${}^{\tau}u_2$ denotes the displacement component along the second Cartesian axis, or E_{ij} represents strain tensor component along i and j coordinates. Throughout this thesis, the *summation convention*, also known as *Einstein convention*, will be employed [122]. According to this convention, repetition of an index indicates its summation. For example $F_{ij}F_{ik} = F_{1j}F_{1k} + F_{2j}F_{2k} + F_{3j}F_{3k}$.

Using the index notation, the Green strain tensor $\boldsymbol{E}$ from (3.10) equals to

$$E_{ij} = \frac{1}{2}(F_{ki}F_{kj} - \delta_{ij}) \tag{3.11}$$

where δ_{ij} is Kronecker delta defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{3.12}$$

On the other hand, $F_{ij} = \partial^\tau x_i / \partial^0 x_j$ after substituting ${}^\tau x_i$ by ${}^0 x_i + {}^\tau u_i$ from (3.1), can be written as $F_{ij} = \partial^\tau u_i / \partial^0 x_j + \delta_{ij}$. Replacing this form of the deformation gradient into Eq.(3.10) and after some simplification, the Green strain tensor of the current configuration is derived in terms of derivatives of the current deformation vector

$$E_{ij} = \frac{1}{2}(\underbrace{\partial^\tau u_i / \partial^0 x_j + \partial^\tau u_j / \partial^0 x_i}_{\text{linear term}} + \overbrace{\partial^\tau u_k / \partial^0 x_i . \partial^\tau u_k / \partial^0 x_j}^{\text{quadratic term}}) \tag{3.13}$$

In small deformations, the quadratic term in (3.13) can be ignored. This leads to *linear strain* model, also known as *engineering strain*,$\boldsymbol{\varepsilon}$.

$$\boldsymbol{\varepsilon} = \frac{1}{2}(\nabla^\tau \mathbf{u} + \nabla^\tau \mathbf{u}^{\mathrm{T}}) \quad \text{or equivalently} \quad \varepsilon_{ij} = \frac{1}{2}(\partial^\tau u_i / \partial^0 x_j + \partial^\tau u_j / \partial^0 x_i) \tag{3.14}$$

However, the quadratic term in the Green strain enables accurate modeling in presence of large deformations and rotations; this is the measure of strain adopted in this thesis to address geometric nonlinearity. The use of a linear strain model in large rotation/deformation results in artificial unacceptable distortions. Examples and further discussion in this regard can be found in [123].

3.3 Stress Measures

Stress represents internal force intensity developed within an object in response to external boundary constraints, i.e. force or displacement. Traction vector associated with vector $\mathbf{n}$ is defined as force per unit area with normal $\mathbf{n}$,

$$\mathbf{t}(\mathbf{n}) \triangleq \lim_{\Delta a \to 0} \frac{\Delta \mathbf{p}}{\Delta a} \tag{3.15}$$

where Δa is an infinitesimal area with normal $\mathbf{n}$ and $\Delta\mathbf{p}$ is the resultant force vector applied on this area. *Cauchy stress tensor* represented by $\boldsymbol{\sigma}$, is defined in a way to relate normal vector $\mathbf{n}$ to traction vector $\mathbf{t}$ [18],

$$\mathbf{t}(\mathbf{n}) = \boldsymbol{\sigma}\mathbf{n} \tag{3.16}$$

It is noted that Cauchy stress tensor is defined in the current deformed state and is used in conjunction with small deformation strains. For large deformations, some other stress measures such as *Piola-Kirchhoff stress* tensors need to be defined. Sometimes in case of stress analysis of deformable objects undergoing large deformations, it is desired to express stress tensor with reference to the initial configuration where the area is known.

A stress tensor that relates the force in the current state to the area in the initial state, should yield the same traction as Cauchy stress tensor. Using Eq.(3.15) and (3.16), the following equality should hold

$$\begin{aligned} \mathrm{d}\mathbf{p} = \mathrm{d}a\mathbf{t}(\mathbf{n}) &= \mathrm{d}a\boldsymbol{\sigma}\mathbf{n} \\ &= \mathrm{d}A\boldsymbol{\sigma}_{\boldsymbol{p}}\mathbf{N} \end{aligned} \tag{3.17}$$

where $\mathrm{d}\mathbf{p}$ is the resultant force vector at the current state, $\boldsymbol{\sigma}_{\boldsymbol{p}}$ is the 1^{st} *Piola-Kirchhoff* stress tensor, $\mathrm{d}A$ is the mapping of infinitesimal area segment $\mathrm{d}a$ at the initial configuration and $\mathbf{N}$ is the normal vector of $\mathrm{d}A$. By assigning $\mathrm{d}\mathbf{s} \triangleq \mathrm{d}a\mathbf{n}$ and $\mathrm{d}\mathbf{S} \triangleq \mathrm{d}A\mathbf{N}$, Eq.(3.6) can be used to map the area parameters in (3.17),

$$\boldsymbol{\sigma}(\mathrm{d}a\mathbf{n}) = \boldsymbol{\sigma}_{\boldsymbol{p}}(\mathrm{d}A\mathbf{N}) \tag{3.18a}$$

$$\boldsymbol{\sigma}\left(J\boldsymbol{F}^{-\mathrm{T}}\mathrm{d}A\mathbf{N}\right) = \boldsymbol{\sigma}_{\boldsymbol{p}}(\mathrm{d}A\mathbf{N}) \tag{3.18b}$$

Since the equation in (3.18b) is valid for any given $\mathrm{d}A\mathbf{N}$, 1^{st} Piola-Kirchhoff stress tensor is derived as follows

$$\boldsymbol{\sigma}_{\boldsymbol{p}} = J\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}} \tag{3.19}$$

Here the stress tensor $\boldsymbol{\sigma}_{\boldsymbol{p}}$ gives the actual force based on the initial state. However, there are some problems in the application of 1^{st} Piola-Kirchhoff stress in numerical computations. The tensor $\boldsymbol{\sigma}_{\boldsymbol{p}}$, unlike the Cauchy stress $\boldsymbol{\sigma}$, is not symmetric. Additionally this description is not work conjugate of the Green tensor, i.e. the product of the 1^{st} Piola-Kirchhoff stress and the Green strain rate is not equivalent to the work per current volume [18].

Let $\mathrm{d}\mathbf{P}$ be defined as a mapping of force vector $\mathrm{d}\mathbf{p}$ from Eq.(3.17) in the undeformed configuration, i.e. $\mathrm{d}\mathbf{p} = \boldsymbol{F}\mathrm{d}\mathbf{P}$. Then 2^{nd} *Piola-Kirchhoff* stress tensor, denoted by $\mathbf{S}$, is defined to express $\mathrm{d}\mathbf{p}$ in terms of area at the initial configuration.

$$\text{by definition} \qquad \mathrm{d}\mathbf{P} = \mathbf{S}\mathrm{d}A\mathbf{N} \tag{3.20a}$$

$$\text{from (3.17) and (3.19)} \qquad \mathrm{d}\mathbf{p} = J\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}}\mathrm{d}A\mathbf{N} \tag{3.20b}$$

$$\Rightarrow \quad J\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}}\mathrm{d}A\mathbf{N} = \boldsymbol{F}\boldsymbol{S}\mathrm{d}A\mathbf{N} \tag{3.20c}$$

In (3.20c), 2^{nd} Piola-Kirchhoff stress tensor is related to Cauchy stress tensor as in

$$\boldsymbol{S} = J\boldsymbol{F}^{-\mathbf{1}}\boldsymbol{\sigma}\boldsymbol{F}^{-\mathrm{T}} \tag{3.21}$$

From Eq.(3.21) the symmetry of tensor $\boldsymbol{S}$ is evident.

3.4 Constitutive Equations

The stress and strain measures developed in the preceding sections are general and can be applied to all types of materials. These descriptions are not sufficient to distinguish between different material types and some relationships between stress and strain need to be established to describe the material behavior. These relationships are known as *constitutive equations*.

No change of material behavior under rigid body transformation would be expected, therefore constitutive equations need to be independent of the coordinate system used to describe the deformation. Materials with elastic behavior store energy under deformation and this energy is not dissipated. As such, elastic materials return to their initial configuration when the loads are removed. Where the stress can be derived from the stored energy function, the material is called *hyperelastic*. Constitutive equations in hyperelastic materials are not path dependent and the stored energy can be expressed based on the current state [121].

There are different classes of constitutive models, such as linear elastic, hyperelastic, viscoelastic and plastic among others. In this chapter we will limit the discussion to linear elastic and hyperelastic materials. A comprehensive study of different models can be found in [18, 121, 124].

3.4.1 Linear Elasticity

Robert Hooke in 1660 discovered there is a linear relation between the stretch and force applied on an elastic object [125]. Augustin-Louis Cauchy extended this important law in elasticity to three-dimensional elastic objects, which is known as *generalized Hooke's law*. In this law, the stress-strain relation is expressed in terms

of *elastic modulus tensor*, $\boldsymbol{C}$,

$$\boldsymbol{S} = \boldsymbol{C}\boldsymbol{E} \quad \text{or equivalently} \quad S_{ij} = C_{ijkl}E_{kl} \tag{3.22}$$

Using Voigt notation (see Appendix A), the fourth order elasticity modulus tensor can be expressed as a 6×6 matrix $\mathbf{C}$.* For homogeneous and isotropic elastic materials, this matrix is expressed as in (3.23)

$$\mathbf{C} = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \tag{3.23}$$

where parameters λ and μ are *Lamé* constants, describing linear elastic material characteristics [126]. Note that matrix $\mathbf{C}$ is symmetric. For a linear elastic material, matrix $\mathbf{C}$ is invariant.

3.4.2 Hyperelastic Material Model

In practice, few materials exhibit linear elastic behavior and in most cases some form of nonlinear relation between stress and strain is observed. *Hyperelastic model* has been developed for large deformations and is applicable to rubber-like substances [127]. This model is used for materials with nonlinear stress-strain relation where stress and constitutive tensors, as in Eq.(3.24) can be found according to strain energy density function Ψ, i.e.

*In the adopted notation, ***italic bold*** letters represent tensors and **upright bold** letters represent the corresponding compressed matrix form.

$$S_{ij} = \frac{\partial \Psi}{\partial E_{ij}} \qquad \text{and} \qquad C_{ijkl} = \frac{\partial^2 \Psi}{\partial E_{ij} \partial E_{kl}} \tag{3.24}$$

As such, the strain energy density function should be expressed in terms of the finite strain $\boldsymbol{E}$, or equivalently the right Cauchy-Green tensor $\boldsymbol{C_r}$. Since the energy function should be objective with respect to rotation and translation, it is defined in terms of principal invariants of $\boldsymbol{C_r}$ tensor. Tensor invariants do not change with rotation or translation of the coordinate system.

$$\Psi = \Psi(I_C, II_C, III_C) \tag{3.25}$$

Equation (3.25) is the general form of strain energy density function of compressible materials in terms of the invariants of right-Cauchy stress, $\boldsymbol{C_r}$. In case of incompressible behavior, the volume of material remains unchanged, therefore $J = 1$ and the energy function is only dependent on two invariants of I_C and II_C. These invariants are defined as [121]:

$$I_C = tr(\boldsymbol{C_r}) \tag{3.26}$$

$$II_C = \frac{1}{2}\left(tr(\boldsymbol{C_r})^2 - tr(\boldsymbol{C_r^2})\right) \tag{3.27}$$

$$III_C = det(\boldsymbol{C_r}) = J^2 \tag{3.28}$$

Different models are proposed for hyperelastic energy function, among which Neo-Hookean, Moony-Rivlin and Ogden models can be named [18, 128]. Neo-Hookean is a special case of the more general Moony-Rivlin and Ogden models. This constitutive model has been successfully applied in a number of previous studies for soft tissue modeling [103, 129]. In this thesis, compressible Neo-Hookean model is adopted. This model is characterized by its dependence on the

first and third invariants. Constitutive equations for compressible Neo-Hookean material are obtained from the following strain energy function [18],

$$\Psi = \frac{\mu}{2}(I_C - 3) - \mu(\ln J) + \frac{\lambda}{2}(\ln J)^2 \tag{3.29}$$

where λ and μ are the Lamé parameters defined based on material properties. It is noted that J, the determinant of the deformation gradient tensor, is the root square of the third principal invariant of the right Cauchy-Green tensor, III_C. The second Piola-Kirchhoff stress is obtained by taking derivatives of Ψ with respect to the Green strain tensor $\boldsymbol{E}$ [18],

$$\boldsymbol{S} = \frac{\partial \Psi}{\partial \boldsymbol{E}} = \lambda(\ln J)\ \boldsymbol{C_r}^{-1} + \mu(\boldsymbol{I} - \boldsymbol{C_r}^{-1}) \tag{3.30}$$

The Lagrangian elasticity tensor is found by carrying out differentiation on $\boldsymbol{S}$ with respect to $\boldsymbol{C_r}$ [130],

$$C_{ijkl} = \frac{\partial^2 \Psi}{\partial E_{ij} \partial E_{kl}} = \lambda C^{-1}_{r_{ij}} C^{-1}_{r_{kl}} + (\mu - \lambda\ \ln J)(C^{-1}_{r_{ik}} C^{-1}_{r_{jl}} + C^{-1}_{r_{il}} C^{-1}_{r_{jk}}) \tag{3.31}$$

This fourth order tensor $\boldsymbol{C}$ can be converted to a compressed matrix form (see Appendix A), which is preferable for matrix computations.

3.5 Principle of Virtual Displacement

Displacement-based finite element method, which will be discussed in Chapter 4, can be derived based on *principle of virtual displacement*; this is a form of principle of virtual work [20]. Consider a deformable body undergoing small virtual displacement $\delta\mathbf{U}$. The principle of virtual displacement states that at the equilibrium,

the internal and external virtual work applied as a result of virtual displacement would be equal. In other words, the work done by external forces through the virtual displacement on the body equals to the work done by internal stresses and the virtual strain caused by virtual displacement. The virtual displacement $\delta\mathbf{U}$ is chosen arbitrarily, but it should satisfy boundary constrains imposed on the body. Using compressed vector form of stress/strain tensors, this principle is formulated as follows

$$\underbrace{\int_V \mathbf{S}.\delta\mathbf{E}\mathrm{d}V}_{\text{internal virtual work}} = \underbrace{\int_V \mathbf{f_B}.\delta\mathbf{U}\mathrm{d}V + \int_S \mathbf{f_S}.\delta\mathbf{U}\mathrm{d}S + \sum_i \mathbf{R}_i.\delta\mathbf{U}}_{\text{external virtual work}} \tag{3.32}$$

On the right hand side of the equation, $\mathbf{f_B}$ represents body forces and the integration is over volume V, $\mathbf{f_S}$ represents surface forces and the integration is performed over the surface S and finally $\mathbf{R}_i$'s represent external concentrated loads applied on the deformable body.

Finite element modeling consists of two main steps, (a) establishing a weighted integral of the physical equilibrium equations, and (b) using interpolation functions to approximate the solution in terms of nodal variables of the finite elements [58]. The principle of virtual displacement enables deriving the weighted integral equation as the basis for finite element formulation discussed in Section 4.2.

CHAPTER 4

FEM FORMULATION

In this chapter, nonlinear finite discretization of static and dynamic deformation models will be formulated. In the preceding chapter, the strain and stress measures were discussed and the material characteristics were defined through constitutive equations. The principle of virtual displacement was introduced as an effective tool for deriving differential equations describing body deformation subject to external constraints. In this chapter, a numerical procedure for formulating these differential equations in discrete domain is discussed.

The derivation of the finite element method for non-linear static and dynamic analysis is a lengthy procedure. Therefore, only a brief overview of some of its critical steps leading to the final form of equations are presented here; the reader is referred to [20, 21, 40] for more details.

4.1 General Overview

The finite element method consists of the following basic steps [58]:

1. Partitioning geometry of the object model into smaller *elements*. This allows approximating complex geometries using elements with basic shapes. Partitioning yields a systematic approach for solving differential equations over regions of general shape. The collection of the elements is known as the FEM *mesh*. Each element in the mesh is associated with some *nodes*. The solution of differential equations in the discrete domain is obtained at these nodes.
2. Computing FEM matrices per element. The continuum domain variables are interpolated by nodal values per element. Differential equations are approximated based on these nodal values.
3. Assembling the elemental equations. This is done by considering continuity of the solution between the elements. Elemental equations can not be solved individually as the boundary constraints are available only for the entire mesh; therefore these equations are assembled into a global matrix form.
4. Applying boundary constraints to the global matrix form followed by solving the system of equations using a proper numerical method.

The above steps are summarized in Fig. 4.1.

After spatial discretization in step (1) and deriving the FEM matrices per element in step (2), the equation of deformation per element in static analysis is given in the following general form,

$$(\mathbf{K}_\mathbf{N}^\mathbf{e} + \mathbf{K}_\mathbf{L}^\mathbf{e})\mathbf{u}^\mathbf{e} = \mathbf{R}^\mathbf{e} - \mathbf{F}^\mathbf{e} \tag{4.1}$$

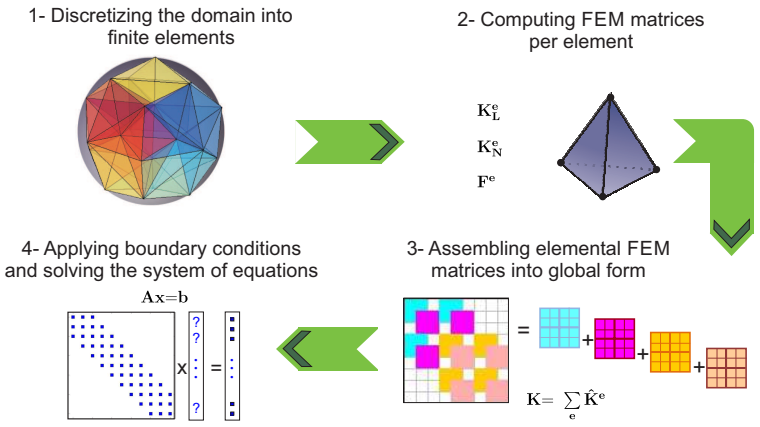


Figure 4.1: General steps in the finite element method.

Here the superscript **e** implies that the equation is defined over an element of the model. Moreover, $\mathbf{R^e}$ represents external forces acting on the element, $\mathbf{u^e}$ is the vector of nodal displacements with respect to the current time step, $\mathbf{K_L^e}$ and $\mathbf{K_N^e}$ are linear and non-linear components of stiffness matrix, and $\mathbf{F^e}$ is the vector of nodal internal forces caused by the element stresses. The deformation model taking into account dynamic effects has the following general form [20]

$$\underset{m\times m}{\mathbf{M^e}}\ \underset{m\times 1}{\ddot{\mathbf{u}}^{\mathbf{e}}} + \underset{m\times m}{\mathbf{D^e}}\ \underset{m\times 1}{\dot{\mathbf{u}}^{\mathbf{e}}} + \left(\underset{m\times m}{\mathbf{K_L^e}(_{\mathbf{0}}\mathbf{u^e})} + \underset{m\times m}{\mathbf{K_N^e}(_{\mathbf{0}}\mathbf{u^e})}\right)\underset{m\times 1}{\mathbf{u^e}} = \underset{m\times 1}{\mathbf{R^e}} - \underset{m\times 1}{\mathbf{F^e}(_{\mathbf{0}}\mathbf{u^e})} \tag{4.2}$$

where $\mathbf{M^e}$ and $\mathbf{D^e}$ are elemental mass and damping matrices, and $\dot{\mathbf{u}}^{\mathbf{e}}$ and $\ddot{\mathbf{u}}^{\mathbf{e}}$ are the first and second time derivatives of the nodal displacement vector per element, $\mathbf{u^e}$. $_{\mathbf{0}}\mathbf{u^e}$ within the parentheses indicates the dependency of the matrices to the nodal displacement vector with respect to the initial configuration. The dimension m in Eq.(4.2) depends on the model dimension and the number of the nodes per element, e.g. for a linear tetrahedral element $m = 3(\text{model dimension}) \times 4(\text{tetrahedral nodes})$. In most cases, the elemental mass and damping matrices $\mathbf{M^e}$ and $\mathbf{D^e}$ can be assumed constant and need not be recomputed in real time.

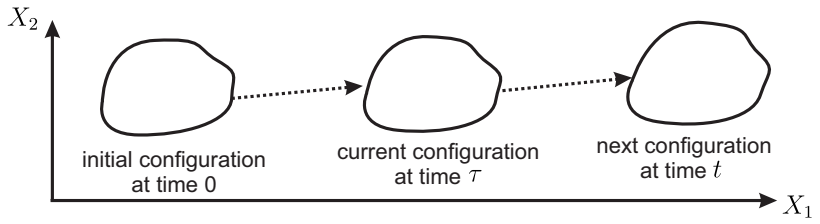


Figure 4.2: Configuration of a deformable body at different time increments.

There are two classes of numerical integration techniques for solving the differential equations in (4.2) in the discrete-time domain, namely *explicit* and *implicit* integrations. The explicit approaches such as the central difference method are easier to implement but are only *conditionally* stable. To maintain stability, the simulation time-step Δt must be smaller than a critical value Δt_{cr} which would depend on characteristics of the worst element in the mesh [20]. In contrast, implicit methods such as the Newmark integration scheme (see Appendix B) guarantee numerical stability independent of the time-step used [20]. They do, however, require solving a linear system of equations similar to that in (4.1) at each time-step.

A powerful strategy to solve nonlinear equilibrium equations is to adopt an incremental approach in which loads/displacements are applied in an adequate number of steps in virtual (in static analysis) or real (in dynamic analysis) time-steps. Equations of motion are solved at $time = 0,\ \Delta t,\ 2\Delta t, \ldots\ \tau,\ \tau + \Delta t$. Deformation quantities of the next time-step in Fig. 4.2 can be stated with reference to one of the previously known configurations, i.e. initial configuration in the Total Lagrangian (TL), or the last previously calculated configuration in Updated Lagrangian (UL) formulation. A consistent material description leads to the same result in both formulations. Since the TL formulation is expressed with respect to a fixed initial configuration, some of its spatial derivatives such as those in the Jacobian matrix

(see Appendix D), can be precomputed to reduce real-time computations [64]. The TL formulation is employed in this thesis.

4.2 Derivation of FEM Matrices

In this section, a brief formulation of the total Lagrangian finite element with large strains/displacements and material nonlinearity is presented. The Green strain tensor $\boldsymbol{E}$ defined in (3.10) accounts for geometric nonlinearity in large deformations. The second Piola-Kirchhoff stress tensor $\boldsymbol{S}$ is work conjugate of the Green strain and is adopted in the TL formulation. As mentioned in Section 3.4, the stress tensor $\boldsymbol{S}$ is related to the strain tensor $\boldsymbol{E}$ through the material constitutive tensor $\boldsymbol{C}$,

$$S_{ij} = C_{ijkl} E_{kl} \tag{4.3}$$

Tensor $\boldsymbol{C}$ is constant in linear elastic material. In a nonlinear material, this tensor is evaluated according to the material model as discussed in Section 3.4.2. It is useful to employ a notation to represent the time-step where a quantity (a) refers to (b) with reference to; such notation is particularly helpful in making distinction between next time-step unknown quantities and the current ones. A left superscript τ represents the current time-step and t is the next time-step. In ${}^{t}_{0}\boldsymbol{S}$ and ${}^{t}_{0}\boldsymbol{E}$, the left superscript and subscript denote that the stress and strain tensors are defined for the configuration at time t and with reference to the initial configuration at time 0.

Fig. 4.3 shows the important steps involved in numerically solving the partial differential equations arising from a continuum mechanics based model of the deformation using the method of finite element. These steps are summarized in the following subsections.

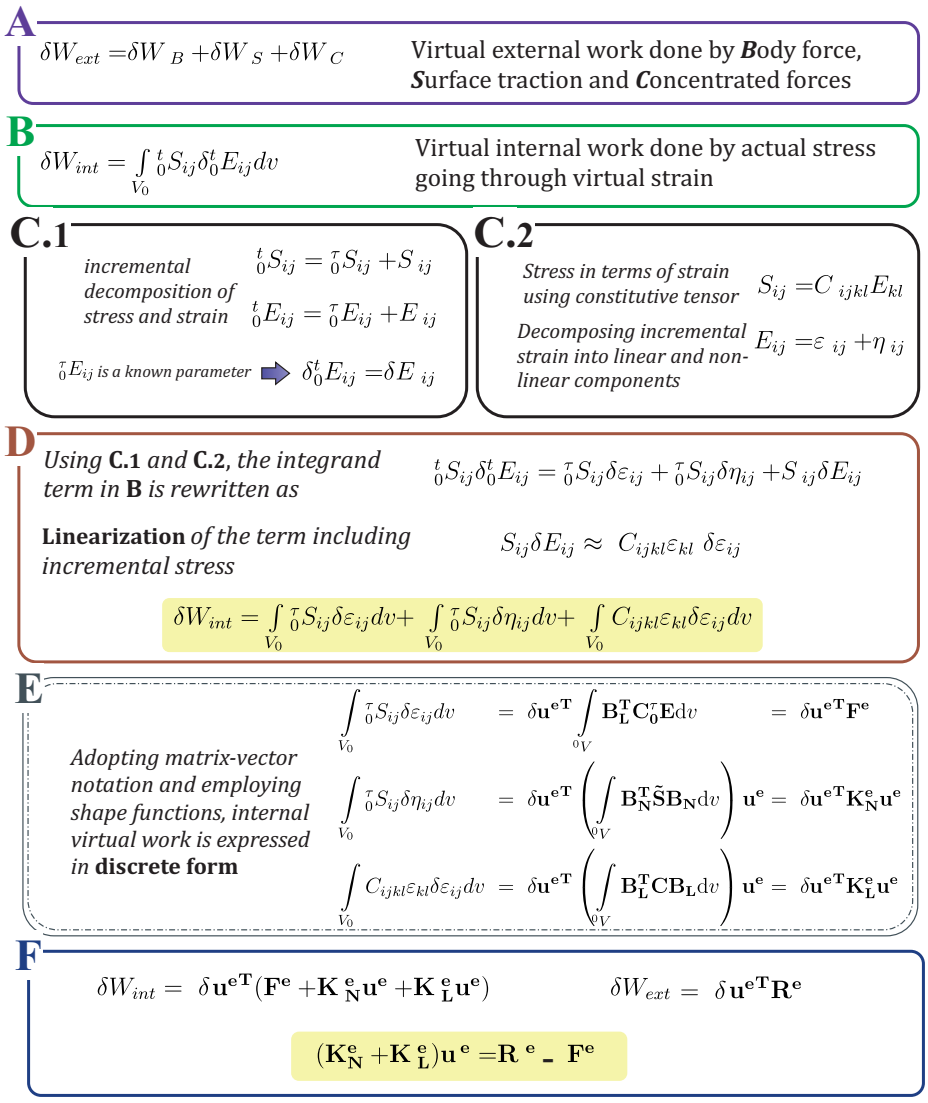


Figure 4.3: Establishing nonlinear finite element formulation based on the principle of virtual displacement.

4.2.1 The Differential Equation

As mentioned in Secion 3.5, the principle of virtual displacement gives a variational formulation of equilibrium equations in continuum mechanics which is used in displacement-based finite element formulation. The virtual external work in

Eq.(3.32), δW_{ext}, is expressed as a summation of work of different external forces (i.e. body force, surface traction and concentrated forces) going through the virtual displacement $\delta\mathbf{u}$. In dynamic analysis, virtual work due to the inertia and damping forces are added up to δW_{ext}. The virtual internal work at the next time-step t can be expressed in terms of the second Piola-Kirchhoff stress and the virtual Green strain tensors. Fig. 4.3.a and 4.3.b represent these equations.

4.2.2 Incremental Stress and Strain Terms

In the total Lagrangian incremental FEM formulation, the next time-step stress and strain tensors, ${}^{t}_{0}\boldsymbol{S}$ and ${}^{t}_{0}\boldsymbol{E}$, can be decomposed into known tensors at the current time-step with reference to the initial state, ${}^{\tau}_{0}\boldsymbol{S}$ and ${}^{\tau}_{0}\boldsymbol{E}$, and unknown incremental terms, $\boldsymbol{S}$ and $\boldsymbol{E}$. Furthermore, the Green strain tensor can be decomposed into linear and nonlinear terms defined in Eq.(3.13); $\boldsymbol{\varepsilon}$ is the linear component and $\boldsymbol{\eta}$ is the quadratic term of Green strain tensor. Fig. 4.3.c1 and c2 demonstrate these decompositions. Considering deformation increment $\mathbf{u}$ defined as in (4.4),

$$ {}^{t}\mathbf{u} = {}^{\tau}\mathbf{u} + \mathbf{u} \tag{4.4} $$

and using Green strain equation defined in (3.13), $\boldsymbol{\varepsilon}$ and $\boldsymbol{\eta}$ can be derived as follows

$$ \varepsilon_{ij} = (u_{i,j} + u_{j,i} + {}^{\tau}u_{k,i}u_{k,j} + u_{k,i}{}^{\tau}u_{k,j})/2 \tag{4.5} $$

$$ \eta_{ij} = u_{k,i}u_{k,j}/2 \tag{4.6} $$

where $u_{i,j} = \partial u_i/\partial^0 x_j$ and ${}^{\tau}u_{i,j} = \partial^{\tau} u_i/\partial^0 x_j$.

As given in Fig. 4.3.d, the stress and strain decompositions followed by linear approximation of strain in the incremental term result in approximate formulation of the virtual internal work,

$$\delta W_{int} = \int_{0_V} {}^{\tau}_{0}\boldsymbol{S}\delta\boldsymbol{\varepsilon}\ \mathrm{d}v + \int_{0_V} {}^{\tau}_{0}\boldsymbol{S}\delta\boldsymbol{\eta}\ \mathrm{d}v + \int_{0_V} \boldsymbol{C}\boldsymbol{\varepsilon}\delta\boldsymbol{\varepsilon}\ \mathrm{d}v \tag{4.7}$$

4.2.3 FEM Discretization

In finite element method, the continuum displacement field is expressed in terms of displacement values at the nodal points. *Interpolation functions*, also known as *shape functions*, set up a bridge between continuum and discrete domains. The continuum displacement field within a finite element domain can be expressed in terms of nodal displacement values, i.e.

$$\mathbf{u} = \sum_{i=1}^{ne} h_i \mathbf{u_i^e} \tag{4.8}$$

where h_i's are shape functions, $\mathbf{u_i^e}$ is the the displacement at node i of the element and ne is the number of nodes representing the element. For a discussion regarding shape functions see Appendix D. Using this discrete form of displacement in Eqs. (4.5) and (4.6), it is possible to define linear and nonlinear strain-displacement matrices $\mathbf{B_L}$ and $\mathbf{B_N}$ [40] such that the strain tensor components are given by

$$\boldsymbol{\varepsilon} = \mathbf{B_L}\mathbf{u^e} \qquad \text{and} \qquad \boldsymbol{\eta} = \mathbf{B_N}\mathbf{u^e} \tag{4.9}$$

where $\mathbf{u^e}$ represents the vector of nodal displacements. $\boldsymbol{\varepsilon}$ and $\boldsymbol{\eta}$ are vector form of linear and nonlinear strain tensor components defined as

$$\boldsymbol{\varepsilon} = [\varepsilon_{11} \quad \varepsilon_{22} \quad \varepsilon_{33} \quad 2\varepsilon_{23} \quad 2\varepsilon_{13} \quad 2\varepsilon_{12}]^{\mathrm{T}} \tag{4.10}$$

$$\boldsymbol{\eta} = [u_{1,1} \quad u_{1,2} \quad u_{1,3} \quad u_{2,1} \quad u_{2,2} \quad u_{2,3} \quad u_{3,1} \quad u_{3,2} \quad u_{3,3}]^{\mathrm{T}} \tag{4.11}$$

The strain expressions in terms of nodal displacement vector in Eq.(4.9) can be applied to the continuum differential equation derived by the principle of virtual displacement. To this end, the tensor differential equation in (4.7) can be rewritten in matrix form,

$$\delta W_{int} = \int_{{}_0V} \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\, {}^{\tau}_{0}\mathbf{S}\, \mathrm{d}v + \int_{{}_0V} \delta\boldsymbol{\eta}^{\mathrm{T}}\, {}^{\tau}_{0}\tilde{\mathbf{S}}\boldsymbol{\eta}\, \mathrm{d}v + \int_{{}_0V} \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\mathbf{C}\boldsymbol{\varepsilon}\, \mathrm{d}v \tag{4.12}$$

where $\mathbf{C}$ is the compressed material constitutive matrix extracted from tensor $\boldsymbol{C}$ (see Appendix A) and

$${}^{\tau}_{0}\mathbf{S} = [{}^{\tau}_{0}S_{11}\ {}^{\tau}_{0}S_{22}\ {}^{\tau}_{0}S_{33}\ {}^{\tau}_{0}S_{23}\ {}^{\tau}_{0}S_{13}\ {}^{\tau}_{0}S_{12}]^{\mathrm{T}} \tag{4.13}$$

$${}^{\tau}_{0}\tilde{\mathbf{S}} = \begin{bmatrix} {}^{\tau}\hat{\mathbf{S}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & {}^{\tau}\hat{\mathbf{S}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & {}^{\tau}\hat{\mathbf{S}} \end{bmatrix} \quad \text{where} \quad {}^{\tau}\hat{\mathbf{S}} = \begin{bmatrix} {}^{\tau}S_{11} & {}^{\tau}S_{12} & {}^{\tau}S_{13} \\ {}^{\tau}S_{21} & {}^{\tau}S_{22} & {}^{\tau}S_{23} \\ {}^{\tau}S_{31} & {}^{\tau}S_{32} & {}^{\tau}S_{33} \end{bmatrix} \quad \text{and} \quad \mathbf{0} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.14}$$

Now using Eq.(4.9) and vector form of Eq.(4.3), i.e. ${}^{\tau}_{0}\mathbf{S} = \mathbf{C}\, {}^{\tau}_{0}\mathbf{E}$, we have

$$\int_{{}_0V} \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\, {}^{\tau}_{0}\mathbf{S}\, \mathrm{d}v = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}} \int_{{}_0V} \delta\mathbf{B_L}^{\mathrm{T}}\mathbf{C}\, {}^{\tau}_{0}\mathbf{E}\, \mathrm{d}v = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}}\mathbf{F}^{\mathbf{e}} \tag{4.15}$$

$$\int_{{}_0V} \delta\boldsymbol{\eta}^{\mathrm{T}}\, {}^{\tau}_{0}\tilde{\mathbf{S}}\boldsymbol{\eta}\, \mathrm{d}v = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}} \left(\int_{{}_0V} \mathbf{B_N}^{\mathrm{T}}\, {}^{\tau}_{0}\tilde{\mathbf{S}}\mathbf{B_N}\, \mathrm{d}v \right) \mathbf{u}^{\mathbf{e}} = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}}\mathbf{K}^{\mathbf{e}}_{\mathbf{N}}\mathbf{u}^{\mathbf{e}} \tag{4.16}$$

$$\int_{{}_0V} \delta\boldsymbol{\varepsilon}^{\mathrm{T}}\mathbf{C}\boldsymbol{\varepsilon}\, \mathrm{d}v = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}} \left(\int_{{}_0V} \mathbf{B_L}^{\mathrm{T}}\mathbf{C}\mathbf{B_L}\, \mathrm{d}v \right) \mathbf{u}^{\mathbf{e}} = \delta\mathbf{u}^{\mathbf{e}\mathrm{T}}\mathbf{K}^{\mathbf{e}}_{\mathbf{L}}\mathbf{u}^{\mathbf{e}} \tag{4.17}$$

where $\mathbf{K}^{\mathbf{e}}_{\mathbf{L}}$ and $\mathbf{K}^{\mathbf{e}}_{\mathbf{N}}$, as mentioned earlier in this chapter, are linear and nonlinear components of the stiffness matrix, and $\mathbf{F}^{\mathbf{e}}$ is the nodal force vector due to the internal stress field. Fig. 4.3.e illustrates a summary where the internal virtual work

using shape functions is expressed in terms of FEM matrices and displacement at nodal points of the element.

4.2.4 Steady-state Equilibrium Equation

By setting equal δW_{int} and δW_{ext} in Fig. 4.3.f for any virtual displacement consistent with boundary conditions, the steady-state equilibrium is formulated as in Eq.(4.1). Static and dynamic equations per element in (4.1) and (4.2) usually need to be assembled into a global form to obtain the equilibrium equation for the complete model object. The assembled form of static and dynamic equations are

$$(\mathbf{K_N} + \mathbf{K_L})\mathbf{U} = \mathbf{R} - \mathbf{F} \tag{4.18}$$

$$\mathbf{M\ddot{U}} + \mathbf{D\dot{U}} + (\mathbf{K_N} + \mathbf{K_L})\mathbf{U} = \mathbf{R} - \mathbf{F} \tag{4.19}$$

The assemblage process maps and adds the elemental matrices into a global matrix [131]. Using the symbol of **A** for $\mathbf{K_L}$, $\mathbf{K_N}$, **M** and **D** matrices, and symbol of **x** for **U**, **R** and **F** vectors, FEM assembly is represented as

$$\mathbf{A} = \biguplus_{i=1} \mathbf{A_i^e} = \sum_i \hat{\mathbf{A}}_\mathbf{i}^\mathbf{e}, \qquad \mathbf{x} = \biguplus_{i=1} \mathbf{x_i^e} = \sum_i \hat{\mathbf{x}}_\mathbf{i}^\mathbf{e} \tag{4.20}$$

where accent $\hat{}$ denotes the mapping of the elemental parameter into a global parameter. For example, matrix $\hat{\mathbf{A}}_\mathbf{i}^\mathbf{e}$ has the same size as **A** with all its components except those corresponding to elemental matrix $\mathbf{A_i^e}$ set to zero. In the structure of matrix **A**, each row corresponds to x, y or $z-$ dimension of one mesh node in the FEM mesh. Nonzero entries in that row are limited to the number of neighbor nodes in connection to the corresponding node; therefore for a mesh with large number of nodes, matrix **A** is a large sparse matrix.

4.3 Computational Cost

As formulated in the previous section, the finite element vectors and matrices are computed as volume or surface integrals. These integrations can be carried out using a numerical technique such as the Gauss quadrature [20]. Gaussian quadrature is a numerical method to evaluate an integral as a weighted sum of the integrand function at specific points $\mathbf{x_i}$ (see Appendix C). The number and location of these points are chosen based on the element type and the required precision for the numerical integration [20]. Eq.(4.21) gives the general numerical integration form of $\mathbf{K_L^e}$, $\mathbf{K_N^e}$ and $\mathbf{F^e}$.

$$\mathbf{K_L^e} = \int_{{}^0V} \mathbf{B_L^T C B_L} \mathrm{d}v = \sum_i \omega_i \mathbf{B_L^T}(\mathbf{x_i}) \mathbf{C B_L}(\mathbf{x_i}) \tag{4.21a}$$

$$\mathbf{K_N^e} = \int_{{}^0V} \mathbf{B_N^T S B_N} \mathrm{d}v = \sum_i \omega_i \mathbf{B_N^T}(\mathbf{x_i}) \mathbf{S B_N}(\mathbf{x_i}) \tag{4.21b}$$

$$\mathbf{F^e} = \int_{{}^0V} \mathbf{B_L^T C E} \mathrm{d}v = \sum_i \omega_i \mathbf{B_L^T}(\mathbf{x_i}) \mathbf{C E}(\mathbf{x_i}) \tag{4.21c}$$

where ${}^0V$ is the volume of the initial element configuration, $\mathbf{B_L}$ and $\mathbf{B_N}$ refer to linear and non-linear strain-displacement matrices, $\mathbf{C}$ is the incremental material property matrix, $\mathbf{S}$ is the second Piola-Kirchhoff stress matrix, and $\mathbf{E}$ is the Green-Lagrange strain vector.

Table 4.1 gives examples of the dimensions of the local (elemental) matrices in the finite element analysis and the number of floating point operations for computing $\mathbf{K_L^e}$, $\mathbf{K_N^e}$ and $\mathbf{F^e}$. The numbers in this table do not include the computation cost of intermediate matrices such as $\mathbf{B_L}$, $\mathbf{B_N}$ and the second Piola-Kirchhoff stress matrix $\mathbf{S}$ in Eq.(4.21).

It is noted that some of the matrices contain fixed patterns of zeros. Considering

	Element	$\mathbf{B_L}$	$\mathbf{B_N}$	$\mathbf{C}$	$\mathbf{S}$	$\mathbf{E}$	$\mathbf{K_L^e}$	$\mathbf{K_N^e}$	$\mathbf{F^e}$
		Matrix size					Multiplications/additions		
2D	Triangle	3×6	4×6	3×3	4×4	3×1	$162\times,108+$	$240\times,180+$	$27\times,18+$
	Quadrilateral	3×8	4×8	3×3	4×4	3×1	$264\times,176+$	$384\times,288+$	$33\times,22+$
3D	Tetrahedron	6×12	9×12	6×6	9×9	6×1	$1296\times,1080+$	$2268\times,2016+$	$108\times,90+$
	Hexahedron	6×24	9×24	6×6	9×9	6×1	$4320\times,3600+$	$7128\times,6336+$	$180\times,150+$

Table 4.1: Dimensions of FE matrices for different types of elements. Number of flops in computing FE matrices are given for one integration point using Gauss-quadrature rule.

these patterns, redundant arithmetic operations can be eliminated. For example, the number of floating point operations (flops) in the computation of $\mathbf{K_N^e}$ can be reduced by 93%, by avoiding zero multiplications in a tetrahedral element with linear shape functions. However, such reductions would not be available in the cases of $\mathbf{K_L^e}$ and $\mathbf{F^e}$.

An accurate finite element model may consist of thousands of elements, resulting in a huge number of computations. Nonetheless, data independency of the elemental matrices and intensive arithmetic operations make this problem particularly suitable for parallel implementation on many-core architectures such as GPUs. Modern GPU devices consist of hundreds of processing cores and handle thousands of threads simultaneously. By assigning each GPU thread to compute one elemental matrix of the TL FEM, a high level of parallelism can be achieved for large FEM models. A detailed discussion of this matter will be provided in Chapter 7.

CHAPTER 5

FEM IN PRESENCE OF CONTACT

Deformable objects in contact are subject to external boundary constraints due to physical interaction. The contact forces depend very much on the geometry of the contact boundary. The contact geometry, itself changes based on boundary constraints, including contact forces. In this chapter, a mathematical formulation of contact in deformation analysis is presented. After some preliminary discussions about discretized contact surface, a penalty-based approach of contact formulation in FEM deformation analysis is reviewed. Contact surfaces can be exposed to large stress and strain gradients. Therefore, in order to achieve accurate numerical results, extra care should be taken in spatial discretization of contact domains. At the end of this chapter, a method for local mesh refinement at contact surfaces is presented to enhance the accuracy of computations of contact stresses.

5.1 Discretized Contact Surface

Considering contact of a hitting node, $\mathbf{x_h}$, with a target surface in Figure 5.1, *gap* is defined as the distance between the hitting node and the target surface along the normal direction, $\mathbf{N}$. Contact occurs when the hitting node penetrates the target surface and the gap becomes negative. Assuming a surface for the contact target, it is always possible to define an outward vector $\mathbf{N}$, in the normal direction to the surface. In case of a tetrahedral mesh, the surface patch would be triangular. The gap g can be expressed as

$$g = \mathbf{N^T}(\mathbf{x_h} - \mathbf{x_t}) \tag{5.1}$$

where $\mathbf{x_t}$ is the position vector of the nearest point on the target surface with respect to $\mathbf{x_h}$.

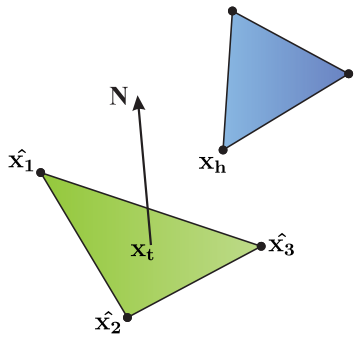


Figure 5.1: The *contact pair nodes* refer to the set of $[\mathbf{x_h}\ \hat{\mathbf{x_1}}\ \hat{\mathbf{x_2}}\ \hat{\mathbf{x_3}}]$

In the presence of contact, Eq.(4.2) has to be modified as follows

$$\mathbf{M^e\ddot{u}^e + D^e\dot{u}^e + (K_N^e + K_L^e)u^e = R^e + R_c^e - F^e} \tag{5.2}$$

where $\mathbf{R_c^e}$ denotes the contribution of contact forces per contact pair.

Using the Newmark scheme for implicit time integration, the solution for the next time step deformation is found as

$$\mathbf{K}^{\mathbf{e}}_{*}\ {}^{t}\mathbf{u}^{\mathbf{e}} = {}^{t}\mathbf{R}^{\mathbf{e}} + {}^{t}\mathbf{R}^{\mathbf{e}}_{\mathbf{c}} + {}^{\tau}\mathbf{R}^{\mathbf{e}}_{\mathbf{u}} \tag{5.3}$$

where $\mathbf{K}^{\mathbf{e}}_{*}$ is a linear combination of mass, damping and stiffness matrices and ${}^{\tau}\mathbf{R}^{\mathbf{e}}_{\mathbf{u}}$ is defined by a linear combination of ${}^{\tau}\mathbf{u}^{\mathbf{e}}$, ${}^{\tau}\dot{\mathbf{u}}^{\mathbf{e}}$ and ${}^{\tau}\ddot{\mathbf{u}}^{\mathbf{e}}$. t and τ left superscripts refer to the next and current time steps respectively. In the previous equations (4.1, 4.2 and 5.2), the left superscript representing time was dropped to simplify the notation. The time at which each term in (5.3) is evaluated is important and therefore is included in the notation here. To solve for the incremental deformation ${}^{t}\mathbf{u}^{\mathbf{e}}$, the contact force vector at the next time step ${}^{t}\mathbf{R}^{\mathbf{e}}_{\mathbf{c}}$ should be determined according to contact constraints.

Physical constraints require that two objects in contact do not overlap, i.e. $g \geq 0$. Also, no tensile traction is allowed on the contacting boundaries, i.e. $f_c \geq 0$, where f_c refers to the contact force magnitude at the hitting node. This implies zero interpenetration between the two non-rigid objects in contact and zero interaction force when the two objects are apart. These constraints are known as Signorini's law [84] and can be stated as

$$g \geq 0, \quad f_c \geq 0, \quad g.f_c = 0 \tag{5.4}$$

The complementary constraint $g.f_c = 0$ in (5.4), consisting of the gap function and the contact force product, is energy related. A physical interpretation of this constraint is that the net work done by equal and opposite contact forces in equilibrium is zero.

Contact problem can be viewed as an optimization problem for minimizing the energy function subject to inequality constraints in Eq.(5.4). There are different methods to impose contact constraints including the penalty method, the Lagrange multiplier method and combinations of the two [86].

The Lagrange multiplier approach enforces the zero interpenetration constraint. In this method, the contact constraints are treated as extra unknown parameters, introducing that many new equations. Depending on the number of contact points, the problem size can significantly increase.

In the penalty-based method, small penetrations are allowed between contact boundaries. Contact forces are calculated based on the penetration scaled by a penalty parameter, α. Physically, this is equivalent to introducing some springs with stiffness equal to α at the contact boundaries [132]. One drawback of the penalty method is dependency of its solution accuracy to the penalty parameter selection [132, 133]. In this approach, the contact constraints do not introduce any extra degrees of freedom to the problem at hand. But allowing penetration between the contact boundaries would violate the conditions stated in (5.4), resulting in an approximate solution. The work done by contact forces in the presence of penetration is not zero. This can be explained by storage of energy in the interface penalty springs [21]. This energy would be fully released after the contact.

Theoretically, it can be shown as the penalty parameter becomes infinitely large, the penalty method and Lagrange multiplier method would become equivalent [134]. However, in practice very large values of penalty parameter lead to ill-conditioned system of equations. Lack of robustness against interpenetration of contact surfaces in the penalty methods is more pronounced in simulation of thin

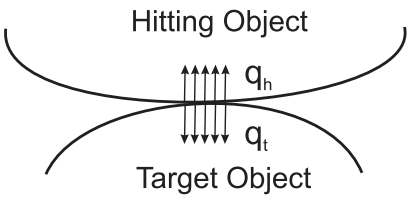


Figure 5.2: Contact between two non-rigid objects.

highly deformable cloth layers. But in the simulation of volumetric objects interacting with each other, the penalty method can be effectively employed. The penalty method is computationally more efficient than the other methods and can be easily integrated in a GPU-based computing platform, hence is adopted in this work.

5.2 Penalty-based Formulation of Contacts

Using the penalty method, the next time step contact force, ${}^t\mathbf{R_c^e}$, can be evaluated in terms of the next time step deformation, ${}^t\mathbf{u^e}$. To derive the penalty-based formula, the work done by the contact forces should be determined. Without loss of generality, consider the two objects interacting in Figure 5.2. In such case, the contact work equals to

$$W_c = \int_{\Gamma_c} (\mathbf{q}_h.\mathbf{u}_h + \mathbf{q}_t.\mathbf{u}_t) dS \tag{5.5}$$

where Γ_c is the contact area, $\mathbf{q}_h$ and $\mathbf{q}_t$ represent contact stress vectors (dimension 3×1 in a three-dimensional model) applied on the hitting and target objects respectively, and $\mathbf{u}_h$ and $\mathbf{u}_t$ are the corresponding deformation vectors. According to the Newton's third law of motion, $\mathbf{q}_h$ and $\mathbf{q}_t$ are equal in magnitude, but opposite in

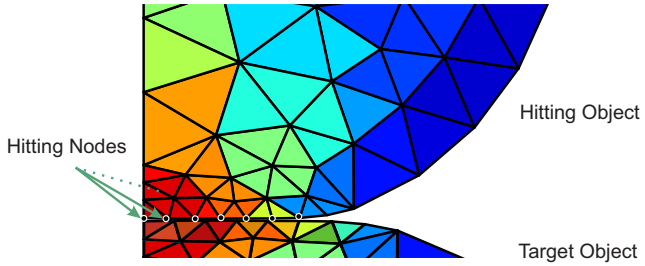


Figure 5.3: Contact in discrete form.

direction. By defining $\mathbf{q} \triangleq \mathbf{q}_t = -\mathbf{q}_h$, Eq.(5.5) can be rewritten as

$$W_c = \int_{\Gamma_c} \mathbf{q}.(\mathbf{u}_t - \mathbf{u}_h) dS \tag{5.6}$$

Considering the contact of two objects in Fig. 5.3 in discrete form as the collision of contact nodes on the hitting object with the contact surface on the target object, W_c can be stated as a summation of contact work of each hitting node,

$$W_c = \sum_n (w_c)_n \tag{5.7}$$

In a discretized finite elements formulation, the contact work per hitting node, w_c, can be expressed in terms of the concentrated force at that hitting node, f_c, as

$$w_c = f_c \mathbf{N}.(\mathbf{u}_t - \mathbf{u}_h) \tag{5.8}$$

Dot product by $\mathbf{N}$ represents the work along the normal direction. For simplicity, tangential forces are neglected in (5.8). This assumption is valid in the absence of frictional forces; otherwise Eq.(5.8) can be modified to include the friction work [21, 133]. Using interpolation (shape) functions, $\mathbf{u}_t$ can be expressed in terms of

nodal displacements of the element on target object that the contact point belongs to, i.e.

$$\mathbf{u}_t = \sum_i h_i \mathbf{u}^{\mathbf{e}}_{\mathbf{t}\,i} = \mathbf{H}^{\mathbf{e}} \mathbf{u}^{\mathbf{e}}_{\mathbf{t}} \tag{5.9}$$

Defining $\mathbf{u}^{\mathbf{e}}_{\mathbf{c}}$ as displacement vector of 'contact pair nodes', $\mathbf{u}^{\mathbf{e}}_{\mathbf{c}} = [\mathbf{u}_h^T,\ \mathbf{u}^{\mathbf{e}T}_{\mathbf{t}}]^T$ and $\mathbf{Q}^{\mathbf{e}} \triangleq [-\mathbf{I},\ \mathbf{H}^{\mathbf{e}}]$, the contact work of each hitting node in Eq.(5.8) can be rewritten as follows [21],

$$w_c = f_c \mathbf{N}.\mathbf{Q}^{\mathbf{e}} \mathbf{u}^{\mathbf{e}}_{\mathbf{c}} = \mathbf{u}^{\mathbf{e}T}_{\mathbf{c}} \mathbf{Q}^{\mathbf{e}T} f_c \mathbf{N} \tag{5.10}$$

Defining $\mathbf{R}^{\mathbf{e}}_{\mathbf{c}}$ as the nodal force vector contributed by the contact at the corresponding contact pair nodes in Eq.(5.11), the contact work in Eq.(5.10) can be replaced with $w_c = \mathbf{u}^{\mathbf{e}T}_{\mathbf{c}} \mathbf{R}^{\mathbf{e}}_{\mathbf{c}}$, where

$$\mathbf{R}^{\mathbf{e}}_{\mathbf{c}} = \mathbf{Q}^{\mathbf{e}T} f_c \mathbf{N} \tag{5.11}$$

In the penalty-based method, small penetrations are allowed and the contact force at the hitting node, f_c, is directly related to the penetration. For one hitting node, f_c is defined as

$${}^t f_c \triangleq -\alpha\ {}^t g \tag{5.12}$$

where α is the penalty parameter and ${}^t g$ is the measure of overlap at the next time step. The penalty parameter can be defined constant or as a function of penetration. As mentioned earlier, the magnitude of the penalty parameter affects the accuracy of the solution. A small choice of α would result in excessive interpenetration and may contribute to stress inaccuracies. On the other hand, very large

values for α would yield an ill-conditioned system of equations, causing convergence issues. Additionally, using large values of α in dynamic analysis may result in noisy and oscillatory response and unrealistic separation of the contact boundaries. A guideline for selecting an optimal penalty parameter for static contact is proposed in [135]. Penalty parameter for dynamic analysis can be selected by ad hoc approaches, such as defining it proportional to the Young's modulus of the material [132, 136].

Equations (5.1), (5.11) and (5.12) define the contact force vector at contact pair nodes, $\mathbf{R_c^e}$, in terms of the position of contact pair nodes. In the Newmark integration in (5.3), the contact force must be evaluated at the next time step. To this end, ${}^t g$ is derived as

$$\begin{aligned} {}^t g &= \mathbf{N^T}({}^t\mathbf{x}_t - {}^t\mathbf{x}_h) \\ &= \mathbf{N^T}({}^\tau\mathbf{x}_t - {}^\tau\mathbf{x}_h) + \mathbf{N^T}({}^t\mathbf{u}_t - {}^t\mathbf{u}_h) \\ &= {}^\tau g + \mathbf{N^T Q^e u_c^e} \end{aligned} \tag{5.13}$$

Substituting (5.12) and (5.13) into (5.11), the nodal force vector corresponding to the contact nodes is derived as [21]

$${}^t\mathbf{R_c^e} = -\alpha\ {}^\tau g\ \mathbf{Q^e}^T\mathbf{N} - \left(\alpha \mathbf{Q^e}^T\mathbf{N}\mathbf{N}^T\mathbf{Q^e}\right){}^t\mathbf{u_c^e} \tag{5.14}$$

Stating ${}^\tau\mathbf{R_c^e}$ and $\mathbf{K}_\alpha^\mathbf{e}$ as in Eq.(5.15) and (5.16), ${}^t\mathbf{R_c^e}$ is expressed via (5.17). Equation (5.17) can be substituted into (5.3) to solve for the displacement vector.

$${}^\tau\mathbf{R_c^e} = \alpha\ {}^\tau g\ \mathbf{Q^e}^T\mathbf{N} \tag{5.15}$$

$$\mathbf{K}_\alpha^\mathbf{e} = \alpha\mathbf{Q^e}^T\mathbf{N}\mathbf{N}^T\mathbf{Q^e} \tag{5.16}$$

$${}^t\mathbf{R_c^e} = -{}^\tau\mathbf{R_c^e} - \mathbf{K}_\alpha^\mathbf{e}\,{}^t\mathbf{u_c^e} \tag{5.17}$$

5.3 Mesh Refinement

There is a direct correlation between the accuracy of the finite element method and the corresponding mesh size. In order to achieve smaller errors, finer spatial discretization is required. A uniformly refined mesh may result in unnecessary high computation times. In non-uniform meshes, the mesh granularity is increased only where necessary and not over the entire mesh. This would help improve modelling accuracy with a smaller increase in the computation cost than that achievable with a uniform mesh. There are a number of different ways to obtain a non-uniform mesh [131]:

i) **h-refinement** where the size and number of elements are subject to change.

ii) **p-refinement** where higher order interpolation functions are employed.

iii) **r-refinement** where the mesh nodes are relocated to adjust the mesh density. The number of elements and the order of interpolation functions remain unchanged in this approach.

In general, areas prone to large stress or strain gradients demand a high density mesh for accurate finite element modeling. In *adaptive mesh refinement*, a tentative solution is obtained through an initial mesh, and then using some form of error estimation [137,138] candidate elements are selected for remeshing. After this step, one or a combination of the remeshing techniques mentioned above can be applied to improve the modeling accuracy. However, in case of real-time deformable body simulation, this iterative solution of the problem followed by a posteriori error estimation is not practical. Noting the fact that surface areas in physical contact are subject to large stress/strain gradients, choosing contact elements for refinement

presents a simple and intuitive approach to achieve better accuracy in deformation modeling.

In r-refinement, only positional data of the mesh is subject to changes, therefore updating mesh data on GPU is simpler. However this approach can result in low-quality elements, in particular at areas with fixed boundary and its accuracy is limited [139]. Additionally, extra care should be taken to avoid nodes crossing over the elements boundaries in the mesh [140].

Using p-refinement requires employing higher-order formulation for interpolating functions. This approach can result in an improved conditioning of the system of equations and better convergence rates compared to h-refinement [141]. But in p-refinement, the computing kernels of the FEM matrices should be modified to incorporate hierarchical shape functions with higher orders.

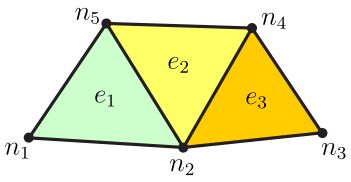
In h-refinement, new elements are added to improve the spatial discretization. This method is more commonly used compared to other adaptive schemes [142]. Using h-refinement allows for direct integration of the local refinement to the FEM solution without introducing any changes to the GPU kernels for computing FEM matrices. For this reason, h-refinement is selected for local mesh refinement in this thesis. However, other adaptive schemes remain as possible candidates for future work.

It should be noted that mesh refinement has been extensively studied in the literature and its treatment is beyond the scope of this thesis. Rather, the goal here is to find a simple way of integrating some form of local mesh refinement in our proposed solution for fast/real-time deformable body simulation on GPUs. In what follows, first the mesh data structure and challenges imposed by mesh

refinement are discussed. Then a solution for GPU implementation of local mesh refinement at contact elements is proposed.

5.3.1 A Note on Data Storage Scheme

According to the definition provided in Section 4.1, FEM mesh consists of nodes and elements. The initial mesh data is stored in elemental matrix **t** and nodal position matrix **p**. Each row of matrix **t** is associated with one element in the FEM model, and consists of the node numbers of that element. Each row of **p** corresponds to one node in the mesh and contains the Cartesian coordinates of that node. Based on **t** and **p**, a nodal-elemental matrix **pt** is formed such that the nodal coordinates of each element are accessed in consecutive memory locations. This data is used to compute FEM matrices per element. These matrices are stored in row-major format in one-dimensional arrays in GPU global memory. Figure 5.4 gives an example of a 2D triangular mesh with three elements and five nodes.



$$\mathbf{t} = \begin{bmatrix} n_1 & n_2 & n_5 \\ n_2 & n_4 & n_5 \\ n_2 & n_3 & n_4 \end{bmatrix}, \mathbf{p} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \\ x_5 & y_5 \end{bmatrix}, \mathbf{pt} = \begin{bmatrix} x_1 & y_1 & x_2 & y_2 & x_5 & y_5 \\ x_2 & y_2 & x_4 & y_4 & x_5 & y_5 \\ x_2 & y_2 & x_3 & y_3 & x_4 & y_4 \end{bmatrix}$$

Figure 5.4: Elemental and nodal mesh matrices for a sample 2D triangular mesh.

From this example, it is clear that the information in matrix **pt** is already contained in the elemental and nodal matrices; therefore this matrix may seem redundant. However, a performance issue justifies using the matrix **pt** instead of **p** and **t**. The global memory in GPU is optimized for access to contiguous locations. This means the running threads on GPU can read from or write onto contiguous addresses

much faster than they would with irregular access patterns. This topic is further explored in Section 7.3. When reading the node coordinates of an element using **t** and **p**, memory access is indirect, i.e. first the node indices are read from **t** and then the corresponding node coordinates are read from **p**. Additionally, access to **p** entries is not contiguous. Therefore, it is much faster to access the node coordinates using the **pt** matrix than through the **p** and **t** matrices.

The above mentioned nodal-elemental data structure provides a simple and compact representation of 2D and 3D finite element meshes. However, its efficient application is limited to cases where the mesh is not subject to dynamic changes. For surface meshes in 2D models, other data structures such as Winged Edge [143] offer more flexibility for dynamic changes such as splitting or merging which could be useful in applications such as cutting simulation. Winged Edge data structure explicitly represents the nodes, elements (faces) and edges of a mesh, imposing larger memory requirement compared to the nodal-elemental data structure.

Mesh refinement can introduce several challenges of its own. As an example, consider a case in which element e_3 in Fig. 5.4 is selected for refinement. In such case,

(a) Array **pt** (or alternatively arrays **t** and **p**) should be updated in an efficient way. This requires removing the corresponding data of the parent element e_3 and adding the data of the new elements to the array.

(b) Refinement of contact elements results in hanging nodes on the adjacent elements. To rectify this problem, the refinement should be propagated. In the example presented in Fig. 5.5, this can be accomplished by bisecting e_2.

(c) Parallel implementation of mesh refinement requires multiple threads to be able to update **pt** array by adding the new values to the end of the array. But since the end of the array is dynamically changing, a method should be devised to avoid simultaneous access to a same memory location.

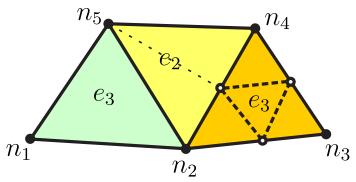


Figure 5.5: Removing hanging node in an adjacent element.

In the following, a solution is proposed to address all the above noted problems. The solution is implemented for a 2D triangular mesh, however it can be extended for 3D tetrahedral mesh. In this approach, initially contact elements and their adjacent elements with hanging nodes are marked for refinement. Next the mesh data structure is updated accordingly.

5.3.2 Marking Elements for Refinement

Consider an element e_1 with nodes n_1, n_2 and n_3 and edges d_1, d_2 and d_3. There are many ways to refine this element. One way would be to bisect one or multiple edges of e_1. Figure 5.6 shows different possible subdivisions of a triangular element.

As mentioned earlier, local mesh refinement is performed on contact elements that are provided by a collision detection module. The contact elements are regularly refined into four smaller isotropic triangles. The refinement is propagated to

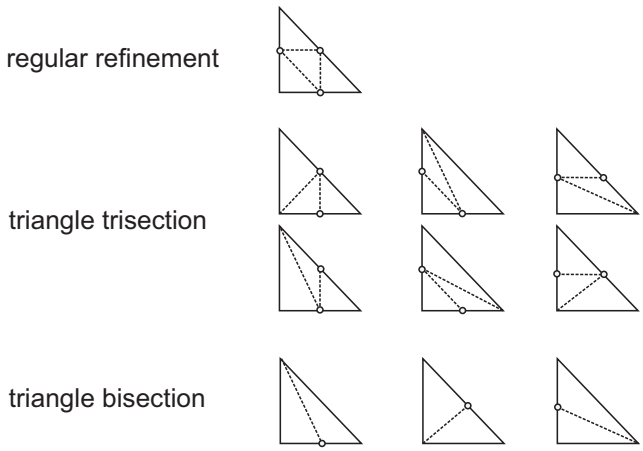


Figure 5.6: Different scenarios for triangle refinement.

the neighbor elements and depending on the number of hanging nodes, the proper partitioning is selected. Figure 5.7 demonstrates an example for this type of mesh refinement. The two shadowed elements are in contact and regularly refined.

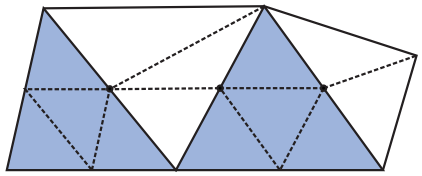


Figure 5.7: Refinement of contact and neighbor elements.

In order to detect which elements require a certain type of partitioning, mesh refinement can be performed in two stages:

- in stage one, the elements are marked for refinement,
- in stage two, the **pt** array is updated.

As noted previously, each row of the elemental matrix **t** corresponds to one element, consisting of the node numbers of that element. Since any d_j edge in a triangle element corresponds to a node n_j, $j = 1,2,3$, a simple yet effective approach

to mark edge j of element i for bisection is changing the sign of t_{ij} in matrix **t**. To mark edges of the neighbor elements, an edge-element adjacency list is required, which can be computed off-line. Therefore, the elements are marked by changing the sign of those rows in matrix **t** corresponding to the contact elements and then altering the sign of the edges of the adjacent elements. Alternatively, a vector with length equal to the number of mesh elements can be employed for marking.

It is noted that this marking algorithm is parallel and can be efficiently implemented on GPUs. Fig. 5.8 presents an example of the marking scheme for a mesh with two contact elements. Each GPU thread is assigned to mark one contact element and the corresponding edges of the adjacent elements. Following this scheme, each edge is accessed by only one thread, therefore threads can perform the marking step in parallel without race condition.

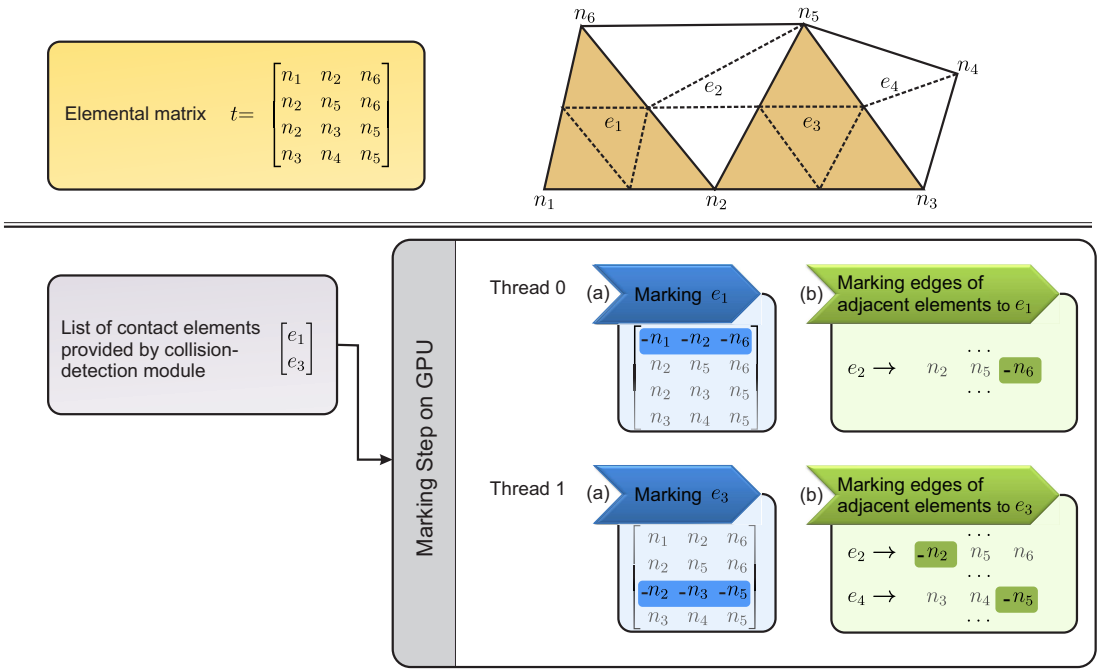


Figure 5.8: Marking edges of contact and adjacent mesh elements for refinement.

5.3.3 Updating Elemental Matrix

As explained in Section 5.3.1, a main challenge in parallel updating of the refined mesh data is to avoid memory contention due to concurrent access of multiple threads for writing the new mesh data on a same memory location. To explain this problem, consider a mesh with initial number of m elements which after detecting the contact elements and marking for refinement, l_1, l_2 and l_3 elements are selected to be bisected, trisected and regularly refined respectively.

In the simplest scenario for updating the mesh matrices due to local mesh refinement, only l_1 elements are marked for refinement to be bisected and $l_2 = l_3 = 0$. In this case, two new triangles are derived from each marked parent element. One replaces the parent element and the other is added to the end of the mesh data array. Assigning one thread per marked element, it is straightforward to allocate a specific address for contiguous storage without memory contention. See Fig. 5.9 for an example where thread i updates memory location $m+i$ and the corresponding location of the parent element.

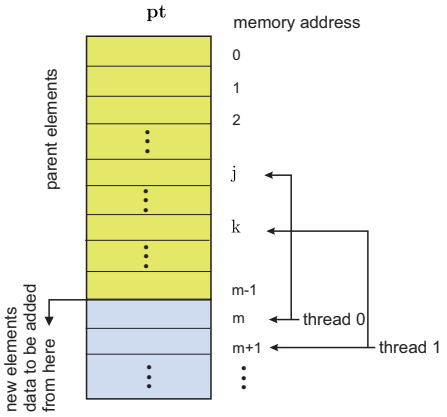


Figure 5.9: Updating nodal-elemental matrix **pt** for bisected elements to include mesh data of newly added elements.

In case that l_2 and l_3 are non-zero, some threads correspond to elements that are refined into three or four smaller triangles. Then the address increment at the end of the array for storing new data is variant and not known to the parallel GPU threads. To avoid this problem, the nodal elemental matrix **pt** can be updated in three steps such that in each step, each thread adds only one new element to the end of the mesh data array. Fig. 5.10 demonstrates this approach.

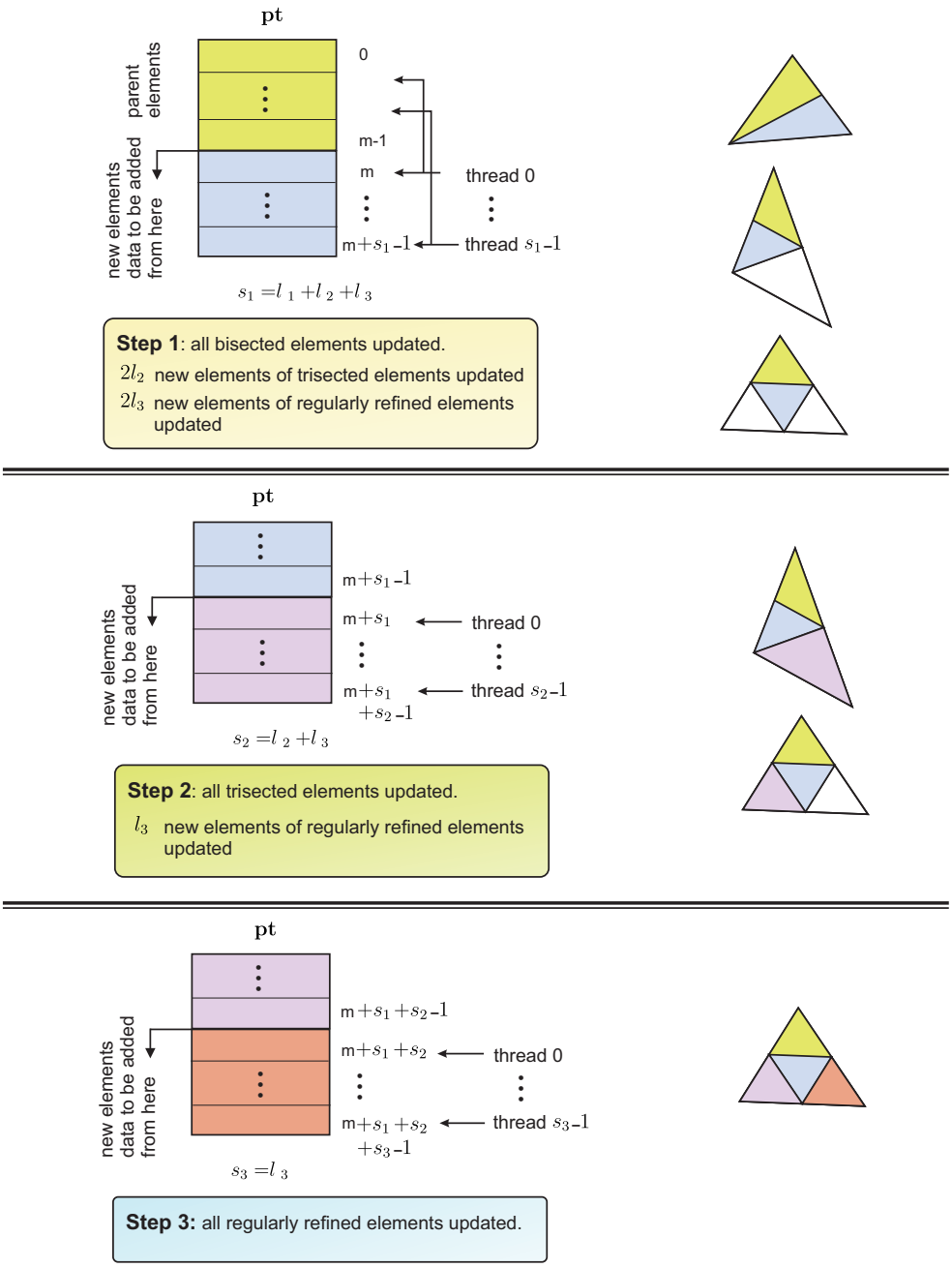


Figure 5.10: Updating nodal-elemental matrix **pt** for the general case to include mesh data of newly added elements.

CHAPTER 6

GPU PARALLEL COMPUTING FOR SOLVING A LINEAR SYSTEM OF EQUATIONS

The elemental equations for static analysis (4.1) and dynamic analysis after implicit numerical integration are both in the general form of

$$\mathbf{A}_i^e \mathbf{x}_i^e = \mathbf{b}_i^e \tag{6.1}$$

where the superscript e denotes elemental quantities and subscript i is the element number in the finite element mesh. These elemental equations are usually assembled into a global matrix equation in the form of $\mathbf{Ax} = \mathbf{b}$ by taking into consideration interactions among the elements as well as boundary conditions [131]. Note that

the elemental equations can not be independently solved for $\mathbf{x_i^e}$ since $\mathbf{A_i^e}$'s are rank deficient and in practice only the vector $\mathbf{b}$ is known but not individual $\mathbf{b_i^e}$'s. The external boundary constraints are defined over the complete model rather than each $\mathbf{A_i^e}$. Only after application of the boundary constraints [131, 144], the global matrix $\mathbf{A}$ becomes full-rank.

In the general case where the matrix $\mathbf{A}$ is nonlinearly dependent on displacement vector $\mathbf{x}$, an incremental approach can be adopted in which loads and displacements are applied in multiple increments. If the desired accuracy is not achieved, the well-known *Newton-Raphson* successive approximation method can be employed per increment to enforce equilibrium [40, 145]. In the real-time experiments for dynamic analysis of deformation performed in this thesis, the time increments are small and Newton-Raphson is not used. Several direct and iterative methods exist for solving large linear systems of equations [110, 111, 118]. In the following sections, some of these methods are briefly discussed.

6.1 Direct Solvers

The basic idea of direct solvers is to decompose matrix $\mathbf{A}$ into simpler forms in order to facilitate solving the system of equations. In the popular *LU decomposition*, $\mathbf{A} = \mathbf{LU}$, where $\mathbf{L}$ is a lower triangular and $\mathbf{U}$ is an upper triangular matrix. Despite the sparse pattern of matrix $\mathbf{A}$, $\mathbf{L}$ and $\mathbf{U}$ matrices may not be sparse. This problem is more pronounced while dealing with large sparse matrices, both in terms of memory footprint and computing cost. The nonzero elements that are introduced in $\mathbf{L}$ and $\mathbf{U}$ factors with no counterpart in matrix $\mathbf{A}$ are known as *fill-ins*. Several algorithms have been developed, where by changing the order of the matrix rows,

the number of fill-ins is reduced [110]. In general, sparse direct solvers consist of following steps [146]:

(*a*) reordering phase to minimizes the number of fill-ins,

(*b*) an analysis phase, also known as symbolic factorization, to determine the non-zero structure of **L** and **U** factors. This step is used to determine the data structure and memory allocation.

(*c*) numeric factorization phase, and

(*d*) triangular solver phase where forward elimination is followed by backward substitution.

The numeric factorization is usually the most computationally expensive step. The first three steps need to be done once for a certain matrix **A** and then can be used for different vectors **b**. Step four has to be repeated for different vectors **b**. The triangular solver for dense structures is inherently a sequential algorithm. However for sparse structures, depending on the structure of the matrix, it is possible to exploit some degrees of parallelism. One method for parallel GPU implementation of sparse triangular solvers is explored in [147].

In the case of a symmetric positive definite matrix, **A** can be decomposed as $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$, where **L** is a lower triangular matrix and $\mathbf{L}^{\mathbf{T}}$ is its transpose. This decomposition is known as *Cholesky decomposition* and is numerically more stable and faster than LU-decomposition [148].

6.2 Iterative Solvers

In general, iterative methods are more amendable to massive parallelization of the computations. They are also typically more efficient in terms of memory usage compared with direct solvers [149]. The basic idea in iterative methods is to produce a sequence of improving approximations to the solution of $\mathbf{Ax} = \mathbf{b}$. There are two main classes of iterative solvers for linear system of equations: *stationary methods* and *Krylov subspace*-based methods.

In stationary methods, or classical iterative methods, matrix $\mathbf{A}$ is split into non-singular matrices $\mathbf{M}$ and $\mathbf{N}$, such that $\mathbf{A} = \mathbf{M} - \mathbf{N}$. Then the solution can be expressed iteratively as $\mathbf{x^{(k+1)}} = \mathbf{M^{-1}}(\mathbf{Nx^{(k)}} + \mathbf{b})$. The key point for effectiveness of such methods is to choose a matrix $\mathbf{M}$ that can be easily inverted and yields fast convergence. The main algorithms of this class are Jacobi, Gauss-Seidel, Successive Overrelaxation (SOR), and Symmetric Successive Overrelaxation (SSOR) [111].

In Krylov subspace methods, a sequence of Krylov subspace $\mathscr{K}_n$ is created to find the approximate solutions $\mathbf{x^{(n)}}$ in that subspace.

$$\mathscr{K}_n = \{\mathbf{b}, \mathbf{Ab}, \mathbf{A^2b}..., \mathbf{A^{n-1}b}\} \tag{6.2}$$

Some known algorithms of this class include the Conjugate Gradients (CG), Biconjugate Gradients Stabilized (BiCGStab) and Generalized Minimum Residual Method (GMRES). For symmetric positive definite matrices, one of the most popular Krylov subspace algorithms is the CG [24]. The finite element matrices derived in Chapter 4 are symmetric positive definite, hence can be solved using the CG method. From a numerical perspective, the CG method is robust, has relatively fast convergence rate, and requires less computations compared to other iterative

methods such as BiCGStab or GMRES, which are applicable to a broader range of matrices compared to CG [118, 150].

6.3 Preconditioning Methods

The number of iterations required to converge to the solution in $\mathbf{Ax} = \mathbf{b}$ depends on the initial guess for $\mathbf{u}$, desired error tolerance, and the condition number of matrix $\mathbf{A}$. The condition number of the stiffness matrix in finite element analysis depends on the meshing quality and increases by the problem size, slowing down the convergence. To counter this problem, a preconditioning step may be used [111]. Preconditioning usually replaces $\mathbf{Ax} = \mathbf{b}$ with an equivalent system of equations $\mathbf{P^{-1}Ax} = \mathbf{P^{-1}b}$, where $\mathbf{P^{-1}A}$ has a better condition number compared to $\mathbf{A}$, therefore requires fewer number of iterations to converge. The computational overhead imposed by $\mathbf{P^{-1}}$ should not cancel out the saving achieved by reduced number of iterations.

A symmetric matrix $\mathbf{A}$ can be decomposed as $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{L^T}$, where $\mathbf{L}$ is the lower triangular part of $\mathbf{A}$ and $\mathbf{D}$ is the main diagonal of $\mathbf{A}$. Using this decomposition, some possible choices for the preconditioning matrix $\mathbf{P}$ include [111],

$$\mathbf{P} = \mathbf{D} \qquad \text{Jacobi preconditioning} \tag{6.3}$$

$$\mathbf{P} = \mathbf{L} + \mathbf{D} \qquad \text{Gauss-Seidel preconditioning} \tag{6.4}$$

$$\mathbf{P} = \frac{\mathbf{1}}{\omega}(\mathbf{L} + \omega\mathbf{D}) \qquad \text{SOR preconditioning} \tag{6.5}$$

where in (6.5) $\omega > 1$ is a scalar. Another common preconditioning technique is based on incomplete factorization, where approximate factors are employed. The approximate factors are in some sense close to the original factors. Incomplete

LU factorization, or for symmetric positive definite matrices, incomplete Cholesky factorization are examples of this technique [151]. Due to a high level of parallelism achieved by Jacobi preconditioning compared to other techniques, this method is adopted in two different GPU implementations of CG algorithm on GPU.

6.4 Conventional and Element-by-Element PCG

This section elaborates on two different approaches to implement CG algorithm. The pseudo-code, advantages and drawbacks of each approach with respect to GPU implementation are explained. Further details of GPU kernels to perform CG will be given in Chapter 7.

Fig. 6.1 presents a pseudo-code of the Preconditioned Conjugate Gradients (PCG) method. Each iteration of the algorithm consists of one sparse matrix by vector multiplication (line 9), inverse of preconditioner matrix by vector multiplication (line 13), a couple of vector by vector multiplications (lines 10 and 14) and some scalar multiplications and vector additions. In case of the Jacobi preconditioning, $\mathbf{P^{-1}}$ is simply the inverse of diagonal entries of the sparse matrix $\mathbf{A}$ and solving for $\mathbf{z}$ in line 13 turns into component-wise vector product. This allows for efficient parallel implementation on GPU. Using other types of preconditioning techniques mentioned in Section 6.3 requires two sets of triangular solving procedures, which would be less amendable to parallelization.

The matrix assemblage in Line $2a$ of the algorithm is a memory-bound function, i.e. the limiting factor for speed is the memory access time rather than computation time of arithmetic operations. Matrix assemblage is the most time-consuming step in a GPU implementation of the PCG algorithm [105] and is performed only once

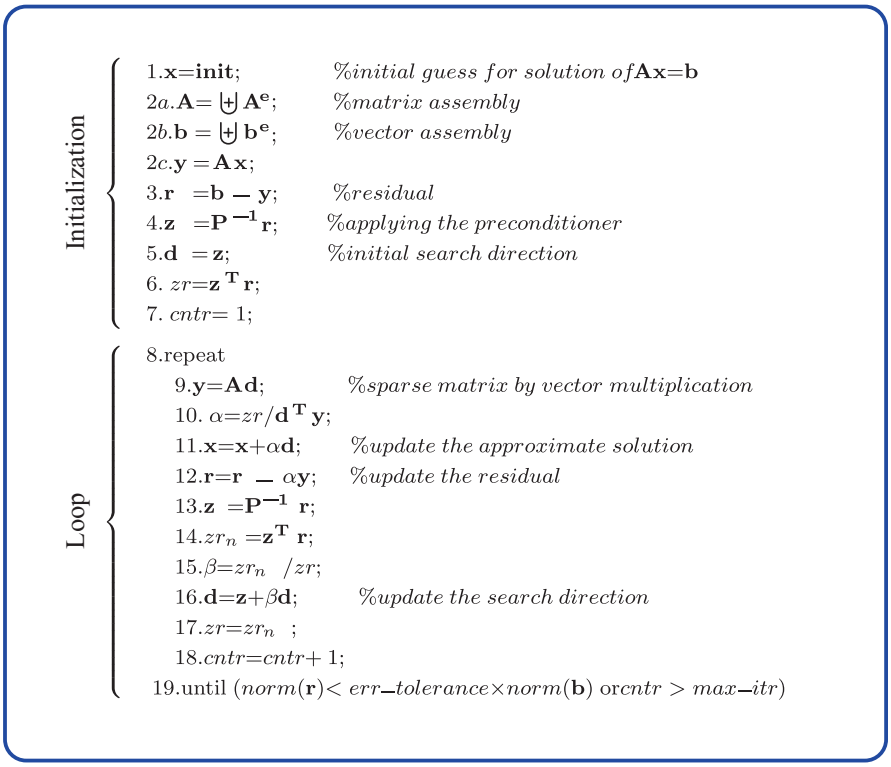


Figure 6.1: Pseudo-code for the conventional PCG method, $\mathbf{x} = PCG(\mathbf{A}, \mathbf{b})$.

at the initialization phase. In the Jacobi PCG loop, the matrix by vector multiplication is the most computationally intensive step. When the elemental matrices are assembled into a sparse matrix $\mathbf{A}$ (see Eq.6.6a), one could exploit existing efficient algorithms in sparse matrix operations, i.e. see [152], for this multiplication.

$$\mathbf{y} = (\sum_{\mathbf{i}} \hat{\mathbf{A}}_{\mathbf{i}}^{\mathbf{e}})\mathbf{x} = \mathbf{A}\mathbf{x} \tag{6.6a}$$

$$= \sum_{\mathbf{i}} \hat{\mathbf{A}}_{\mathbf{i}}^{\mathbf{e}}\mathbf{x} = \sum_{\mathbf{i}} \hat{\mathbf{y}}_{\mathbf{i}}^{\mathbf{e}} \tag{6.6b}$$

Alternatively in the so-called *element-by-element* (EbE) or *matrix-free* Conjugate Gradients Method, the elemental matrices can be multiplied by the corresponding components from the vector and then assembled into the resultant vector (see

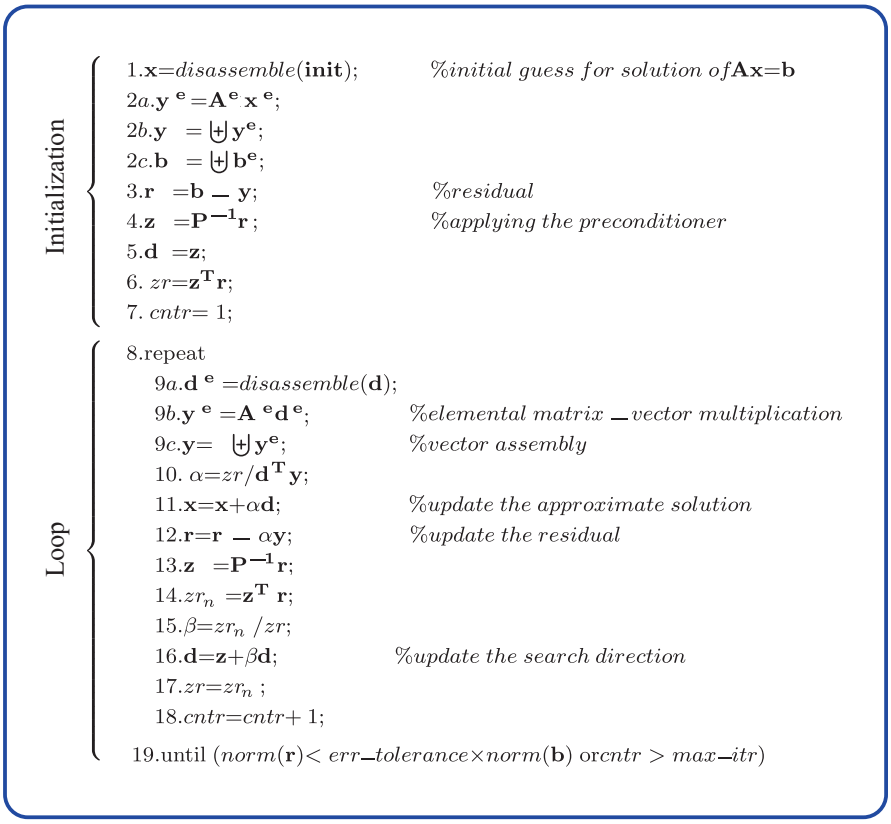


Figure 6.2: Pseudo-code for element-by-element PCG method, $\mathbf{x} = EbEPCG(\mathbf{A^e}, \mathbf{b})$.

Eq.6.6b). This seemingly small difference in the order of summation and multiplication allows one to forgo the matrix assemblage step and hence avoid issues pertaining to indirect memory access for storage and operation on a sparse matrix. This element-based approach can be efficiently implemented on a GPU compute platform.

The PCG method in Fig. 6.1 can be modified so it can operate on elemental $\mathbf{A_i^e}$ matrices instead of the sparse matrix $\mathbf{A}$. Fig. 6.2 presents the pseudo-code for matrix-free PCG algorithm. The main changes with respect to the original PCG

method are as follows. First vector $\mathbf{d}$ should be disassembled into elemental vectors $\mathbf{d_i^e}$ consisting of the corresponding elements of $\mathbf{d}$ that are multiplied by $\mathbf{A_i^e}$. Then the sparse matrix by vector multiplication should be replaced by an elemental matrix by vector multiplication for all the elements, i.e. $\mathbf{y_i} = \mathbf{A_i^e d_i^e}$, followed by a kernel for vector assemblage to obtain $\mathbf{y}$ from $\mathbf{y_i}$'s. Fig. 6.3 shows a schematic of these three kernels.

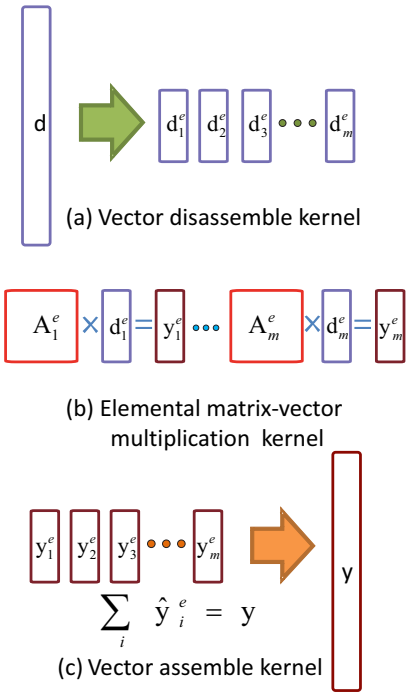


Figure 6.3: Schematic of kernels used to replace sparse matrix-vector multiplication in PCG algorithm with elemental matrix-vector multiplications.

Pros:

An element-based computing approach avoids matrix assemblage and needs not to store indices of non-zero entries. Direct access to elemental matrices components is faster compared to indirect accessing to global matrix elements. Additionally EbE PCG scheme would be suitable for the use with adaptive mesh refinement

algorithms since only the refined elements would need extra computation [39]. An element-based approach also facilitates application of deformable contact constraints. As will be explained in Chapter 7, an implementation of vector assembly in element-by-element PCG algorithm can be presented that results in more consistent performance for a vast range of different FEM meshes compared to conventional PCG algorithm.

Cons:

More computations need to be performed at each iteration of the EbE PCG. Sparse matrix by vector multiplication (Step 8 in Fig. 6.1) is replaced by three steps of vector disassembling, elemental matrix-vector multiplications and vector assembly in EbE PCG (8*a*, *b* and *c* in Fig. 6.2). Vector assemblage process, as will be discussed in Section 7.5, takes less effort compared to matrix assemblage; however this task needs to run at each iteration of the EbE PCG method. The proposed vector assembly is based on atomic operations which using CUDA 5.0 and current GPU compute capability, 3.5, is limited to single precision. Both methods are discussed and compared in Chapter 8.

Several publications in the literature address the mathematics of element-by-element CG algorithm [39], and deriving a pre-conditioner for this method [108] with the goal of reducing memory usage. The contribution of this thesis in the EbE PCG is the development of a fast and scalable GPU-based implementation for the algorithm. The element-based structure of this algorithm enables efficient memory loads and massive parallelization of its computations on many-core architectures.

CHAPTER 7

GPU-based Compute Platform for Deformation Analysis

In recent years many-core computing platforms, including GPUs and FPGAs, have been employed as specialized coprocessors in heterogeneous computing along CPUs. In general, FPGAs can be customized at a low level for the problem at hand to yield very high performance in computing acceleration. However designing FPGA-based computing architectures is a complicated and time consuming process, which is highly customized to the FPGA hardware and the computational problem. Small changes in the computing problem may require substantial changes in the compute architecture and moving designs across different platforms may require significant design revisions. FPGAs also cost significantly higher than GPUs.

GPUs were initially developed for special purpose graphics rendering. The architecture of modern GPUs has evolved to point that they have become not only powerful graphics engines, but also highly parallel programmable devices for other high performance applications. There is a growing trend in using parallel computing on large number of computational cores to improve performance rather than increasing the single-thread performance [153]. The increasing power of GPUs and their use in high performance computing applications has spurred an active area of research in GPU-based computing. GPUs offer large memory bandwidth and high throughput computing that outpaces state-of-the-art CPUs. They are particularly suitable for the type of computational problems discussed in this chapter.

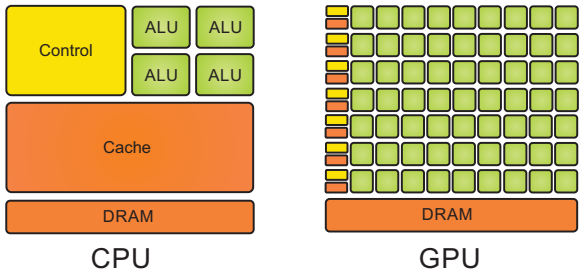


Figure 7.1: CPU and GPU architectures [154].

As shown in Fig. 7.1, in a modern CPU architecture, a major portion of the transistors are dedicated to control units and memory chache. In contrast in a modern GPU architecture, the majority of transistors are allocated to the computing cores. This explains the fundamental difference between the different types of problems that each one is specialized for. In general, CPUs are optimized for low-latency access to cached data sets and performing sequential and control-intensive tasks. Although with the emergence of multi-core architectures, CPUs can handle a limited multi-threading parallelization as well. On the other hand, GPUs are best

suited for high throughput data-parallel applications where a sequence of computations are performed on a large set of data in Single Instruction Multiple Data (SIMD) fashion. Therefore, GPUs can be considered as co-processors that can offload parallel compute-intensive portions of the application, while the remainder of the code is executed on CPU.

The Compute Unified Device Architecture (*CUDA*) programming model introduced by NVIDIA is designed to support joint CPU/GPU execution of an application [155]. Different architectural designs and hardware features in CUDA GPUs are represented by a number known as *compute capability* [154]. CUDA provides a multi-threaded SIMD architecture to program NVIDIA GPUs. The set of data to be processed is referred to as *stream* in computer programming context and GPU instructions on streams are referred to as *kernel*. Kernels are executed by a number of threads. Threads are grouped in a *grid* of thread *blocks*. Fig. 7.2 presents this structure. The grid and block sizes are set at the kernel launch time. Threads within each block run on the same *streaming multiprocessor* (SM) and can be synchronized with each other. SM executes threads in groups of 32, called *warp*.

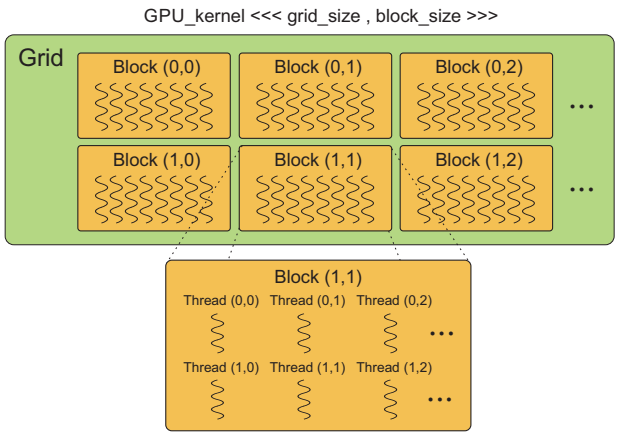


Figure 7.2: Grid and blocks structure in CUDA [154].

Threads can access different CUDA device memory types. *Shared memory* and *registers* are fast, but limited in size. *Global memory* on the device is large but relatively slow. However, while some threads are waiting for memory access time, other threads can become active and perform arithmetic operations to hide the latency. Fig. 7.3 displays this memory hierarchy. A comprehensive introduction on CUDA API principles and practices can be found in [31,91,156].

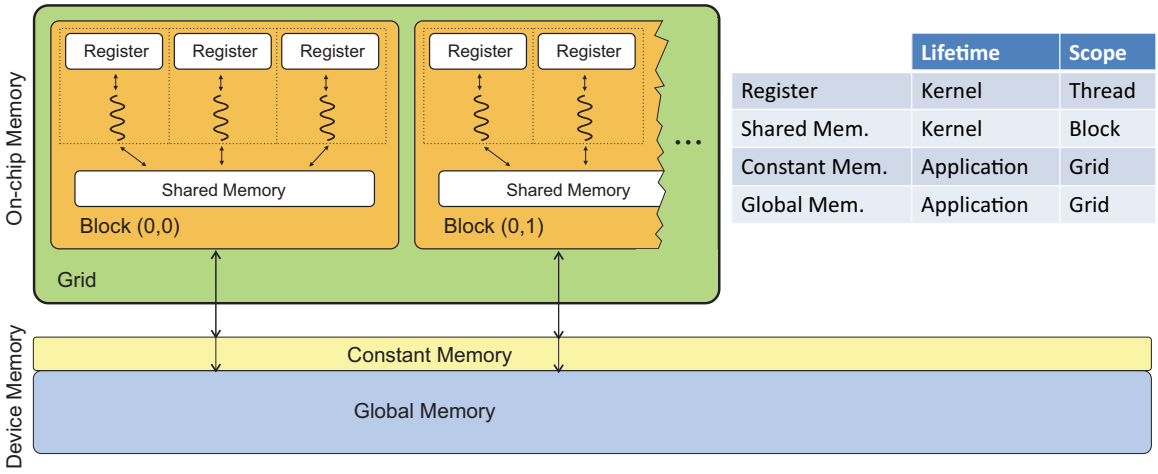


Figure 7.3: Structure, lifetime and scope of global/constant/shared memory and registers.

For numerical simulation of a deformable object, initially the mesh data discussed in Section 5.3.1, is computed off-line and transferred to GPU through two matrices, namely matrix **p**, consisting of position coordinates of each node in the mesh and matrix **t**, comprising of tetrahedral (triangular for 2D models) indices associated with each element. For efficient memory access, as explained in 5.3.1, these data are unified into **pt** array. The GPU kernels that will be discussed shortly,

address the two main problems described in Chapters 4 to 6, i.e. real-time updating of the non-linear FEM matrices and solving of a large system of linear equations. Fig. 7.4 gives an overview of the proposed GPU-based computing scheme for soft-tissue deformation analysis.

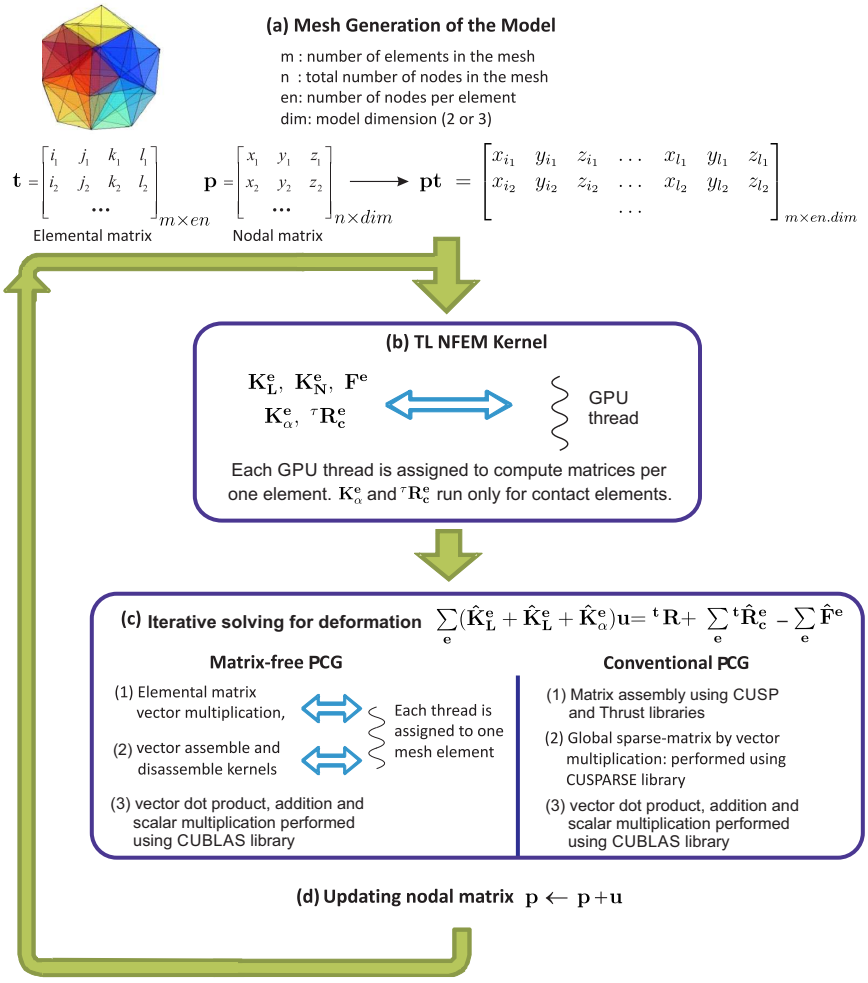


Figure 7.4: The general computation scheme: (a) mesh generation is done off-line, mesh data is sent to GPU via elemental and nodal matrices, (b) in total Lagrangian nonlinear FEM kernel, computations per element run in parallel (c) matrix linear algebra operations in PCG are performed in parallel (d) mesh nodal matrix, containing the coordinates of all vertices is updated and the procedure is repeated for the next time step.

7.1 GPU Kernels for FEM Matrix Construction

The matrices $\mathbf{K}_{\mathbf{L}}^{\mathbf{e}}$, $\mathbf{K}_{\mathbf{N}}^{\mathbf{e}}$ and $\mathbf{F}^{\mathbf{e}}$ in the nonlinear finite-element formulation, as well as $\mathbf{K}_{\alpha}^{\mathbf{e}}$ and ${}^{\tau}\mathbf{R}_{\mathbf{c}}^{\mathbf{e}}$ in presence of contact, were presented in Chapters 4 and 5. Computing these matrices involves some basic linear operations performed on small-sized matrices/vectors per element in the FEM model, see 4.15-4.17 and 5.15-5.17. The sizes of these elemental matrices/vectors are determined by the degrees of freedom of the corresponding finite element in the discrete model, e.g. see Table 4.1 for some examples.

Strain-displacement matrices $\mathbf{B}_{\mathbf{L}}^{\mathbf{e}}$ and $\mathbf{B}_{\mathbf{N}}^{\mathbf{e}}$ are computed based on spatial derivatives of the interpolation functions. In the total Lagrangian formulation, these derivatives are with respect to the initial configuration and remain unchanged for all time steps. The partial derivatives can be calculated once and then stored in the device global memory for the use in the update of FE matrices. Threads can be assigned on GPU to compute the elemental matrices due to data independency among the elements. GPUs can substantially accelerate these types of computations due to the data-parallel structure and arithmetic-intensive nature of this problem, particularly for models with large number of elements.

As seen in the general scheme in Fig. 7.4, FEM matrices can be computed based on total Lagrangian formulation using the mesh data, i.e. elemental and nodal matrices stored in **pt** array. The kernels for these computations assign one thread per element. These kernels are fully scalable and can execute for different-sized problems on single or multiple GPUs, and as long as the corresponding portion of mesh data is stored on each device, no extra communication is required among multiple GPUs.

The main compute intensive task in deriving FEM matrices is matrix-matrix multiplication. There exist highly optimized kernels for matrix multiplication of relatively large-sized matrices on GPUs [157,158], available in NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS) library [159]. However, matrix multiplication in computing FEM matrices is different in the sense that it is performed on relatively small-sized matrices, but needs to be repeated for all the elements in the FEM model. It is more challenging to achieve high performance in matrix multiplication for small-sized matrices due to smaller ratio of arithmetic operations to the data operands. An example quoted from [159] helps to clarify this matter:

"a single $m \times m$ large matrix-matrix multiplication performs m^3 operations for m^2 input size, while 1024 $\frac{m}{32} \times \frac{m}{32}$ small matrix-matrix multiplications perform $1024(\frac{m}{32})^3 = \frac{m^3}{32}$ operations for the same input size."

Some CUDA-capable GPU devices allow launching multiple kernels. The user can define any number of streams, but the concurrent execution is limited to sixteen kernels. One approach to compute FEM matrices is to create as many streams as FEM elements and preface each call to CUBLAS matrix multiplication developed for large matrices [159]. However, this approach would not yield the best performance. Another method is to use the newly introduced kernel in CUBLAS version 4.1, developed to perform multiplication on a batch of small-sized matrices, i.e. roughly 128×128 and smaller. This CUBLAS kernel referred to as *batched GEMM API* offers a better performance for matrices which their dimension has an integer factor of 16. As an example, multiplying a batch of 100,000 single-precision matrices $\mathbf{A}_{16\times16}\mathbf{B}_{16\times16}$ on GTX 480 using the batched GEMM API yields 177.1 GFLOPS, while the same batch size of $\mathbf{A}_{12\times12}\mathbf{B}_{12\times12}$ and $\mathbf{A}_{6\times6}\mathbf{B}_{6\times6}$ would

achieve 51.2 and 8.8 GFLOPS respectively.

In this thesis, a fine-tuned kernel for multiplication of matrix sizes defined by the FEM model is developed. As an example, in computing $\mathbf{K_L^e}$ of a tetrahedral FEM model, the matrix multiplication $\mathbf{B_L^T}_{12\times6}\,\mathbf{C}_{6\times6}\,\mathbf{B_L}_{6\times12}$ yields up to 210 GFLOPS on GTX 480. The kernels involved in computing FEM matrices are memory-bounded, hence optimal usage of memory can significantly improve the overall performance. Section 7.2 gives an example of employing low-latency shared-memory and registers for high-performance FEM matrix computations. The FEM matrices for a model with n elements can be stored in n one-dimensional arrays. Large data size of FEM arrays fits to global memory in GPU memory hierarchy. In Section 7.3, a data structure for efficient access to these arrays in global memory is presented.

7.2 Shared Memory and Registers in FEM Computations

Efficient use of on-chip shared memory and registers can help achieve a greater performance with GPUs in memory-bounded kernels. In computing FEM matrices, small-sized matrix by matrix multiplication is the core routine. In matrix multiplication, row and column values should be loaded repeatedly to compute the new result entries. Such repeated memory transactions from the global memory in GPUs are costly due to limited bandwidth and high latency. One strategy to reduce the global memory bandwidth traffic is to use the available on-chip memory. In this way, multiplicand entries can be loaded once onto the on-chip memory to eliminate the need of global memory transactions for the next access times.

The main constraint in using shared memory and registers is the limited size of the on-chip memory. Additionally, in using registers, the maximum number of registers which can be assigned per thread is limited. If a thread exceeds that limit, the excess data is automatically transferred to the global memory local to that thread. The use of global memory in absence of available registers is known as *register spilling*. As seen in Table 7.1, physical constraints vary on GPUs with different compute capabilities.

Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
Max. Registers per Thread	124	124	124	124	63	63	63	255
32-bit Registers per SM	8192	8192	16384	16384	32768	32768	65536	65536
Shared Mem. per SM (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
Max. Threads per SM	768	768	1024	1024	1536	1536	2048	2048
Max. Blocks per SM	8	8	8	8	8	8	16	16

Table 7.1: Physical constraints for different compute capabilities.

In order to achieve the maximum performance on GPU, it is desired to maximize the parallel execution. There is a metric known as *occupancy*, indicating how effectively the GPU resources are kept busy [91]. Occupancy is the ratio of the active threads to the maximum possible threads per streaming multiprocessor. There are three main sources for limiting the occupancy: 1- registers, 2- shared memory and 3- block size, i.e. assigned number of threads per block. For example, considering compute capability 2.1, a maximum number of 1536 threads can reside per SM. However, if each thread uses 63 registers, no more than 520 active threads can reside per SM, limiting the occupancy to 33%. A similar example justifies the limitation enforced by the shared memory. Block dimension is chosen according to the hardware constraints and performance considerations at the kernel launch

time. If the block dimension on a device with compute capability 2.1 for example is chosen 32, since the number of thread blocks is limited to 8 per multiprocessor, the maximum number of active threads equals to $32 \times 8 = 256$ and the occupancy is limited to 17%.

While the occupancy metric provides a tool to determine how efficiently the parallelization is employed, it does not necessarily mean that high performance can not be achieved at lower occupancies. This issue is discussed in detail in [160]. The main argument relies on using *instruction level parallelism* (ILP) in addition to the thread level parallelism. If a number of independent arithmetic tasks are assigned per thread, similar high performance to that in 100% occupancy can be achieved at lower occupancies. Additionally, efficient use of registers and shared memory can offer significant performance gain despite limiting the occupancy. The performance gain, as mentioned previously, is due to larger memory bandwidth and smaller latency.

Matrix multiplications to compute elemental matrices need be performed per element of the FEM model. Each elemental computation can be assigned to a GPU thread and run in parallel. Figure 7.5 presents the sequence of the steps in performing $\mathbf{B_L^T}_{12\times6}\ \mathbf{C}_{6\times6}\ \mathbf{B_L}_{6\times12}$ to compute $\mathbf{K_L^e}$ in Eq.(4.17) in a tetrahedral mesh. These steps are assigned per single thread, allowing instruction level parallelism of the independent arithmetic operations. To perform the matrix multiplication, $\mathbf{B_L}_{6\times12}$ is partitioned into two blocks of $\mathbf{B_1}_{6\times6}$ and $\mathbf{B_2}_{6\times6}$. Larger block sizes limit the occupancy due to limited on-chip memory and can result in register spilling. Smaller block sizes on the other hand, do not reduce much from the global memory bandwidth traffic.

$\mathbf{B_1}$ and $\mathbf{B_2}$ blocks are separately loaded onto on-chip registers, multiplied by the elastic modulus matrix $\mathbf{C}_{6\times6}$ which is stored in the constant memory. Constant memory is not on-chip, but it is cached and optimized to broadcast data among different threads. After the first call, the data in the constant memory can be accessed efficiently from the cache. For the nonlinear material models where the material matrix is not constant but is symmetric, registers can be used to store half of the $\mathbf{C}$ matrix, i.e. 21 more registers in addition to 36 registers used for storing $\mathbf{B_1}$ or $\mathbf{B_2}$. The intermediate results of the matrix products are stored in the shared memory to allow fast access when being reused. Since in general the material matrix $\mathbf{C}$ is symmetric, the computed block $\mathbf{B_2^T C B_1}$ is transpose of $\mathbf{B_1^T C B_2}$, therefore the extra computation in Step 5 in Fig. 7.5 is avoided.

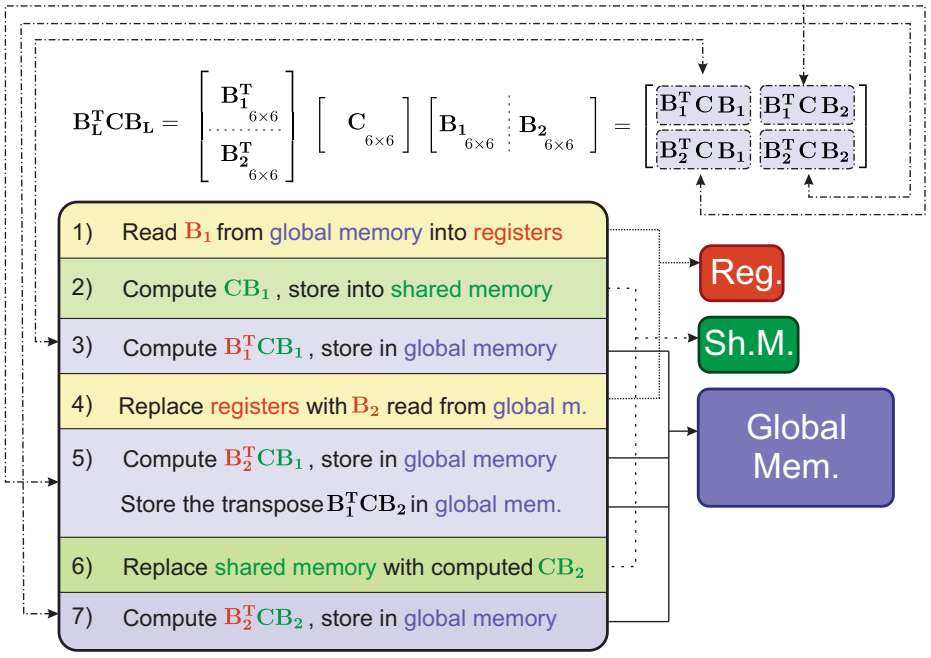


Figure 7.5: Using shared memory and registers to compute $\mathbf{K_L^e}$

7.3 Memory Coalescing

In addition to efficient usage of shared memory and registers, access to the global memory should be coalesced for an optimal performance [31]. Memory coalescing is a technique which allows optimal usage of the global memory bandwidth. That is, when parallel threads running the same instruction access to consecutive locations in the global memory, the most favourable access pattern is achieved.

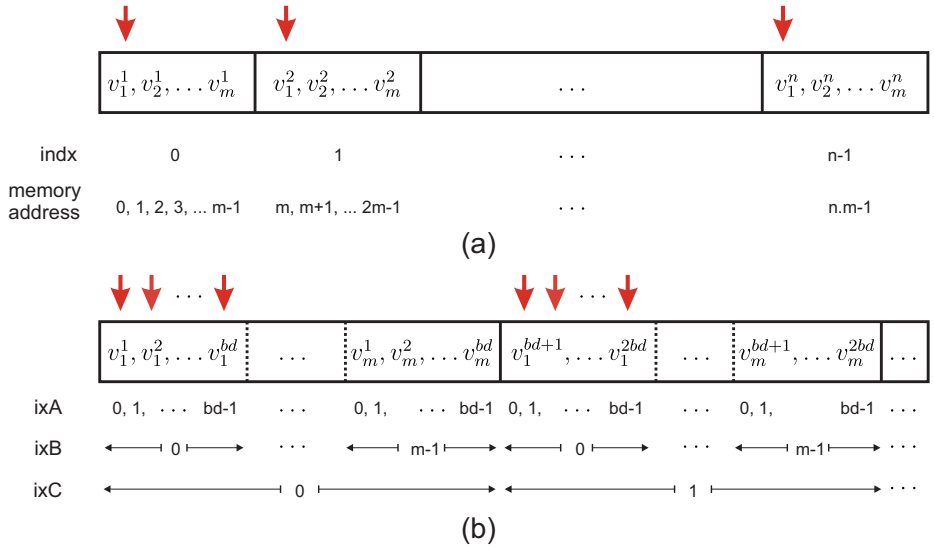


Figure 7.6: Storing n m-element vectors (a) in linear order (b) in coalesced pattern.

The example in Fig. 7.6 helps explain the coalesced arrangement. In Fig. 7.6.a, n vectors of length m are stored in a linear fashion. Element i of vector j is denoted by v_i^j. Each thread in GPU kernel is assigned to one m-length vector. Threads in CUDA are grouped in an array of blocks and every thread in GPU has a unique id which can be defined as $indx = bd \times bx + tx$, where bd represents block dimension, bx denotes the block index and tx is the thread index in each block. Vertical arrows demonstrate the case that parallel threads access to the first components of each

vector, i.e. addresses 0, m, $2m$... of the memory. In this case where memory access is not consecutive, zeroing the gap between these addresses coalesces the memory access. Since the allowed size of residing threads per GPU block is limited, the coalesced data arrangement can be done by storing the first elements of the first bd vectors in consecutive order, followed by first elements of the second bd vectors and so on. The rest of vectors elements are stored in a similar fashion, as shown in Fig. 7.6.b.

In the linear data storage in Fig. 7.6.a, component i $(0 \leq i < m)$ of vector $indx$ $(0 \leq indx < n)$ is addressed by $m \times indx + i$; the same component in the coalesced storage pattern in Fig. 7.6.b is addressed as $(m \times bd)ixC + bd \times ixB + ixA$, where $ixC = floor\left[(m.indx + j)/(m.bd)\right] = bx$, $ixB = j$ and $ixA = mod(indx, bd) = tx$. In summary, for a number of vectors with size m, linear indexing is mapped to coalesced indexing according to (7.1).

$$m.indx + i \quad \longrightarrow \quad m.bd.bx + i.bd + tx \tag{7.1}$$

This data rearrangement can lead to a significant higher memory bandwidth of GPU global memory [91]; for example writing $\mathbf{B_L}$ matrices of length $m = 72$ for 100k elements using coalesced pattern is 11.8 times faster on Geforce GTX 470.

7.4 GPU Kernels for Solving Large System of Linear Equations

In the GPU implementation of the PCG algorithms in Fig. 6.1 and 6.2, in addition to kernels for matrix-vector multiplication, other kernels are required for scalar multiplication and vector addition (SAXPY) and vector dot product. These functions

are implemented using the CUBLAS library by NVIDIA [159]. To avoid unnecessary communication overhead with the host, the return location of the resultant scalar by dot product is set to be on the device. In Jacobi preconditioning, inverse of diagonal entries of the assembled sparse matrix needs to be stored in a vector. In EbE PCG where the assembled matrix is not available, this vector needs to be assembled only for once before the solver loop. Vector assembly on GPU is further discussed in Section 7.6.

In the conventional PCG algorithm, the sparse matrix by vector multiplication can be performed efficiently using the functions provided in CUSPARSE library [161]. In the matrix-free PCG algorithm, three kernels for vector disassembling, elemental matrix-vector multiplication and vector assembly are required to perform Step 8 in Fig. 6.2. The threads for these kernels are defined per element, e.g. a finite element model of 10,000 elements would have as many active threads. The elemental matrices are typically small, for example 12×12 for a tetrahedral mesh from Table 4.1. As a result, the number of arithmetic operations per thread is modest and a high level of parallelism can be attained in large meshes.

In the Jacobi PCG algorithm in Fig. 6.1, the most time consuming steps are matrix assembly (line $2a$) and sparse matrix by vector multiplication (line 8). The latter is expensive due to huge amount of computations in addition to indirect memory accessing to non-zero components of the sparse matrix. The assembly kernel is a bottleneck due to indirect memory access pattern and serialization to avoid memory contention. The problems associated with assembly process and an efficient GPU kernel for vector assembly are further elaborated in Sections 7.5 and 7.6.

7.5 Assembly Process on GPU

According to Eq.(4.20), the assembly process consists of two steps, first mapping elemental matrices/ vectors into global-sized type, then adding them up to form the global matrix/vector. Fig. 7.7 shows an example for mapping elemental matrix $\mathbf{A^e}$ to $\hat{\mathbf{A}}^{\mathbf{e}}$ in the first step. Matrix $\hat{\mathbf{A}}^{\mathbf{e}}$ is mainly comprised of zeros and therefore needs to be stored in one of the standard sparse formats [152]. This requires that the data associated with non-zero indices be stored in addition to their values. The use of this mapping is referred as to *indirect addressing*. In general, accessing data from global memory in GPU is slow and indirect addressing signifies this degradation.

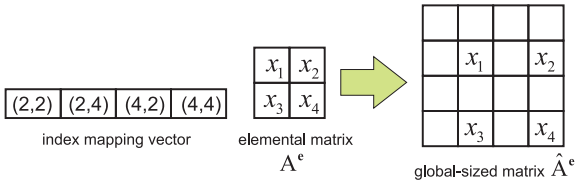


Figure 7.7: Mapping procedure of a sample elemental matrix $\mathbf{A^e}$ into a global-sized matrix $\hat{\mathbf{A}}^{\mathbf{e}}$.

The other challenge associated with assembly process appears in the second step, i.e. adding up the elemental matrices/vectors. There are two general different approaches to perform this addition on GPU. A method known as *scatter* assigns each thread to one elemental matrix/vector. Another method known as *gather* assigns each thread to the computation of one entry of the global matrix/vector. In the scatter method, *race condition* can take place where concurrent access of different threads to a same memory location on the global matrix/vector leads to indefinite result. Fig. 7.8 is an example of such situation. The problem

can be addressed by using *atomic add* provided in CUDA, at the expense of serial execution of threads which cause the race condition. This results in an extra instruction overhead and slower run-time. Double precision operations are not natively supported by atomic add, hence single precision float is used in the scatter method.

Beside using atomic operations, there is also another technique known as graph coloring [162]. The goal is to color the mesh in a way that no two elements with the same color cause a race condition. In GPU implementation, elements of the same color run in parallel, and different colors are executed is sequence. This method requires off-line computation. In this approach double precision can be employed for the scatter method.

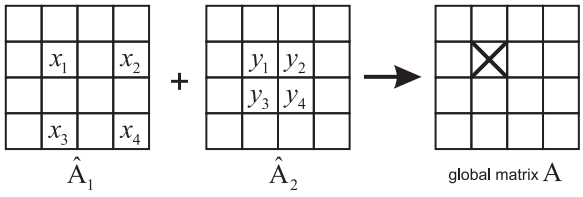


Figure 7.8: Adding up the mapped elemental matrices and storing the results into the global assembled matrix **A**; the location marked with X encounters memory contention.

In the gather method, a mapping array is required to determine which elemental matrices/vectors contribute to each entry of the global assembled form. In general, this approach involves indirect accessing, non-coalesced load transactions from global memory, and can be less efficient for non-structured FEM meshes with many elements connected to each node.

The problems associated with assembly process are common in both matrix and vector assembly, but are more pronounced in matrix assembly since it consists of

more entries. The next section explores an efficient GPU implementation of vector assembly. Assembly of $m \times m$ elemental matrices can be considered as assembly of m elemental vectors of length m; therefore vector assembly kernel, with some changes can be extended for application in matrix assembly.

7.6 Optimized Vector Assembly on GPU

In the scatter method by assigning each GPU thread to one elemental vector $\mathbf{V}_i^{\mathbf{e}}$ in Fig. 7.9.a, memory loads can be coalesced, but storing the results onto the global vector $\mathbf{V}$ is non-coalesced and encounters race condition. In the gather method, each thread is assigned to one element of the global vector $\mathbf{V}$ and memory loads of corresponding elemental vector components from $\mathbf{V}^{\mathbf{e}}$'s are non-coalesced. The gather method was employed in a previous work [163]. In our proposed implementation, a scatter method is employed that by efficient use of shared memory, requires smaller number of non-coalesced global memory transactions.

To address memory contention in the scatter method efficiently, serialization can be performed at two levels. In the first level, a number of elemental vectors are grouped per GPU block, and atomic operations are performed on shared memory which is much faster in comparison to the global memory. This produces locally assembled vectors $\tilde{\mathbf{V}}_i$'s in Fig. 7.9.b. The maximum number of local vectors processed in each thread-block is limited by the allocated shared memory storing vector $\tilde{\mathbf{V}}_i$. In the second level, atomic operations are performed on the global memory to add $\tilde{\mathbf{V}}_i$'s. A mapping array is used to assign $\tilde{\mathbf{V}}_i$ components to the corresponding addresses in the global vector $\mathbf{V}$.

This method by transferring the global memory traffic to the shared memory

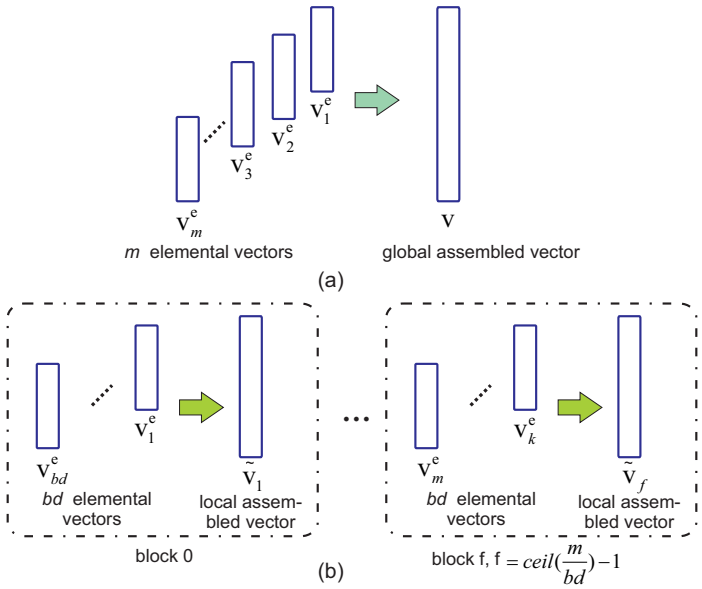


Figure 7.9: Vector assembly.

in its first level, yields about 7× speed-up compared to single-step assembly on the global memory. Fig. 7.10 compares the timing of the two-level vector assembly using shared memory versus the direct atomic add on global memory for a number of different FEM meshes. When there are enough number of threads to fully utilize the GPU device, the relation of the problem size and GPU run-time is expected to be linear. However, small finite element models with smaller sizes do not produce enough threads to keep the GPU completely busy. Consequently the graph for speed-up ratio in Fig. 7.10 is not necessarily flat for models with smaller number of elements.

The sparse matrix by vector multiplication kernel in conventional PCG algorithm is subject to execution divergence due to variations of sparse matrix pattern in different mesh structures [152]. In element-by-element PCG algorithm, this correlation with the mesh structure is rooted in the vector assembly procedure. FEM

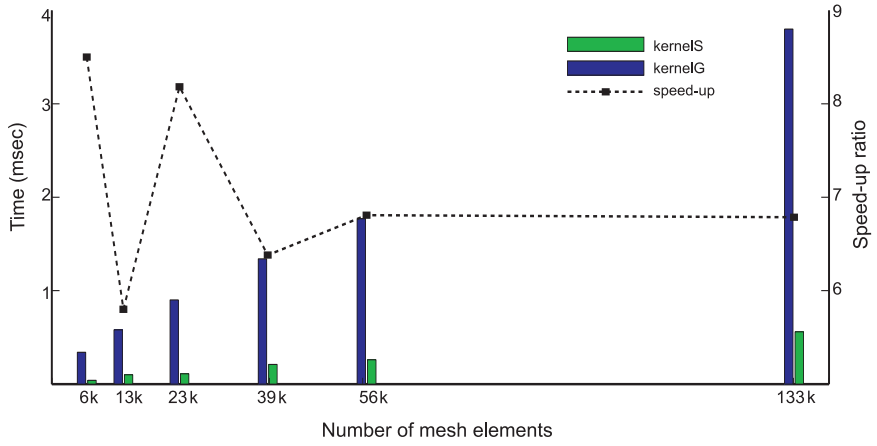


Figure 7.10: Comparison of two kernels for vector assembly, kernelG: scatter method assembly, directly storing on global memory, kernelS: scatter method assembly, two-level adding, using shared memory in the first level.

mesh structure determines data locality and non-coalesced memory access pattern in vector assembly. Employing the vector assembly presented above reduces this number of non-coalesced global memory transactions, and dependency on the mesh structure.

In matrix assembly procedure, in addition to forming the global matrix, the non-zero indices should be stored in one of the sparse storage formats. Matrix assembly using gather approach has been previously addressed in [105]. In this thesis, a code sample from CUSP library is employed*. This code performs matrix assembly in coordinate format using highly optimized kernels in THRUST library [164]. Since coordinate format does not yield the best performance for sparse matrix by vector multiplication kernel in CUSPARSE library, the storage format of the global matrix is converted into compressed sparse row (CSR) before entering the PCG loop.

*Link: `https://code.google.com/p/cusp-library/source/browse/examples/MatrixAssembly/unordered_triplets.cu`

CHAPTER 8

RESULTS

The GPU kernels for non-linear FEM matrix update and iterative solver methods discussed in the previous sections are combined together to implement an implicit dynamic FEM analysis using the total Lagrangian formulation. To measure the performance of GPU implementation, several experiments with different FEM test cases are carried out on a system with an Intel Core i7-3770 processor running at 3.40 GHz, 8 GB RAM and a GeForce GTX470 GPU. CUDA Version 5.0 is used in the experiments. The test meshes are 3D with tetrahedral elements. The CPU results are based on an optimized single-threaded C++ code. Fig. 8.1 and Table 8.1 present the mesh structure and parameters used in the tests. It is noted that the numerical results are compared and validated with Abaqus FEA [96] for a number of test cases. Dynamic simulations reveal transient damping oscillatory effects which are not present in static analysis.

Mesh	# of elements	# of nodes	# of non-zeros in global matrix
6k	6361	1295	157,383
13k	12926	2517	313,065
23k	23000	4385	551,205
39k	38924	7250	922,680
56k	56688	11099	1,374,183
133k	133784	25462	3,197,664

Table 8.1: Mesh parameters for the test cases.

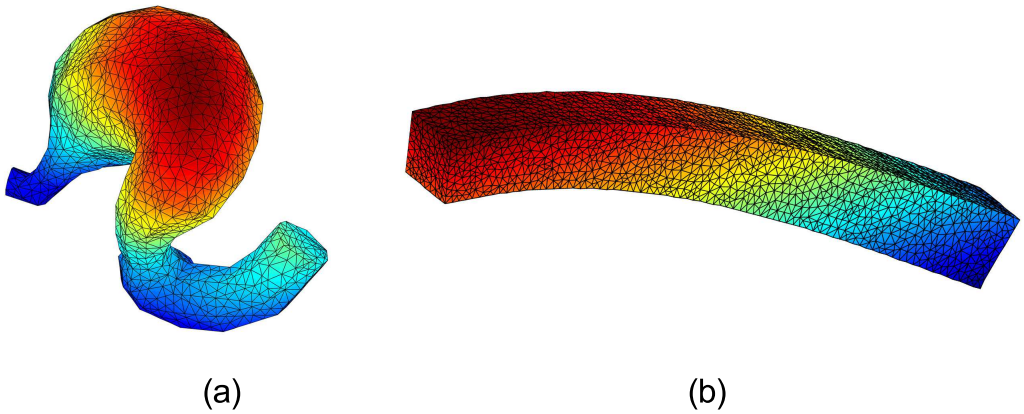


Figure 8.1: (a) The model used for 6k, 13k and 23k meshes (b) The model used for 39k, 56k and 139k meshes; mesh (a) is based on a 3D surface model of stomach from [165] meshed using iso2mesh toolkit [166].

8.1 Performance in Computation of FEM Matrices

Fig. 8.2 shows the overall execution time as well as the GFLOPS of kernels used to compute FEM matrices with elastic material behavior. The computations due to the non-constant material matrix for a Neo-Hookean material would add up to 9% to the overall time reported in Fig. 8.2. The speed-gain compared to an equivalent optimized implementation on CPU is also presented in this figure. Note that

GFLOPS are lower for smaller meshes since there are not enough number of elements to keep all the GPU cores busy; however as the number of elements increases, GFLOPS reach a maximum level determined by the GPU capacity. As observed in Fig. 8.2, the execution time is relatively linear with respect to the number of the elements. The GPU kernels for computing FEM matrices were initially developed for 2D models. The execution time for a 2D mesh with similar number of elements to that in 3D is about an order of magnitude smaller.

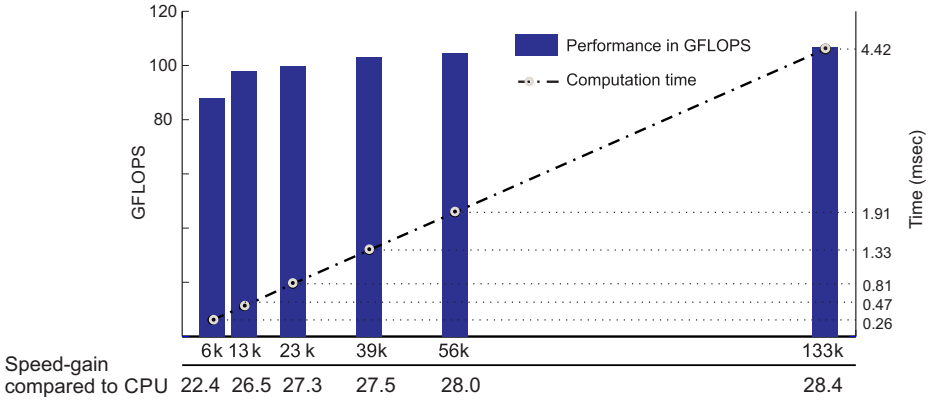


Figure 8.2: Computing performance and execution time of kernels used to generate FEM matrices based on TL non-linear FEM formulation.

Figs. 8.3, 8.4, 8.5 and 8.6 provide the performance results of the individual kernels used in computing the total Lagrangian FEM on the GPU.

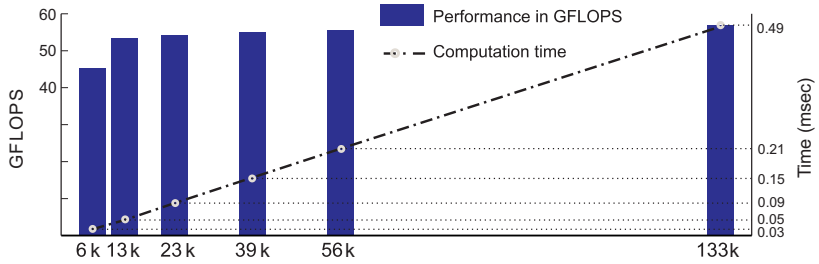


Figure 8.3: Computing performance and execution time of the kernel used to compute the linear component of the strain-displacement matrix.

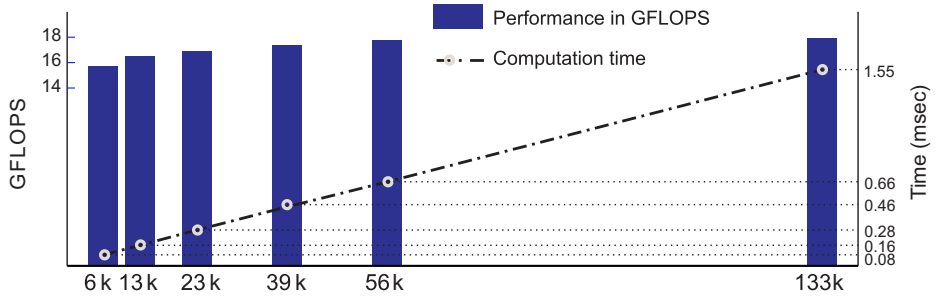


Figure 8.4: Computing performance and execution time of the individual kernel used to compute the Green strain matrix.

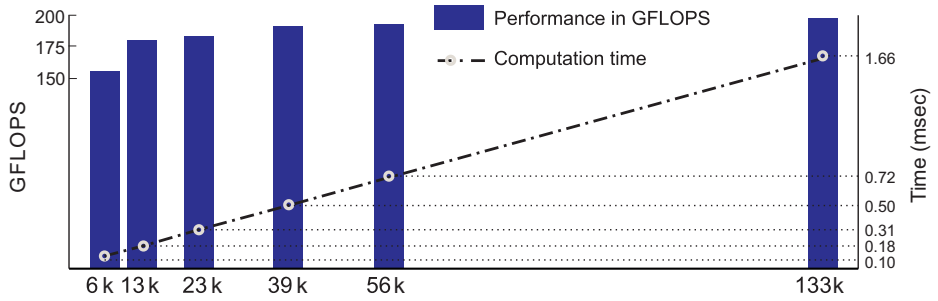


Figure 8.5: Computing performance and execution time of the kernel used to compute the linear component of the strain-displacement matrix.

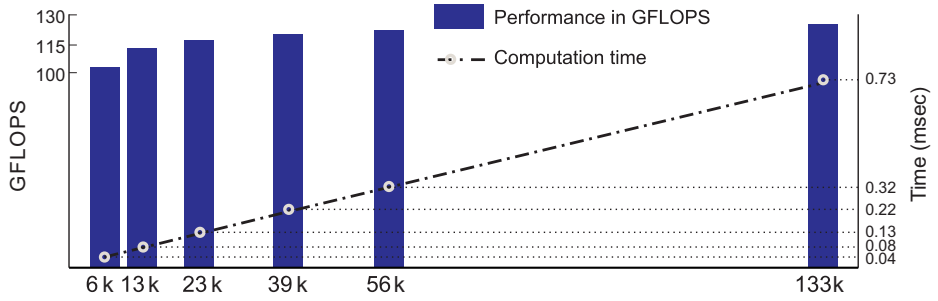


Figure 8.6: Computing performance and execution time of the kernel used to compute the nonlinear component of the strain-displacement matrix.

Computing FEM matrices is an embarrassingly parallel problem and the kernels can be scaled to be executed on multiple GPUs without extra communication overhead.

Table 8.2 presents the computing time of contact matrix $\mathbf{K}_{\alpha}^{\mathbf{e}}$ and vector ${}^{\tau}\mathbf{R}_{\mathbf{c}}^{\mathbf{e}}$ based on the penalty method defined in Eqs.(5.15) and (5.16), for a different number of contact pair sets. These results indicate that the penalty method, even for a relatively large number of contact pair sets has a minor overhead in the FEM computing. It is noted the computing time of one contact pair is in the same order to that for hundred or thousand contact pairs. This can be explained by the parallel execution of the computations. When n threads are using CUDA memory bandwidth and computing resources, thread $n+1$ does not introduce any extra overhead if there are available memory bandwidth and computing cores. This behavior is observable only for a small number of threads that do not saturate the GPU capacity.

Fig. 8.7 displays two examples of contact deformation in 2D models. In Fig. 8.7.a a thin cylindrical segment in contact with a rigid surface is under a load pressure from the top surface, while the left surface is constrained along the x-axis. The deformation is illustrated in different time samples. In Fig. 8.7.b, deformation of a thin plate due to a point load and self-collision is displayed.

Contact Pairs	1	100	1000	10000	25000
2D	0.008	0.009	0.013	0.075	0.17
3D	0.041	0.041	0.050	0.38	0.82

Table 8.2: Computation time (ms) for updating the penalty-based contact matrices, $\mathbf{K}_{\alpha}^{\mathbf{e}}$ and ${}^{\tau}\mathbf{R}_{\mathbf{c}}^{\mathbf{e}}$, for different number of contact pair sets.

Mesh refinement tests were carried out based on the h-refinement scheme presented in Section 5.3. Mesh refinement consists of mainly memory transfer operations and very few arithmetic computation, therefore a memory friendly storage of the data is crucial in achieving high performance. The tests are performed on

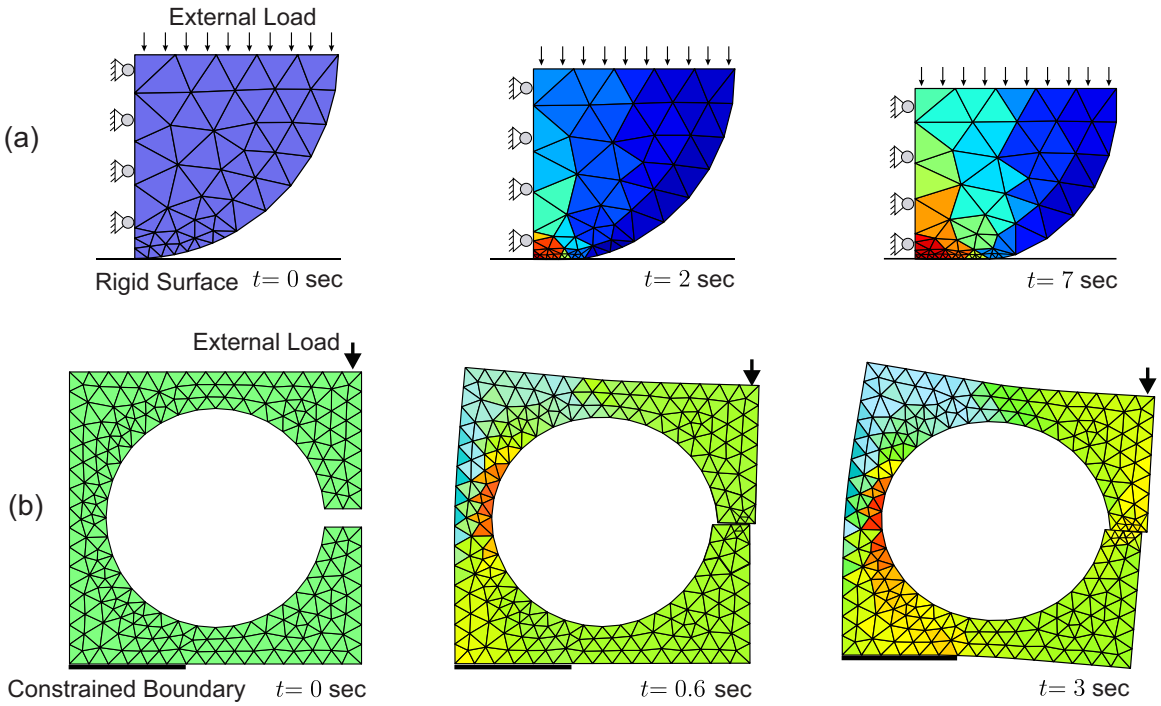


Figure 8.7: (a) Rigid/deformable contat and (b) self-collision in 2D deformable models.

a 2D mesh for different number of contact elements. Expansion of the algorithm for 3D meshes is left for future research. The small difference in the timing results for different number of contact elements is due to the fact that the problem size is not large enough to saturate the GPU resources. Consequently, the time to perform mesh refinement for up to 1000 triangular elements is similar to that for one element.

Contact Elements	1	100	1000	10000	25000
2D	0.88	0.88	1.06	1.92	2.05

Table 8.3: Computation time (ms) for updating the locally refined mesh data for different number of contact elements.

Fig. 8.8 presents an example of the local mesh refinement at a given set of contact elements, which is provided by a collision detection module.

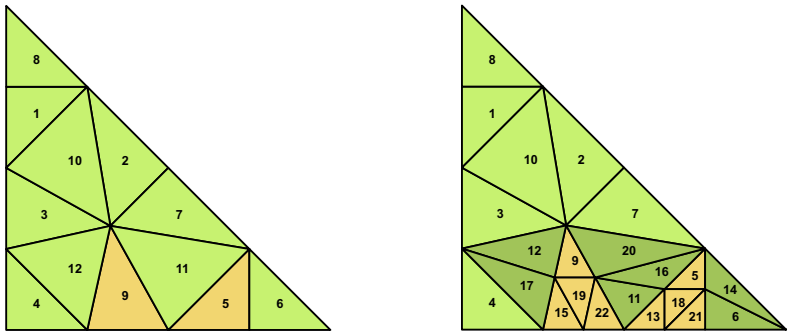


Figure 8.8: An example of the local mesh refinement: (a) Elements 5 and 9 are selected as contact elements. (b) the contact and their adjacent elements are refined accordingly.

8.2 Computing Performance of the Parallel Conjugate Gradients Method

Fig. 8.9 presents the performance and the execution time for one iteration of the element-by-element CG as well as the conventional CG method on GPU. Due to the issues associated with vector assemblage in matrix-free CG discussed in Section 7.5, and also indirect addressing of sparse matrix in the conventional CG, memory band-width is not fully utilized and smaller GFLOPS is achieved in comparison to the FEM kernels in Fig. 8.2. The conventional Jacobi-preconditioned CG algorithm on GPU yields up to 10× speed-up compared to an optimized CPU implementation.

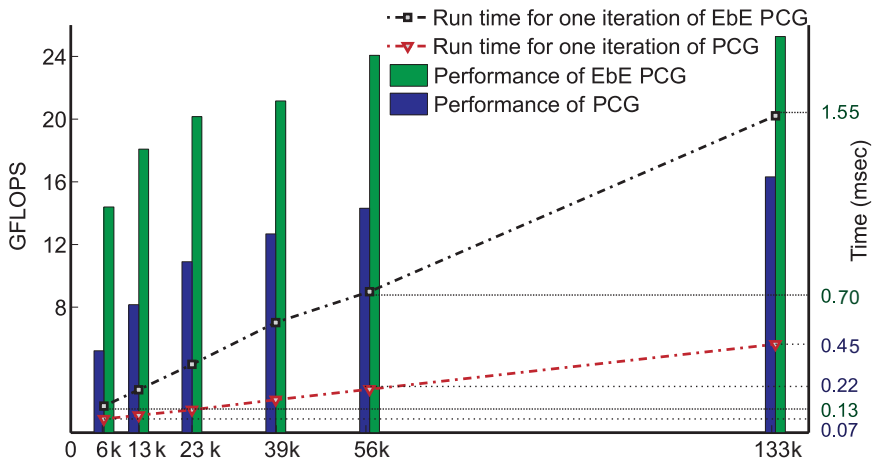


Figure 8.9: Computing performance and execution time of kernels used in one iteration of the EbE CG and the conventional CG algorithm loop on GPU.

The pie charts in Fig. 8.10 illustrate the share of these kernels in the overall timing. Matrix by vector multiplication is the most computational intensive task in the CG loop, which can be performed either in element-by-element or in assembled form. Fig. 8.11 compares the GFLOPS of these two different methods. Memory access in the EbE matrix multiplication is coalesced whereas in sparse matrix by vector multiplication, indirect memory access decreases the performance. Two different libraries of CUSPARSE [161] and CUSP [167] were used for compressed sparse row (csr) matrix by vector multiplication. Despite higher GFLOPS in EbE method, this method is slower compared to the one with the multiplication in assembled form because it involves a larger number of floating point operations. The performance of sparse matrix by vector multiplication depends on the number of non-zero entries and their distribution in the global matrix, which may vary in a mesh with certain number of elements. However, the GFLOPS of EbE matrix multiplication is expected to remain almost the same for different mesh structures.

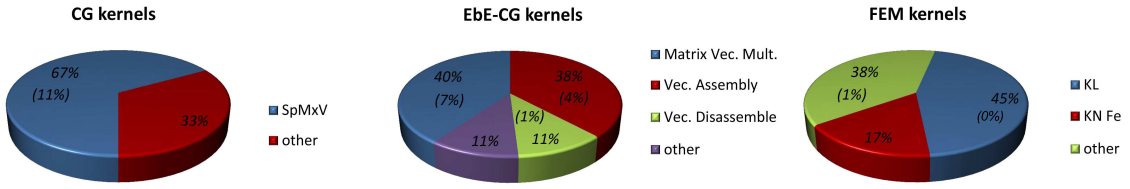


Figure 8.10: Timing distribution of different kernels used in computing FEM matrices, EbE CG and CG. The numbers in parentheses represent standard deviation of each slice. In CG and EbE CG, the portion of matrix-vector multiplication kernel increases as the problem size grows.

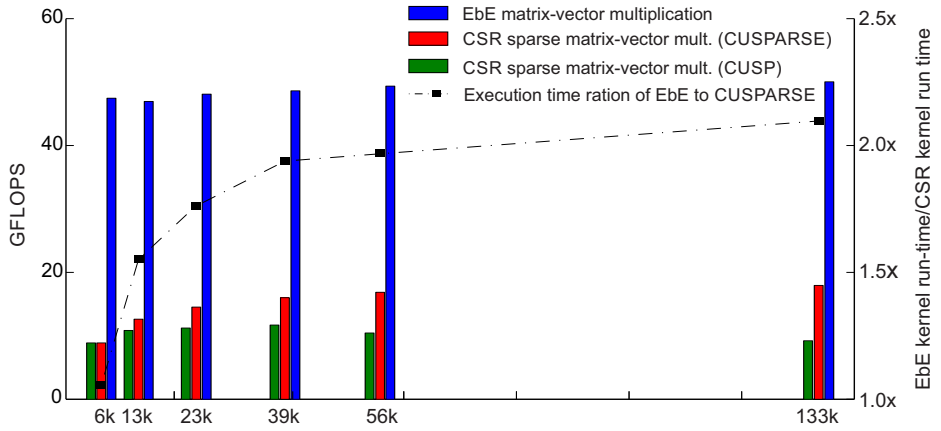


Figure 8.11: GFLOPS of matrix-vector multiplication (a) Element-by-element approach, (b) CSR multiplication using CUSPARSE library (c) CSR multiplication using CUSP library. The second y-axis compares the execution time between EbE and CUSPARSE multiplication kernels.

In the element-by-element CG algorithm, sparse matrix by vector multiplication is replaced by three kernels in Fig. 6.3: vector disassemble, elemental product and vector assemble; therefore this method has a longer execution time for each iteration compared to the best implementation of the conventional CG algorithm using CUSPARSE library. On the other hand, the CG algorithm needs to perform

the matrix assemblage step. Fig. 8.12 compares the execution time for performing 10 iterations of the element-by-element CG algorithm with 10 iterations of the CG method including matrix assembly offset. The optimized global sparse matrix assembly in csr format is performed using CUSP library [167].

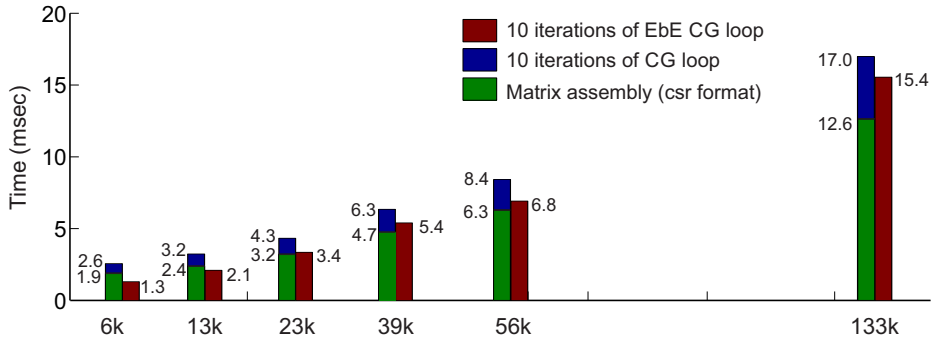


Figure 8.12: Execution time of CG and EbE CG methods with Jacobi preconditioning for 10 iterations.

The timing measurements for the two methods indicate that for small iterations (i.e. $n \simeq 10$), EbE CG is faster than CG kernel. In the intended applications of this thesis, deformation changes in one time step of the analysis are expected to be small. Therefore the iterative PCG, which starts from the previous state of the deformation, is expected to converge quickly. As an example, in the deformation simulation of the smallest mesh test case, 6k mesh, with a boundary deformation occurring at the typical hand movement speed, the Jacobi preconditioned CG algorithm converged in less than 10 iterations with a relative residual error of 1%. The number of iterations increased to about 20 for 0.1% relative error and 50 for $1e-4\%$ error tolerance. Fig. 8.13.a displays deformation of the 6k mesh under a point load. Fig. 8.13.b demonstrates large deformation of a cantilever beam constrained from one end and under a vertical load on the other end.

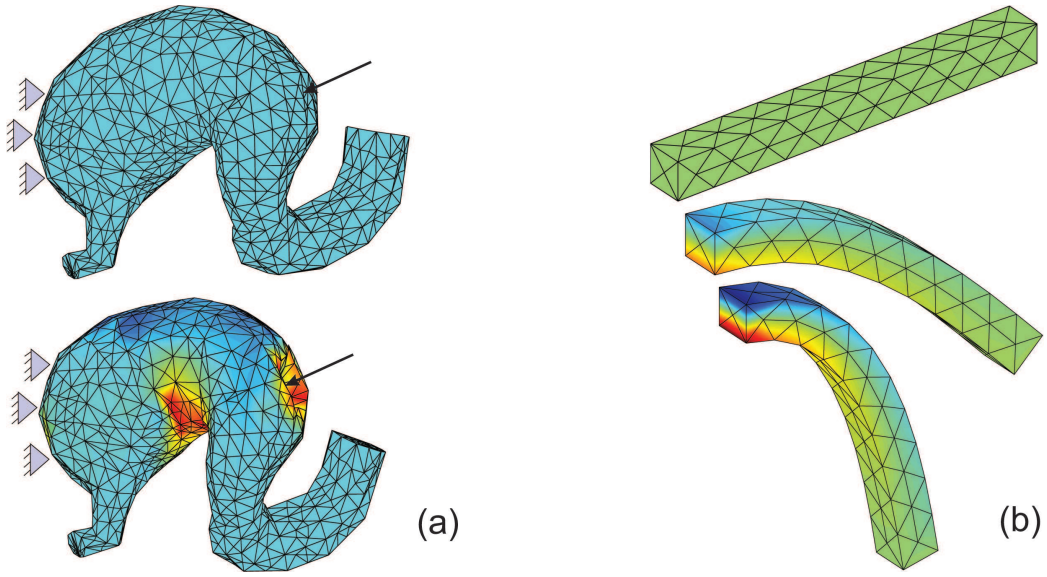


Figure 8.13: (a) Deformation of the 6k mesh under a point load (b) large deformation of a cantilever beam considering geometric nonlinearity.

A Note on the CPU-GPU Setup

Fig. 8.14 presents a general schematic of the CPU-GPU heterogeneous computing of FEM-based deformation simulation. Prior to the simulation, the model mesh is computed on CPU for once and transferred and stored on GPU to avoid extra communication time with CPU. On GPU, the nonlinear finite element matrices are computed according to the mesh data and the contact elements data provided by collision detection module. If the conventional CG is employed, FEM matrices and vectors are assembled into a global form. However, in the EbE PCG method, only the vectors need to be assembled. The computed deformation vector is sent back to the CPU at the end of the simulation loop.

As mentioned in Chapter 7, computational kernels in CUDA are presented in fine-grained thread parallelism nested within coarse-grained block parallelism. Thread blocks are independent from each other and can be mapped to different

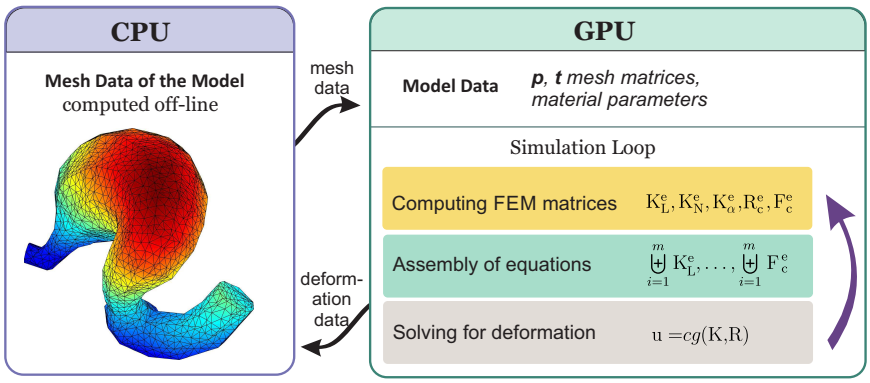


Figure 8.14: Heterogenuous FEM computing

GPU streaming multiprocessors. The hardware is versatile in scheduling these blocks to be executed on different number of multiprocessors. Fig. 8.15 adopted from [154], presents an example that a GPU device with more number of multiprocessors, automatically scales the execution of parallel blocks in the CUDA architecture. This multi-level parallelism explains the scalability of the computational kernels for different sized FEM models on different CUDA GPU devices.

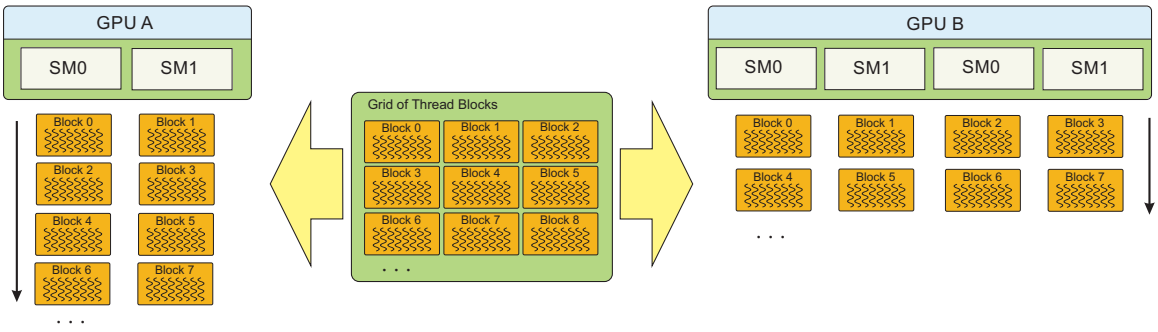


Figure 8.15: Transparent scalability in CUDA [154].

The maximum size of the models is limited due to the global memory capacity of GPU. The GTX 470 with 1280 MB memory can fit a model with up to about half a million tetrahedral elements to compute FEM matrices and solve the resulting equations.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

Deformable objects simulation has found numerous applications in medical training and surgical assisting procedures. Real-time simulation of interaction with deformable objects is challenging due to a large amount of computations that must be completed within a very short time interval. In particular, in non-linear finite element modelling, the object is meshed and partitioned to some smaller geometries referred to as finite elements and the partial differential equations arising from continuum mechanics based models are discretized in the spatial and time domain accordingly. Implicit dynamic or static formulations results in a large linear system of equations that must be solved at each simulation time step.

In this thesis, a GPU-based parallel computing approach was proposed for

fast/real-time analysis of soft objects deformation. The approach is based on the total Lagrangian nonlinear FEM, and the Newmark implicit time-integration. This formulation is general and can be used for modelling large deformations/rotations and material nonlinearities. A penalty method was employed to allow for fast and efficient multi-object deformation analysis. A local meshing paradigm was discussed to enhance the accuracy of the contact stress computations.

To solve the equations arising from the finite element formulation, iterative Jacobi-conditioned conjugate gradients method was employed. Two different implementations of element-by-element and conventional CG algorithms were presented and compared. A novel efficient approach for vector assembly on GPU, utilizing shared memory for reduced memory access time, was also proposed. It was shown that for a small number of iterations, the EbE PCG can outperform the other method. However, the conventional CG performed better with more stringent error requirements that would increase the number of iterations.

The results in Chapter 8 indicated that the computation time for FEM matrices can be up to 28 times faster on GPU (Nvidia GTX 470) compared to a sequential CPU (Intel core i7-3770) implementation. Using the memory optimization strategies discussed in Chapter 7 for computing the FEM matrices, a performance of over 100 GFLOPS was achieved. Moreover, modeling the contact using the penalty method involves modest extra computations and has small effect on the overall computation time. The implementation of the conventional PCG algorithm on GPU yielded up to 10x speed-up compared to an equivalent CPU implementation.

In real-time applications such as surgical simulations with haptic feedback, the simulation update rate could be in the order of 100-1000 Hz and therefore the

changes in object deformation within one sample period is expected to be small. This implies that the solution at the previous sample time is usually a good initial guess for the CG algorithm at the present time step, and the algorithm is expected to converge in a few iterations.The fast convergence behavior would allow for real-time deformation analysis in such applications.

In this thesis, a GTX470 GPU device was used, which was released over three years ago. The computation time of deformation analysis could be further increased using more recent GPU devices already available in the market, such as Tesla K-series or GTX Titan. However, it is emphasized that even with this device, our approach manages to satisfy the timing requirements for real-time soft-tissue deformation analysis of many practical applications. For example the total computation time for nonlinear FEM analysis with 20 iterations of Jacobi preconditioned EbE CG for a 6k mesh from the test meshes was less than 3 msec with this device. This performance is sufficient for a stable high-fidelity haptics/deformation rendering of soft objects.

9.2 Future Work

Although the results of this work are very encouraging, there are still numerous possibilities for future research. Among these are:

- Extension of the proposed vector assembly to sparse matrix assembly. By producing the assembled matrix non-zero indices in a standard sparse storage format, the optimized method for vector assembly in Section 7.6 can be extended to matrix assembly. In applications where the mesh structure is not

subject to changes, the matrix non-zero indices can be computed off-line and stored in GPU global memory. Then, extension of the vector assembly for application in matrix assembly is simply achieved by considering $m \times m$ elemental matrices as a collection of m elemental vectors of length m.

- Considering tangential friction forces in the contact model. The current penalty method is considering only normal contact forces. In general, this simplified assumption does not apply to all real-life applications and a friction model needs to be included in the contact formulation.
- Employing different adaptive meshing schemes. The proposed local mesh refinement is only applied to the contact elements in a triangular 2D mesh. A more advanced technique would consider an error estimation technique to perform one or a combination of h-, p-, or r- mesh refinement methods for 2D and 3D meshes.
- High-fidelity cutting simulation. This type of simulation which introduces topological changes to the model mesh requires a fast and efficient remeshing followed by the FEM matrix computations of the new mesh. A possible remeshing algorithm can benefit from the mesh data structures developed in Section 5.3. Then computing the FEM matrices for the new mesh can be performed using the present GPU kernels.
- Extension of the proposed computing kernels onto multiple GPUs. The FEM matrix computations on multiple GPUs can be performed without extra communication. However, efficient extension of the proposed CG methods in Section 7.4 to solve the linear system of equations on multiple GPUs will need to be investigated in future. This study will be concerned with reducing the

inter-GPU communication and balancing the computation load on multiple GPUs with a minimum effect on the convergence behavior of the CG algorithm. Overlapping the inter-GPU communication with kernel computations would improve the performance of scalability of the system.

- Developing a parallel collision detection method. A robust and efficient collision detection algorithm plays an important role in interactive deformation analysis. Detection of multi-object collisions and self-collisions in presence of large deformations is challenging and its efficient parallel implementation on GPUs can be a line of future research.
- Computational fluid dynamics in a surgical simulation for modeling effects such as blood flow would enhance the realism of the application. Parallel numerical solution of continuum mechanics based Navier-Stokes equations of fluid dynamics offers a possible extension of the research on deformation analysis in future.

APPENDIX A

VOIGT NOTATION

Owing to their symmetry, the second-order stress and strain tensors can be represented in a more compressed form, known as *Voigt notation*. This notation maps the second-order tensors to vectors. Basically the following index mapping is employed in the Voigt notation [130]:

$$\begin{aligned} &11 \rightarrow 1, \quad 22 \rightarrow 2, \quad 33 \rightarrow 3 \\ &23 \rightarrow 4, \quad 13 \rightarrow 5, \quad 12 \rightarrow 6 \end{aligned} \tag{A.1}$$

Consequently, the stress tensor $\boldsymbol{\sigma}$ is mapped to the stress vector $\boldsymbol{\sigma}$, i.e.

$$\boldsymbol{\sigma} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ & \sigma_{22} & \sigma_{23} \\ \text{Sym.} & & \sigma_{33} \end{bmatrix} \Longrightarrow \boldsymbol{\sigma} = \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \sigma_3 \\ \sigma_4 \\ \sigma_5 \\ \sigma_6 \end{bmatrix} = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} \tag{A.2}$$

In a similar vein, the strain tensor $\boldsymbol{E}$ can be redefined in Voigt form as follows

$$\boldsymbol{E} = \begin{bmatrix} E_{11} & E_{12} & E_{13} \\ & E_{22} & E_{23} \\ \text{Sym.} & & E_{33} \end{bmatrix} \Longrightarrow \mathbf{E} = \begin{bmatrix} E_1 \\ E_2 \\ E_3 \\ E_4 \\ E_5 \\ E_6 \end{bmatrix} = \begin{bmatrix} E_{11} \\ E_{22} \\ E_{33} \\ 2E_{23} \\ 2E_{13} \\ 2E_{12} \end{bmatrix} \tag{A.3}$$

There exists a factor of 2 in the last three components of strain vector, i.e. shear components, to satisfy the work conjugacy. In this way the following tensor product and vector dot product yield the same result

$$\boldsymbol{\sigma} : \boldsymbol{E} \triangleq \sigma_{ij} E_{ij} = \boldsymbol{\sigma}.\mathbf{E} = \sigma_i E_i \tag{A.4}$$

It is noted that in this thesis, ***italic bold*** letters represent ***tensors*** and **upright bold** letters represent the corresponding **Voigt form**.

Employing Voigt index mapping for a fourth-order tensor results in a two-dimensional array. As an example, linear elastic modulus tensor component C_{1213},

according to (A.1), is mapped to C_{32}. In this way, tensor definition of stress for a linear elastic material can be expressed in terms of matrix-vector product

$$\boldsymbol{\sigma} = \boldsymbol{C} : \boldsymbol{E} \quad \text{or} \quad \sigma_{ij} = C_{ijkl} E_{kl} \quad \equiv \quad \boldsymbol{\sigma} = \mathbf{CE} \quad \text{or} \quad \sigma_i = C_{ij} E_j \tag{A.5}$$

The reader is referred to [130, Chapter 5] and [168] for further discussion on this matter.

APPENDIX B

NEWMARK TIME INTEGRATION

Newmark integration scheme [20, 131] is a common method for implicit time discretization of dynamics equations arising from finite element models. A brief overview of this algorithm is presented below.

To specify the time step at which a quantity is given, we employ the same notation laid out in Section 4.2 where a left superscript t represents the quantity in the next time step, and a left superscript τ denotes a current state quantity. Consider the standard linear second-order dynamics of an object deformation given by

$$\mathbf{M}\,{}^{t}\ddot{\mathbf{U}} + \mathbf{D}\,{}^{t}\dot{\mathbf{U}} + \mathbf{K}\,{}^{t}\mathbf{U} + {}^{t}\mathbf{F} = \mathbf{0} \tag{B.1}$$

The position and velocity vector at time step t can be approximated as follows

$$\begin{aligned} {}^{t}\mathbf{U} &= {}^{\tau}\mathbf{U} + \Delta t\,{}^{\tau}\dot{\mathbf{U}} + \frac{1}{2}\Delta t^{2}\left((1-\beta)\,{}^{\tau}\ddot{\mathbf{U}} + \beta\,{}^{t}\ddot{\mathbf{U}}\right) \\ {}^{t}\dot{\mathbf{U}} &= {}^{\tau}\dot{\mathbf{U}} + \Delta t\left((1-\gamma){}^{\tau}\ddot{\mathbf{U}} + \gamma\,{}^{t}\ddot{\mathbf{U}}\right) \end{aligned} \tag{B.2}$$

where γ and β are two constants yet to be chosen. The above equations can be solved for ${}^{t}\ddot{\mathbf{U}}$ and ${}^{t}\dot{\mathbf{U}}$ resulting in

$$\begin{aligned} {}^{t}\ddot{\mathbf{U}} &= {}^{t}\hat{\ddot{\mathbf{U}}} + \frac{2}{\beta\Delta t^2}\,{}^{t}\mathbf{U} \\ {}^{t}\dot{\mathbf{U}} &= {}^{t}\hat{\dot{\mathbf{U}}} + \frac{2\gamma}{\beta\Delta t}\,{}^{t}\mathbf{U} \end{aligned} \tag{B.3}$$

where

$$\begin{aligned} {}^{t}\hat{\ddot{\mathbf{U}}} &= -\frac{2}{\beta\Delta t^2}\,{}^{\tau}\mathbf{U} - \frac{2}{\beta\Delta t}\,{}^{\tau}\dot{\mathbf{U}} - \frac{1-\beta}{\beta}\,{}^{\tau}\ddot{\mathbf{U}}_n \\ {}^{t}\hat{\dot{\mathbf{U}}} &= -\frac{2\gamma}{\beta\Delta t}\,{}^{\tau}\mathbf{U} + (1-\frac{2\gamma}{\beta})\,{}^{\tau}\dot{\mathbf{U}} + (1-\frac{\gamma}{\beta})\Delta t\,{}^{\tau}\ddot{\mathbf{U}} \end{aligned} \tag{B.4}$$

Substituting (B.3) and (B.4) into the dynamics equation in (B.1) yields

$$\hat{\mathbf{A}}\,{}^{t}\mathbf{U} = \hat{\mathbf{b}} \tag{B.5}$$

where

$$\begin{aligned} \hat{\mathbf{A}} &= \frac{2}{\beta\Delta t^2}\mathbf{M} + \frac{2\gamma}{\beta\Delta t}\mathbf{D} + \mathbf{K} \\ \hat{\mathbf{b}} &= -({}^{t}\mathbf{F} + \mathbf{D}\,{}^{t}\hat{\dot{\mathbf{U}}} + \mathbf{M}\,{}^{t}\hat{\ddot{\mathbf{U}}}) \end{aligned} \tag{B.6}$$

Note that to obtain the displacement vector ${}^{t}\mathbf{U}$, one needs to solve the system of linear equations in (B.5) similar to that of the static case. The following choice of the parameters β and γ guarantees the stability of the discretized system [20, 131]

$$\gamma \geq 0.5 \qquad \beta \geq 0.5(0.5+\gamma)^2 \tag{B.7}$$

APPENDIX C

GAUSSIAN QUADRATURE INTEGRATION

Guassian quadrature is an effective numerical integration technique widely used in FEM for computing stiffness matrix, mass matrix and force vectors. Gaussian quadrature approximates a definite integration via a weighted sum of the integrand function values at specific points within the integration domain. n-point Guassian quadrature yields the exact result for polynomial functions of order at most $2n-1$.

The basic idea of this integration rule can be described by a simple example. Consider the function $f(x)$ in Fig. C.1. It is desired to approximate the integral of $\int_{-1}^{+1} f(x)\mathrm{d}x$ by $w_0 f(x_0) + w_1 f(x_1)$. In contrast to the trapezoidal integration which

uses fixed end points of the integral domain, x_0 and x_1 are variables to be determined. Since the aim is to produce exact results for integration of polynomials up to degree three, the equations for determining the four unknown parameters w_1, w_2, x_0 and x_1 are given by

$$w_0 f(x_0) + w_1 f(x_1) = \int_{-1}^{+1} f(x)\mathrm{d}x \quad \text{for} \quad f(x) = 1, \tag{C.1}$$

$$f(x) = x, \tag{C.2}$$

$$f(x) = x^2, \tag{C.3}$$

$$f(x) = x^3 \tag{C.4}$$

Solving the equations in (C.1) to (C.4) yields

$$w_0 = w_1 = 1, \quad x_0 = \frac{1}{\sqrt{3}} \quad \text{and} \quad x_1 = \frac{-1}{\sqrt{3}} \tag{C.5}$$

These parameters give exact solution for integrating polynomial functions of degree 3 or less in $[-1,1]$ domain. The above formula, with a change of variable can be applied to integrations with arbitrary domain other than $[-1,1]$. The example above, can be expanded for higher dimension integrations. The integration points and weights for different finite elements such as triangular and tetrahedral are computed and reported in [169].

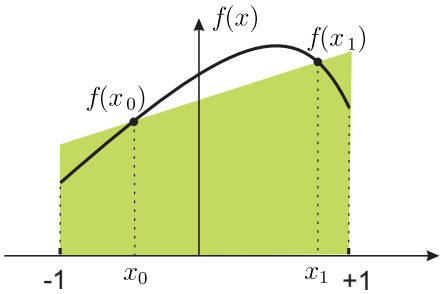


Figure C.1: Gaussian quadrature integration

APPENDIX D

SHAPE FUNCTIONS

Shape functions allow interpolation within the discrete FEM nodal points. The finite element matrices can be developed in a systematic approach by performing the computations on a local coordinates system and map it to the global coordinates system. An example helps to clarify this matter. Fig. D.1 displays a triangle in the global and local coordinates*,

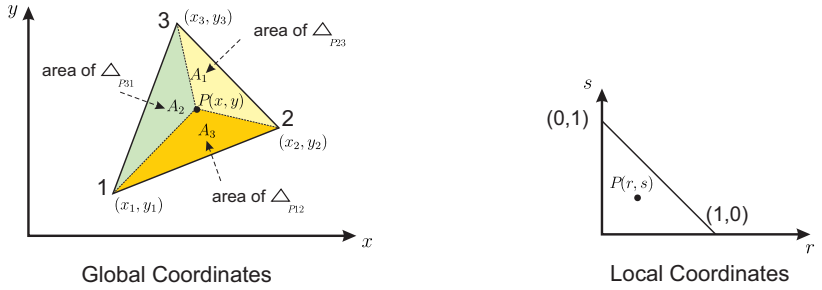


Figure D.1: Three node triangle in local and global coordinates systems [20]

.

*also known as natural coordinates or iso-parametric coordinates.

The global Cartesian coordinates of point P in Fig. D.1 are given by

$$x = \sum_{i=1}^{3} h_i x_i \tag{D.1a}$$

$$y = \sum_{i=1}^{3} h_i y_i \tag{D.1b}$$

where x_i's and y_i's are the global coordinates of the triangle nodes. The shape function parameters in this example, i.e. h_i's are defined as

$$h_1 = \frac{A_1}{A_1 + A_2 + A_3}, \qquad h_2 = \frac{A_2}{A_1 + A_2 + A_3}, \qquad h_3 = \frac{A_3}{A_1 + A_2 + A_3} \tag{D.2}$$

where A_1, A_2 and A_3 are the areas of triangles displayed in Fig. D.1. Following the expression of a point position in Eq.(D.1) based on the nodal values, the displacement value $\mathbf{u} = {}^{\tau}\mathbf{x} - {}^{0}\mathbf{x}$ can also be expressed using h_i parameters, i.e.

$$u_x = \sum_{i=1}^{3} h_i u_{xi} \tag{D.3a}$$

$$u_y = \sum_{i=1}^{3} h_i u_{yi} \tag{D.3b}$$

where u_{xi} and u_{yi} are the nodal displacement values. Since the same h_i parameters are used to express the geometry and the displacement field in (D.1) and (D.3), this method is known as *iso-parametric* formulation. Using the local coordinates system in Fig. D.1, shape function parameters are expressed in terms of the local coordinates as [20]

$$h_1 = 1 - r - s, \qquad h_2 = r, \qquad h_3 = s \tag{D.4}$$

Although it is possible to directly work with the global coordinates system for elements with simple geometries, in general it is more convenient to perform numerical derivatives and integrations in the local coordinate system. Then the results are mapped to the global coordinates system. To this end, a *Jacobian* matrix defined below is used for mapping

$$\begin{bmatrix} \frac{\partial}{\partial r} \\ \frac{\partial}{\partial s} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{bmatrix} \quad \Longleftrightarrow \quad \frac{\boldsymbol{\partial}}{\boldsymbol{\partial r}} = \mathbf{J}\frac{\boldsymbol{\partial}}{\boldsymbol{\partial x}} \tag{D.5}$$

$$\frac{\boldsymbol{\partial}}{\boldsymbol{\partial x}} = \mathbf{J}^{-1}\frac{\boldsymbol{\partial}}{\boldsymbol{\partial r}} \tag{D.6}$$

Using Eq.(D.6), spatial derivatives with respect to the global coordinates system can be expressed in terms of the local coordinates. Additionally, surface integrations on the local coordinates are transformed to the global coordinates using determinant of the Jacobian matrix [20], i.e.

$$\int_S . \, \mathrm{d}x\mathrm{d}y = \int_S . \det(\mathbf{J})\mathrm{d}r\mathrm{d}s \tag{D.7}$$

Examples on finite elements with different number of nodes and geometries and further discussion on this topic can be found in any standard finite element textbook such as [20, 131].

APPENDIX E

TECHNICAL SPECIFICATIONS OF GTX 470

The specifications of NVIDIA GTX 470 with code name GF 100 is listed in Table E.1. GTX 470 belongs to Fermi architecture. GF 100 devices provide compute capability 2.0.

Computing Cores	Memory Size	Memory Bandwidth	Core Clock	Bus Interface	Transistors
448	1280 MB	133.9 GB/s	607 MHz	PCIe 2.0 ×16	3200M

Table E.1: Specifications of GTX 470

Fig. E.2 adopted from [170] presents an overview of Fermi architecture consisting of up to 16 streaming multiprocessors. SMs in compute capability 2.0 are

equipped with 32 cores*. GTX 470 features 15 SMs. CUDA blocks are scheduled and distributed betweens SMs via GigaThread. Threads within each SM are scheduled in groups of 32, known as warps, via dual Warp Schedulers per SM. There are 32k 32-bit register files per SM. The 64 KB of on-chip memory per SM can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache. More technical details can be found in [154,170].

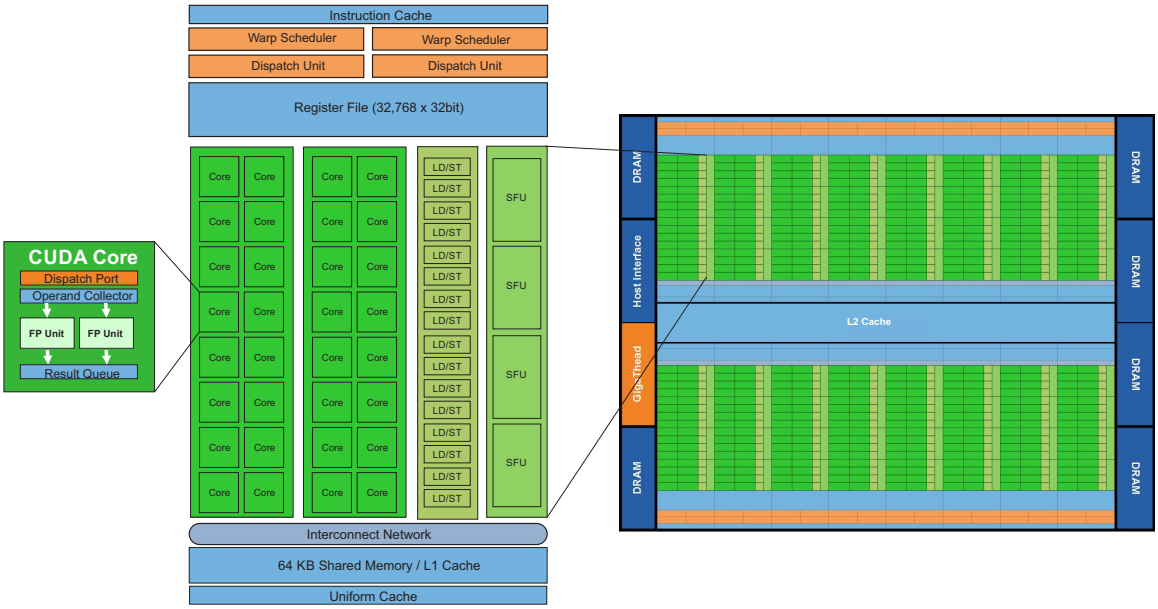


Figure E.2: Fermi architecture [170].

*CUDA cores are also known as *streaming processors* or *shader units*

BIBLIOGRAPHY

[1] A. Liu, F. Tendick, K. Cleary, and C. Kaufmann, "A survey of surgical simulation: Applications, technology, and education," *Presence -Cambridge Massachusetts*, vol. 12, no. 6, pp. 599–614, 2003.

[2] K. S. Gurusamy, R. Aggarwal, L. Palanivelu, and B. R. Davidson, "Virtual reality training for surgical trainees in laparoscopic surgery," *Cochrane Database Syst Rev*, vol. 1, no. 4, 2009.

[3] H. Singh, E. J. Thomas, L. A. Petersen, and D. M. Studdert, "Medical errors involving trainees: a study of closed malpractice claims from 5 insurers," *Archives of Internal Medicine*, vol. 167, no. 19, pp. 2030–2036, 2007.

[4] A. A. Gawande, M. J. Zinner, D. M. Studdert, and T. A. Brennan, "Analysis of errors reported by surgeons at three teaching hospitals," *Surgery*, vol. 133, no. 6, pp. 614–621, 2003.

[5] S. Barry Issenberg, W. C. McGaghie, E. R. Petrusa, D. Lee Gordon, and R. J. Scalese, "Features and uses of high-fidelity medical simulations that lead to effective learning: a beme systematic review," *Medical teacher*, vol. 27, no. 1, pp. 10–28, 2005.

[6] B. Cowan, D. Rojas, B. Kapralos, K. Collins, and A. Dubrowski, "Spatial sound and its effect on visual quality perception and task performance within a virtual environment," in *Proceedings of Meetings on Acoustics*, vol. 19, 2013, pp. 1–7.

[7] B. Marami, S. Sirouspour, and D. W. Capson, "Model-based deformable registration of preoperative 3d to intraoperative low-resolution 3d and 2d sequences of mr images," in *MICCAI (1)'11*, 2011, pp. 460–467.

[8] B. Glocker, N. Komodakis, N. Paragios, G. Tziritas, and N. Navab, "Inter and intra-modal deformable registration: Continuous deformations meet efficient optimal linear programming," in *Information Processing in Medical Imaging*. Springer, 2007, pp. 408–420.

[9] M. Ferrant, S. K. Warfield, A. Nabavi, F. A. Jolesz, and R. Kikinis, "Registration of 3d intraoperative mr images of the brain using a finite element biomechanical model," in *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2000*. Springer, 2000, pp. 19–28.

[10] M. Ferrant, A. Nabavi, B. Macq, F. A. Jolesz, R. Kikinis, and S. K. Warfield, "Registration of 3-d intraoperative mr images of the brain using a finite-element biomechanical model," *Medical Imaging, IEEE Transactions on*, vol. 20, no. 12, pp. 1384–1397, 2001.

[11] K. Miller, A. Wittek, G. Joldes, A. Horton, T. Dutta-Roy, J. Berger, and L. Morriss, "Modelling brain deformations for computer-integrated neurosurgery," *International Journal for Numerical Methods in Biomedical Engineering*, vol. 26, no. 1, pp. 117–138, 2010.

[12] R. Mafi, S. Sirouspour, B. Mahdavikhah, B. Moody, K. Elizeh, A. Kinsman, and N. Nicolici, "A parallel computing platform for real-time haptic interaction with deformable bodies," *IEEE Transactions on Haptics*, vol. 3, no. 3, pp. 211–223, July-September 2010.

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

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

[15] M. Bro-Nielsen, "Finite element modeling in surgery simulation," *Proceedings of the IEEE*, vol. 86, no. 3, pp. 490–503, March 1998.

[16] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable real-time deformations," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. ACM, 2002, pp. 49–54.

[17] L. F. da Silva, A. Öchsner, and R. D. Adams, *Handbook of adhesion technology*. Springer, 2011.

[18] J. Bonet and R. D. Wood, *Nonlinear continuum mechanics for finite element analysis*, 2nd ed. Cambridge university press, 2008.

[19] E. Madenci and I. Guven, *The finite element method and applications in engineering using ANSYS®*. Springer, 2006.

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

[21] Z.-H. Zhong, *Finite Element Procedures for Contact-Impact Problems*. Oxford University Press, 1993.

[22] G. Brussino and V. Sonnad, "A comparison of direct and preconditioned iterative techniques for sparse, unsymmetric systems of linear equations," *International journal for numerical methods in engineering*, vol. 28, no. 4, pp. 801–815, 1989.

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

[24] M. Křížek, *Conjugate Gradient Algorithms and Finite Element Methods*. Springer, 2004.

[25] M. Bro-Nielsen and S. Cotin, "Real-time volumetric deformable models for surgery simulation using finite elements and condensation," *Computer Graphics Forum*, vol. 15, no. 3, pp. 57–66, 1996.

[26] D. L. James and D. K. Pai, "Artdefo: accurate real time deformable objects," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 1999, pp. 65–72.

[27] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[28] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," Intel, white paper, March 2005.

[29] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *Application Specific Processors, 2008. SASP 2008. Symposium on*. IEEE, 2008, pp. 101–107.

[30] A. Heinecke, M. Klemm, and H. Bungartz, "From GPGPU to many-core: Nvidia fermi and intel many integrated core architecture," *Computing in Science & Engineering*, vol. 14, no. 2, pp. 78–83, 2012.

[31] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2010.

[32] K. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak, "The polder computing environment: a system for interactive distributed simulation," *Concurrency and Computation: Practice and experience*, vol. 14, no. 13-15, pp. 1313–1335, 2002.

[33] R. Mafi and S. Sirouspour, "GPU-based acceleration of computations in nonlinear finite element deformation analysis," *International Journal of Numerical Methods in Biomedical Engineering*, p. to appear, 2013.

[34] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[35] B. Chapman, G. Jost, and R. Van Der Pas, *Using OpenMP: portable shared memory parallel programming*. The MIT Press, 2008.

[36] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[37] S. Blair-Chappell and A. Stokes, *Parallel Programming with Intel Parallel Studio XE*. John Wiley & Sons, 2012.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of parallel and distributed computing*, vol. 68, no. 10, pp. 1370–1380, 2008.

[39] G. Carey, E. Barragy, R. Mclay, and M. Sharma, "Element-by-element vector and parallel computations," *Communications in applied numerical methods*, vol. 4, no. 3, pp. 299–307, 1988.

[40] K. Bathe, E. Ramm, and E. Wilson, "Finite element formulations for large deformation dynamic analysis," *International Journal for Numerical Methods in Engineering*, vol. 9, no. 2, pp. 353–386, 1975.

[41] 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.

[42] B. J. Weghorst, H. Gladstone, G. Raugi, and M. Ganter, "Fast finite element modeling for surgical simulation," *Student Health Technol. Inf*, vol. 62, pp. 55–61, 1999.

[43] J. Brown, S. Sorkin, J.-C. Latombe, K. Montgomery, and M. Stephanides, "Algorithmic tools for real-time microsurgery simulation," *Medical Image Analysis*, vol. 6, no. 3, pp. 289–300, 2002.

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

[45] G. E. Farin, *Curves and Surfaces for Computer-Aided Geometric Design: A Practical Code*. Academic Press, Inc., 1996.

[46] S. A. Cover, N. F. Ezquerra, J. F. O'Brien, R. Rowe, T. Gadacz, and E. Palm, "Interactively deformable models for surgery simulation," *Computer Graphics and Applications, IEEE*, vol. 13, no. 6, pp. 68–75, 1993.

[47] T. W. Sederberg and S. R. Parry, "Free-form deformation of solid geometric models," in *ACM Siggraph Computer Graphics*, vol. 20, no. 4. ACM, 1986, pp. 151–160.

[48] S. F. Gibson, "3d chainmail: a fast algorithm for deforming volumetric objects," in *Proceedings of the 1997 symposium on Interactive 3D graphics*. ACM, 1997, pp. 149–154.

[49] S. Misra, K. Ramesh, and A. M. Okamura, "Modeling of tool-tissue interactions for computer-based surgical simulation: a literature review," *Presence: Teleoperators and Virtual Environments*, vol. 17, no. 5, pp. 463–491, 2008.

[50] P. Moore and D. Molloy, "A survey of computer-based deformable models," in *Machine Vision and Image Processing Conference, 2007. IMVIP 2007. International*. IEEE, 2007, pp. 55–66.

[51] S. Cotin, "Computer based interactive medical simulation," Lille University, Tech. Rep., June 2013.

[52] J. Georgii and R. Westermann, "Mass-spring systems on the GPU," *Simulation modelling practice and theory*, vol. 13, no. 8, pp. 693–702, 2005.

[53] M. Dokainish and K. Subbaraj, "A survey of direct time-integration methods in computational structural dynamics:i. explicit methods," *Computers & Structures*, vol. 32, no. 6, pp. 1371–1386, 1989.

[54] K. Subbaraj and M. Dokainish, "A survey of direct time-integration methods in computational structural dynamics:ii. implicit methods," *Computers & Structures*, vol. 32, no. 6, pp. 1387–1401, 1989.

[55] J. Brown, S. Sorkin, C. Bruyns, J.-C. Latombe, K. Montgomery, and M. Stephanides, "Real-time simulation of deformable objects: Tools and application," in *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*. IEEE, 2001, pp. 228–258.

[56] G. Bianchi, M. Harders, and G. Székely, "Mesh topology identification for mass-spring models," in *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2003*. Springer, 2003, pp. 50–58.

[57] B. A. Lloyd, G. Székely, and M. Harders, "Identification of spring parameters for deformable object simulation," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, no. 5, pp. 1081–1094, 2007.

[58] J. N. Reddy, *An introduction to the finite element method*. McGraw-Hill New York, 2006.

[59] S. P. DiMaio and S. E. Salcudean, "Needle insertion modeling and simulation," *Robotics and Automation, IEEE Transactions on*, vol. 19, no. 5, pp. 864–875, 2003.

[60] X. Wu, M. Downes, T. Goktekin, and F. Tendick, "Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes," *Computer Graphics Forum*, vol. 20, no. 3, pp. 349–358, 2001.

[61] Z. A. Taylor, M. Cheng, and S. Ourselin, "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units," *IEEE Transactions on Medical Imaging*, vol. 27, pp. 650–663, 2008.

[62] C. Dick, J. Georgii, and R. Westermann, "A real-time multigrid finite hexahedra method for elasticity simulation using CUDA," *Simulation Modelling Practice and Theory*, vol. 19, no. 2, pp. 801–816, 2011.

[63] H. Courtecuisse, H. Jung, J. Allard, C. Duriez, D. Y. Lee, and S. Cotin, "GPU-based real-time soft tissue deformation with cutting and haptic feedback," *Progress in Biophysics and Molecular Biology*, vol. 103, no. 2, pp. 159–168, 2010.

[64] K. Miller, G. Joldes, D. Lance, and A. Wittek, "Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation," *Communications in Numerical Methods in Engineering*, vol. 23, no. 2, pp. 121–134, 2007.

[65] C. Felippa and B. Haugen, "A unified formulation of small-strain corotational finite elements: I. theory," *Computer Methods in Applied Mechanics and Engineering*, vol. 194, no. 21-24, pp. 2285–2335, 2005.

[66] G.-R. Liu, *Meshfree methods: moving beyond the finite element method*. CRC press, 2010.

[67] S. R. Idelsohn, E. Oñate, N. Calvo, and F. Del Pin, "The meshless finite element method," *International Journal for Numerical Methods in Engineering*, vol. 58, no. 6, pp. 893–912, 2003.

[68] G. Zhang, A. Wittek, G. Joldes, X. Jin, and K. Miller, "A three-dimensional nonlinear meshfree algorithm for simulating mechanical responses of soft tissue," *Engineering Analysis with Boundary Elements*, p. to appear, 2013.

[69] S. De, J. Kim, Y.-J. Lim, and M. A. Srinivasan, "The point collocation-based method of finite spheres (pcmfs) for real time surgery simulation," *Computers & structures*, vol. 83, no. 17, pp. 1515–1525, 2005.

[70] Y.-J. Lim and S. De, "Real time simulation of nonlinear tissue response in virtual surgery using the point collocation-based method of finite spheres," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 31, pp. 3011–3024, 2007.

[71] P. Wang, A. Becker, I. Jones, A. Glover, S. Benford, C. Greenhalgh, and M. Vloeberghs, "Virtual reality simulation of surgery with haptic feedback based on the boundary element method," *Computers & structures*, vol. 85, no. 7, pp. 331–339, 2007.

[72] D. L. James and D. K. Pai, "A unified treatment of elastostatic contact simulation for real time haptics," in *ACM SIGGRAPH 2005 Courses*. ACM, 2005, p. 141.

[73] 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.

[74] M. Mahvash, V. Hayward, and J. Lloyd, "Haptic rendering of tool contact," in *Proc. Eurohaptics 2002*, 2002, pp. 110–115.

[75] P. Jacobs, M. J. Fu, and M. C. Çavuşoğlu, "High fidelity haptic rendering of frictional contact with deformable objects in virtual environments using multi-rate simulation," *The International Journal of Robotics Research*, vol. 29, no. 14, pp. 1778–1792, 2010.

[76] I. Peterlík, M. Sedef, C. Basdogan, and L. Matyska, "Real-time visio-haptic interaction with static soft tissue models having geometric and material nonlinearity," *Computers & Graphics*, vol. 34, no. 1, pp. 43–54, 2010.

[77] M. Mahvash and V. Hayward, "High-fidelity haptic synthesis of contact with deformable bodies," *IEEE Computer Graphics and Applications*, vol. 24, no. 2, pp. 48–55, 2004.

[78] G. Hirota, S. Fisher *et al.*, "An improved finite-element contact model for anatomical simulations," *The Visual Computer*, vol. 19, no. 5, pp. 291–309, 2003.

[79] J. Barbic and D. L. James, "Six-dof haptic rendering of contact between geometrically complex reduced deformable models," *Haptics, IEEE Transactions on*, vol. 1, no. 1, pp. 39–52, 2008.

[80] A.-I. Ştefancu, S.-C. Melenciuc, and M. Budescu, "Penalty based algorithms for frictional contact problems," *Bulletin of the Polytechnic Institute of Iasi - Construction*, vol. 61, no. 3, pp. 119–129, 2011.

[81] N. Galoppo, M. A. Otaduy, P. Mecklenburg, M. Gross, and M. C. Lin, "Fast simulation of deformable models in contact using dynamic deformation textures," in *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 2006, pp. 73–82.

[82] S. Cotin, H. Delingette, and N. Ayache, "Real-time elastic deformations of soft tissues for surgery simulation," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 5, no. 1, pp. 62–73, 1999.

[83] C. Duriez, F. Dubois, A. Kheddar, and C. Andriot, "Realistic haptic rendering of interacting deformable objects in virtual environments," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 1, pp. 36–47, 2006.

[84] M. Sofonea, W. Han, and M. Shillor, *Analysis and Approximation of Contact problems with Adhesion or Damage*. CRC Press, 2010.

[85] K. G. Murty, *Linear complementarity, linear and nonlinear programming*. Heldermann Berlin, 1988.

[86] P. Wriggers, "Finite element algorithms for contact problems," *Archives of Computational Methods in Engineering*, vol. 2, no. 4, pp. 1–49, 1995.

[87] G. Gilardi and I. Sharf, "Literature survey of contact dynamics modelling," *Mechanism and machine theory*, vol. 37, no. 10, pp. 1213–1239, 2002.

[88] V. A. Yastrebov, *Numerical Methods in Contact Mechanics*. John Wiley & Sons, 2013.

[89] J. Mosegaard, P. Herborg, and T. S. Sorensen, "A GPU accelerated spring mass system for surgical simulation," *Studies in health technology and informatics*, vol. 111, pp. 342–348, 2005.

[90] I. Parberry, *Introduction to Game Physics with Box2D*. CRC PressI Llc, 2013.

[91] *CUDA C best practices guide*, nVidia Corporation, Oct. 2012.

[92] K. O. W. Group *et al.*, "The openCL specification," *A. Munshi, Ed*, 2008.

[93] C. A. D. Leon, S. Eliuk, and H. T. Gomez, "Simulating soft tissues using a GPU approach of the mass-spring model," in *Virtual Reality Conference (VR), 2010 IEEE*. IEEE, 2010, pp. 261–262.

[94] B. H. Topping and A. I. Khan, *Parallel finite element computations*. Saxe-Coburg Publications Edinburgh, 1996.

[95] D. T. Nguyen, *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*. Springer, 2006.

[96] Dassault Systèmes, "ABAQUS FEA," http://academy.3ds.com/software/simulia/.

[97] ANSYS, Inc., "ANSYS Academic Research, Release 14.0," http://www.ansys.com/Products.

[98] Rocscience Inc., "RS3," http://www.rocscience.com/products/16/RS3, 2013.

[99] COMSOL, "COMSOL Multiphysics," http://www.comsol.com/.

[100] A. Rao, "MPI-based parallel finite element approaches for implicit nonlinear dynamic analysis employing sparse PCG solvers," *Advances in Engineering Software*, vol. 36, no. 3, pp. 181–198, 2005.

[101] W. D. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. the MIT Press, 1999.

[102] R. Paz, M. Storti, H. Castro, and L. DalcÃn, "Using hybrid parallel programming techniques for the computation, assembly and solution stages in finite element codes," *Latin American applied research*, vol. 41, pp. 365–377, 2011.

[103] G. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations on GPU–application to neurosurgical simulation," *Computer methods in applied mechanics and engineering*, vol. 199, no. 49, pp. 3305–3314, 2010.

[104] W. Hackbusch, *Multi-grid methods and applications*. Springer-Verlag Berlin, 1985.

[105] C. Cecka, A. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International journal for numerical methods in engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[106] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures," *International Journal for Numerical Methods in Fluids*, vol. 71, no. 1, pp. 80–97, 2013.

[107] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications," in *Computer Graphics Forum*. Wiley Online Library, 2012.

[108] G. Carey and B. Jiang, "Element-by-element linear and nonlinear solution schemes," *Communications in applied numerical methods*, vol. 2, no. 2, pp. 145–153, 2005.

[109] I. Kiss, S. Gyimothy, Z. Badics, and J. Pavo, "Parallel realization of the element-by-element fem technique by cuda," *Magnetics, IEEE Transactions on*, vol. 48, no. 2, pp. 507–510, 2012.

[110] T. A. Davis, *Direct methods for sparse linear systems*. Siam, 2006.

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

[112] B. Mahdavikhah, R. Mafi, S. Sirouspour, and N. Nicolici, "Haptic rendering of deformable objects using a multiple FPGA parallel computing architecture," in *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010.

[113] A. Cevahir, A. Nukada, and S. Matsuoka, "High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning," *Computer Science-Research and Development*, vol. 25, no. 1-2, pp. 83–91, 2010.

[114] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *International Journal for Numerical Methods in Fluids*, vol. 64, no. 10-12, pp. 1254–1273, 2010.

[115] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-GPU platform," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 583–592.

[116] Y. Liu, W. Zhou, and Q. Yang, "A distributed memory parallel element-by-element scheme based on jacobi-conditioned conjugate gradient for 3d finite element analysis," *Finite Elements in Analysis and Design*, vol. 43, no. 6, pp. 494–503, 2007.

[117] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *Journal of Computational and Applied Mathematics*, vol. 236, no. 15, pp. 3584–3590, 2012.

[118] 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.

[119] M. A. Heroux, P. Raghavan, and H. D. Simon, *Parallel processing for scientific computing*. SIAM, 2006, vol. 20.

[120] G. T. Mase and G. E. Mase, *Continuum mechanics for engineers*. Crc Press, 2010.

[121] A. A. Shabana, *Computational continuum mechanics*. Cambridge University Press, 2011.

[122] W. M. Lai, D. H. Rubin, D. Rubin, and E. Krempl, *Introduction to continuum mechanics*, 4th ed. Butterworth-Heinemann, 2009.

[123] Y. Zhuang, "Real-time simulation of physically realistic global deformations," Ph.D. dissertation, 2000, chair-John Canny.

[124] N. Ottosen and M. Ristinmaa, *Mechanics of Constitutive Modeling*. Elsevier, 2005.

[125] J. Rao, "Fundamentals of elasticity," in *History of Rotating Machinery Dynamics*. Springer, 2011, pp. 45–47.

[126] A. Maceri, *Theory of Elasticity*. Springer Berlin Heidelberg, 2010.

[127] I. Doghri, *Mechanics of deformable solids: linear, nonlinear, analytical and computational aspects*. Springer, 2000.

[128] G. A. Holzapfel, *Nonlinear solid mechanics: a continuum approach for engineering*. John Wiley & Sons Ltd., 2000.

[129] 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.

[130] T. Belytschko, W. Liu, and B. Moran, *Nonlinear Finite Elements for Continua and Structures*. Wiley, 2000.

[131] O. C. Zienkiewicz and R. L. Taylor, *The Finite Element Method, Vol. 1, 5th Edition*. Butterworth-Heinemann, 2000.

[132] I. Huněk, "On a penalty formulation for contact-impact problems," *Computers & structures*, vol. 48, no. 2, pp. 193–203, 1993.

[133] P. Wriggers, *Computational Contact Mechanics*. Springer Berlin Heidelberg.

[134] D. G. Luenberger, *Linear and nonlinear programming*. Springer, 2003.

[135] B. Nour-Omid and P. Wriggers, "A note on the optimum choice for penalty parameters," *Communications in applied numerical methods*, vol. 3, no. 6, pp. 581–585, 1987.

[136] R. Kulak, "Adaptive contact elements for three-dimensional explicit transient analysis," *Computer methods in applied mechanics and engineering*, vol. 72, no. 2, pp. 125–151, 1989.

[137] I. Babuška and W. C. Rheinboldt, "A-posteriori error estimates for the finite element method," *International Journal for Numerical Methods in Engineering*, vol. 12, no. 10, pp. 1597–1615, 1978.

[138] O. C. Zienkiewicz and J. Z. Zhu, "A simple error estimator and adaptive procedure for practical engineerng analysis," *International Journal for Numerical Methods in Engineering*, vol. 24, no. 2, pp. 337–357, 1987.

[139] M. Molinari, S. Cox, B. Blott, and G. J. Daniell, "Adaptive mesh refinement techniques for electrical impedance tomography," *Physiological Measurement*, vol. 22, no. 1, p. 91, 2001.

[140] M. F. Cohen and J. R. Wallace, *Radiosity and realistic image synthesis*. Access Online via Elsevier, 1993.

[141] O. C. Zienkiewicz, D. S. Gago, and D. W. Kelly, "The hierarchical concept in finite element analysis," *Computers & Structures*, vol. 16, no. 1, pp. 53–65, 1983.

[142] J. Zhu and O. Zienkiewicz, "Adaptive techniques in the finite element method," *Communications in applied numerical methods*, vol. 4, no. 2, pp. 197–204, 1988.

[143] B. G. Baumgart, "Winged edge polyhedron representation," DTIC Document, Tech. Rep., 1972.

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

[145] M. A. Crisfield, *Non-Linear Finite Element Analysis of Solids and Structures*. John Wiley & Sons, 1996.

[146] N. I. Gould, J. A. Scott, and Y. Hu, "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 10, 2007.

[147] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA Technical Report, NVR-2011-001, Tech. Rep., 2011.

[148] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[149] H. A. Van der Vorst, "Parallel iterative solution methods for linear systems arising from discretized PDE's," *Special Course on Parallel Computing in CFD*, 1995.

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

[151] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012.

[152] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[153] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[154] "Cuda C Programming Guide," *NVIDIA Corporation, Oct.*, 2012.

[155] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[156] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Newnes, 2012.

[157] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, 2008, pp. 1–11.

[158] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU kernels for dense linear algebra," in *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2011, pp. 83–92.

[159] *CUDA CUBLAS Library*, nVidia Corporation, 2013.

[160] V. Volkov, "Better performance at lower occupancy," in *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010.

[161] *CUSPARSE Library*, nVidia Corporation, Oct. 2012.

[162] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451–460, 2009.

[163] C. Cecka, A. Lew, and E. Darve, "Application of assembly of finite element methods on graphics processors for real-time elastodynamics," in *GPU Computing Gems 3*, July 2011, pp. 187–205.

[164] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems Jade Edition*, p. 359, 2011.

[165] Stomach 3dxtras. [Online]. Available: http://www.3dxtras.com/3dxtras-free-3d-models-details.asp?prodid=7994

[166] Q. Fang and D. A. Boas, "Tetrahedral mesh generation from volumetric binary and grayscale images," in *Biomedical Imaging: From Nano to Macro, 2009. ISBI'09. IEEE International Symposium on*. IEEE, 2009, pp. 1142–1145.

[167] N. Bell and M. Garland, "CUSP: Generic parallel algorithms for sparse matrix and graph computations," 2012. [Online]. Available: http://cusplibrary.github.io/

[168] P. Helnwein, "Some remarks on the compressed matrix representation of symmetric second-order and fourth-order tensors," *Computer methods in applied mechanics and engineering*, vol. 190, no. 22, pp. 2753–2770, 2001.

[169] G.-R. Liu, *Smoothed finite element methods*. CRC Press, 2010.

[170] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *Micro, IEEE*, vol. 31, no. 2, pp. 50–59, 2011.