Towards Real-Time Simulation of a Finite Element Generic Lumbar Spine Model

by

Nathanial Kenneth Shiyoso Maeda

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Mechanical Engineering
University of Alberta

Abstract

Real-time simulation of biomechanical models has provided exciting potential for improving the outcomes of some medical interventions by allowing clinicians to visualize the displacements and strains of biomechanical tissues in real-time during the intervention. Through real-time visualization, the clinician could plan the intervention quicker and adjust the intervention as they operate in response to the displacements and strains, in addition to improving the training of clinicians by using realistic biomechanics in simulation scenarios. Although investigated in other areas of medicine, no study has attempted to create a real-time simulation of the spine for implementation into spinal interventions, despite the biomechanical nature of many spinal conditions and their treatments. One significant barrier for typical FE spine models is that they require huge amounts of memory and computation time. Considering the lack of developments towards real-time spine simulation and the numerous possible applications, initial work should focus on development of a generic lumbar modelling methodology from which application-specific models may be derived. Also, given the difficulty of FE contact formulations in other real-time biomechanical simulations, facet contact is a complex and difficult problem, and thus, it requires a separate investigation in itself. Therefore, the proposed thesis aims to initiate and develop improvements to clinical applicability of FE spine models by increasing computation speed close to real-time rates. To achieve this primary objective, the research was broken into three studies: create and validate a FE lumbar spine model for gross physiologic movements, to ensure generalization of the current work, using simpler element types and materials than conventional models (Study 1); develop real-time FE techniques for spine models using graphic processing unit (GPU) in conjunction with the CUDA language (Study 2); and apply the real-time FE techniques to the

FE lumbar spine model without facet contact (Study 3). In Study 1, a proposed FE lumbar spine model was created using conventional methodologies but comprised of purely tetrahedral elements and relatively stable material properties. In comparison to a conventional FE lumbar spine model using ANSYS (a conventional FE program), the proposed model exhibited similar accuracy and improved parallel computation capability with approximately 1.6X faster speeds for physiological movements. Then, in Study 2, a custom CUDA program and a simple cube model, of a similar size and material to the proposed spine model, were generated to develop and evaluate numerous parallel real-time FE techniques. In comparison with ANSYS, the CUDA program demonstrated a computation speed-up of approximately 3-4X with similar accuracy. Lastly, in Study 3, the proposed model from Study 1 (but without contact conditions) was implemented into the CUDA program from Study 2 (now the CUDA model), in addition to adding a novel composite element for the annulus fibrosus. In comparison with the conventional ANSYS model, the CUDA model demonstrated similar accuracy with approximately 20.9X speed-up versus the conventional model, in which the CUDA model's computation time was approximately 12 seconds for flexion and lateral bending. Although not real-time, this result represents possibilities toward creating application-specific spine models that may be implemented into clinical scenarios. Still, further work is necessary to reach that long-term real-time goal, including the development of real-time FE facet contact. Future research should also focus on improving the linear solver through efficient GPU implementation and testing the CUDA program on better GPUs with more cores. Altogether, the proposed thesis demonstrated significant advancements towards improving the computation speed, and thus real-time clinical use, of FE spine models.

Preface

The work in this thesis was arranged into five Chapters. In Chapter 1, we present the thesis topic, the motivation driving the thesis work, a literature review, and thesis objectives. Chapters 2 through 4 are each under review or in preparation for submissions as journal articles. In Chapter 5, an overall discussion of the work including conclusions of the thesis and areas of future work are presented.

Chapters 2 and 3 are under review at and Chapter 4 is in preparation for submission to International Journal for Numerical Methods in Biomedical Engineering as the target journal. In Chapter 2, we present the development of a novel lumbar spine model for fast parallel computation, while in Chapter 3, the development of real-time finite element techniques for spine models is described. The implementation of the real-time techniques from Chapter 3 into the novel spine model from Chapter 2 is presented in Chapter 4. Therefore, the publication plan involves submission of Chapters 2 and 3 first, then upon acceptance, Chapter 4 will be submitted. The author's contribution to each Chapter includes: model development and testing, establishment of the mathematical formulations, program development and testing, analysis and interpretation of numerical results, and manuscript preparation.

Acknowledgements

First of all, I would like to express my sincerest gratitude and thanks to my excellent supervisor, Dr. Jason Carey. In addition to his unending guidance, I greatly appreciate the many opportunities provided unto me by Dr. Carey, including and especially related to my research and teaching pursuits throughout both my undergraduate and graduate degrees (and beyond). I am forever grateful for his exceptional mentorship towards my professional development. If it were not for Dr. Carey's encouragement, I would not have found my passion for academia.

Secondly, I would like to convey my thanks to Dr. Pierre Boulanger. His passion for advancing technology was inspirational in my research pursuits. Also, his technical expertise and support were greatly helpful towards developing my programming and technical skills.

Thirdly, I would like to show my appreciation to Dr. Derek Emery. Both as a committee member and mentor, Dr. Emery has shown steadfast support for my work including helpful suggestions and confidence, care for my well-being, and necessary resources.

Fourthly, I would like to thank my numerous lab mates over the years, of which there are too many to name. I feel very fortunate to have worked with the exceptional individuals of the Carey lab group; they made my time here invaluable in terms of personal development, enthusiasm towards research and teaching, and support for each other. I greatly appreciate all the experience and mentorship that I gained thanks to the Carey lab group.

Last, but certainly not least, I would like to thank my family and friends for their encouragement throughout my degree. In particular, I recognize the unconditional support of

my parents. And especially, I appreciate the unconditional support and endless patience of my wife – whom I love more than anything in the world, including this thesis.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AIS	Adolescent Idiopathic Scoliosis
SMT	Spinal Manipulation Therapy
IVD	InterVertebral Disc
GPU	Graphic Processing Unit
FE	Finite Element
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
UL	Updated Lagrangian
TL	Total Lagrangian
CPU	Central Processing Unit
RAM	Random Access Memory
SM	Streaming Multiprocessor
DRAM	Dynamic Random-Access Memory
TLED	Total Lagrangian Explicit Dynamics
SPH	Smoothed Particle Hydrodynamics
PCG	Preconditioned Conjugate Gradient Iterative sparse solver
DOFs	Degrees of Freedom
u/p	displacement-pressure
RoM	Range-of-Motion
Itr	Iterations
ULJ	Updated Lagrangian Jaumann

csr	compressed sparse row
LLT	Symmetric Cholesky factorization direct sparse solver
LDLT	Non-symmetric Cholesky factorization direct sparse solver
SQR	Symmetric QR direct sparse solver
NQR	Non-symmetric QR direct sparse solver
EbE	Element-by-Element
MKL	Math Kernel Library
KB	kiloBytes
mm	millimetres
MPa	MegaPascals
N	Newtons
s	seconds

List of Symbols

t	current configuration (time)
$t + \Delta t$	next configuration (time)
x, y, z	nodal locations and global directions
u, v, w	nodal displacement and displacement directions
∂	partial derivative
δ, Δ	variation and increment
V, A, L	volume, area, and length of an element
S	2nd Piola-Kirchhoff stress
ϵ, e, η	Green-Lagrange strain, linear and nonlinear components
$C_{ijkl}, \boldsymbol{C}$	material property tensor and matrix
$\boldsymbol{K}, \boldsymbol{K_L}, \boldsymbol{K_N}$	stiffness matrix, linear and nonlinear components
$\boldsymbol{F}$	internal force vector
$\boldsymbol{R}$	external force vector
$\boldsymbol{b}$	force vector residual
$\boldsymbol{B}, \boldsymbol{B_L}, \boldsymbol{B_N}$	strain displacement matrix, linear and nonlinear components
m, c, s	mass, damping coefficient, and spring length displacement
E, G, κ	Young's modulus, shear modulus, and bulk modulus
ν	Poisson's ratio
C_{10}, C_{01}	Mooney-Rivlin constants
N_i	shape functions
r, s, t	isoparametric parameters

c_x, c_y, c_z	cosine directions
$\boldsymbol{J}, J$	Jacobian matrix and determinant of Jacobian
W	strain energy density function
D_{ij}	right Cauchy-Green tensor
I_1, I_2, I_3	right Cauchy-Green tensor invariants
$\bar{I}_1, \bar{I}_2, \bar{I}_3$	adjusted right Cauchy-Green tensor invariants
$\delta_{ij}, \hat{\varepsilon}_{ijk}$	Kronecker delta and permutation tensor
left superscript	configuration the parameter is calculated in
left subscript	configuration the parameter is referred from
right superscript	Newton-Raphson iteration
right subscript	node number or global direction

Chapter 1 Introduction

1.1 Motivation

Real-time biomechanical simulation is a fast-growing and exciting area of research. Some algorithms have already instigated massive computation speed increases for various biomechanical applications [1]–[3]. Likewise, virtual reality simulators for training purposes exhibit promising results, in which prospective application of real-time soft tissue deformation would greatly improve the quality of virtual reality simulation [4]–[6]. Studies investigating real-time biomechanical simulation are often motivated by medical errors in surgery [7], [8] resulting from poor training or techniques [9]. These researchers share a vision of providing clinicians with the ability to visualize the results of biomechanical simulation during the procedure [1], which would allow the clinician to evaluate the deformation at each stage and adjust their intervention as issues arise during the procedure. Considerable work has been conducted for real-time simulation of numerous biomechanical scenarios, including estimation of brain shift during image-guided neurosurgery [10] and hepatectomy [11]. Other real-time simulation work has developed general computational techniques for usage in applications involving biological soft-tissue deformation such as advanced surgical simulations [2]. Yet, current research into real-time simulation involving spine intervention scenarios is lacking.

Medical and therapeutic procedures involving the spine are among the most common interventions. Some of these procedures attempt to remedy the growing epidemic of mechanical back ailments. For example, back pain is one of the most common medical ailments in developed populations. Up to 70% of people in developed countries suffer from back pain at some point in their lives [12], in which quality of life is affected via inhibited

activity and decreased worker productivity. Current treatments for back pain include exercise, spinal manipulation therapy (SMT), and surgery. SMT is a non-invasive treatment involving force application to the patient's spinal vertebrae, and although it is deemed relatively safe, SMT sometimes causes adverse events at a rate of up to 30% [13], [14] ranging from headaches and nausea to paralysis and death in extreme cases (for neck manipulations in particular) [13]. As another example of mechanical back ailments, adolescent idiopathic scoliosis (AIS) is a three-dimensional spinal deformity characterized by an unusual spinal curve that affects 2-3% of children [15], in which the etiology of AIS is not well understood. Apart from cosmetic concerns, it can have debilitating muscular and cardiovascular effects especially in severe cases. AIS is often treated using a scoliotic brace to straighten the spine, but brace success is greatly dependent on the orthotist's training and skill [16]. Generally, if fails for AIS or for severe cases of back pain (with a clear pathological cause), surgery is required to correct the spinal abnormalities. Yet, spinal surgery carries considerable cost and risk, in which adverse events occur at a rate of 2.4 to 7% [17], [18]. These examples, among others, drive the need for improved treatment of mechanical back ailments.

Given the mechanical nature of medical interventions for back pain and scoliosis, any improved treatment methods should involve spine biomechanics. For example, prevention of adverse events during SMT relies directly upon how the spinal tissues deform in response to applied forces. The clinician could avoid unintended movements (i.e. potentially damaging) of the spine if they could visualize the spine's deformation as they apply the force (i.e. in real-time). As another example, spine straightness during bracing and surgical treatments of AIS also depends upon the spine's biomechanical response during the intervention. The clinician has no visual tools to quickly evaluate the straightness of the spine and adjust applied forces

to ensure spine straightness while forming the brace or implementing the surgical rods. Clearly, effective real-time spine biomechanical feedback (in the form of displacements and strains) to the clinician is lacking. Currently, few technological aids exist for clinicians because the development of patient-specific technologies is very difficult. Clinically exciting ideas developed through extensive research, such as spine biomechanical models, are often not clinically integrated. Reasons for the lack of translation to practice for spine models are multifactorial, consisting of three main barriers: development and validation of patient-specific models; virtual interaction between the clinician and computational model; and real-time computation speed at a reasonable cost. Since none of these barriers have yet been appropriately addressed in the literature for general spine applications, the proposed thesis focuses on undertaking only the real-time simulation barrier. Once completed, the other two barriers can be more readily addressed, similar to other real-time biomedical simulation developments [10], [11].

Considering the various potential applications that would benefit from a real-time spine model plus the complexities of real-time simulation, real-time simulations techniques should be developed using a general model first before applying to specific applications. Once developed for generic lumbar modelling methodologies, the real-time techniques can be integrated into application-specific spine models. For example, a group of spine researchers investigating interventions to various spinal pathologies [19], [20] are interested in the exciting clinical potential that a real-time model (or at least a fast model) would bring to each of their clinical applications. In application, each research group would generate another specific model using the proposed generic methodology and validate each specific model to fit each of their particular needs. For the purposes of this thesis, a “generic model” refers to a model developed

using generic metholodologies which are common within the generation of many application-specific models, and that the model was validated for displacements against gross physiologic movements including flexion, extension, lateral bending, axial rotation, and compression.

Clinically-usable virtual simulations of patient-specific spine models offer great potential for improving clinician training and treatment outcomes. Although far from realization, application of a real-time virtual spine model into practice could potentially reduce the occurrence of adverse events by allowing the clinician to evaluate the spine's displacements and strains during the procedure, then adjust their technique to account for any unintended movement may have that occurred. Yet, in order to approach that far-away point, real-time simulation techniques specifically designed for a generic lumbar spine model must be developed, to ensure that the proposed results and methodologies apply to the greatest number of possible application-specific spine models, before any specific applications can be considered. Hence, clinical integration of biomechanical spine simulations is the long-term vision of the current work that would provide significant impact on public health. Also, addressing the real-time barrier requires more scope that current thesis can provide; and therefore, the proposed thesis focuses on developing and testing real-time simulation techniques for a generic lumbar spine model as novel progress in the field of real-time spine simulation.

The proposed thesis presents foundational starting steps toward potential technological improvements to spinal interventions. As already done in other areas of medicine [11], [21], the proposed work initiates new development of real-time biomechanical simulation for spinal interventions.

1.2 Literature Review

The proposed thesis presents a multidisciplinary approach; and therefore, literature from numerous areas must be reviewed. These areas include: Lumbar Spine Finite Element Modelling (Section 1.2.1); The Finite Element Method – Total Lagrangian Formulation (Section 1.2.2); GPU Computation and Programming (Section 1.2.3); GPU Computing for Finite Element Analysis (Section 1.2.4); and Real-Time Simulation (Section 1.2.5).

1.2.1 Lumbar Spine Finite Element Modelling

Many experimental studies have investigated spine biomechanics for various purposes [22]: to improve understanding of how the spine responds to loading [23], including stability; to develop new implant designs [24]; to study the effects of degeneration [25], [26]; to analyze various physiological movements and physical activities [27], [28]; and to study the effects of various spinal treatments [20], [29]. Yet, numerous testing scenarios cannot be performed experimentally, such as testing of implant designs before production and extreme displacements for example. Acquiring *in vivo* measurement data of the spinal tissues in living participants has procedural and ethical difficulties, in addition to current technological limitations in which medically unnecessary and invasive procedures are sometimes required to place the sensors. To circumvent these difficulties, researchers have developed finite element (FE) models to simulate the displacements and strains exhibited by the lumbar spine during biomechanical loadings. Numerous patient-specific models have been validated against experimental data and effectively predicted the response of individual lumbar spines to loading [30]–[38]. Subsequently, FE spine modelling has become a powerful tool for visual simulation, and it has greatly improved spine biomechanics knowledge. Early models established that spine stiffness increases with increased loading [39] and the primary load-bearing components

are the IVD and facet joints [40]. Although the vertebral body carries most axial load in the spine, particularly during standing posture, the posterior elements, especially the facets, exhibit a great influence on loads in lateral-medial and posterior-anterior directions [40]–[42]. Despite this, some studies have successfully developed models without contact for their specific applications [43]–[45]. As the spine modelling field developed, some studies determined that the ligaments play a significant role in spine biomechanics. El-Rich *et al*. [46] proved that forces in the ligaments are important during fast movements, and that fracture can occur first in the pedicles during fast movements. Although significant progress has been made using FE spine models, validation of FE biomechanical models against *in vitro* and *in vivo* data remains a challenge for current and future investigations of FE spine models in clinical situations. Dreischarf *et al.* [30] studied a number of full lumbar models and determined that their median response most accurately predicted *in vitro* and *in vivo* spine response to gross physiologic loadings, yet further study is required with respect to validation depending on the specific clinical scenario. Regardless, later models built upon the early models by expanding their methodologies to include more complexity, such as viscoelasticity and muscles [46], [47] depending on their application. Although they are more complex, many current FE models still refer to the methodologies of early spine modelling studies.

Given these previous developments, much of the progress in the modelling of gross lumbar spine biomechanics can be attributed to the generation of lumbar spine models using generic methodologies that explored spine displacements and strains in response to physiologic movements (i.e. flexion, extension, lateral bending, axial rotation, and compression) [30], [44], [46], [48]–[50]. Many application-driven studies have relied upon the generic methodologies used to build these lumbar spine models for generation of their application-specific models.

For example, Schmidt et al [31], [49], [51] built their model for testing lumbar disc arthroplasty from numerous previous sources [38], [50], [52] especially their material properties, annulus fibre distribution, vertebral modelling, and element formulations. Other examples, among many more, include Li et al [53] from similar sources [49], [50], [54] for application to intervertebral cages and scoliosis models [55], [56] acquiring their facet contact and some material properties from previous studies [50], [57].

Although researchers have validated their models against experimental data for certain applications and made significant progress in characterizing the lumbar spine's response to applied loading, their models exhibit some limitations, which reveal potential barriers for directly integrating FE models into clinical scenarios or clinician training, depending on the application. Addressing any of their limitations will not improve applicability of FE lumbar models into their respective real-time clinical scenarios (including clinician training), unless they can achieve real-time computation speeds. Nonetheless, one example of an FE spine model used clinically to directly improve clinical outcomes is scoliosis bracing [58]. Computational models of scoliotic spines have given significant insights into the biomechanics of AIS and bracing techniques [59]. Lately, computer aided drafting software has been used in conjunction with FE modelling software to improve brace design methods [58]. Yet again, this novel method is currently too time-consuming for the orthotist to use practically in the clinic.

As such, a significant restriction of current FE lumbar spine models is their computation speed, in which a single analysis requires substantial amounts of time (i.e. hours to weeks). Although many FE spine models have been validated against experimental data and some have been

developed for use in other clinical situations, such as torso brace design for scoliosis treatment as described above [58], none have yet been integrated into other real-time clinical scenarios such as SMT or spinal surgery for example. Significant limitations, especially computation speed, prevent the usefulness of FE models in such clinical situations, where they could provide the real-time biomechanical understanding to improve spinal interventions. Few researchers to date have attempted to create a clinically-usable real-time FE spine model [60]–[62] that exhibits graphics update speeds nearing real-time (approximately 30 hertz) [2]. Therefore, although spine biomechanics and back pain have been studied extensively both experimentally and numerically, there still exists a large gap between modelling and real-time clinical application.

To ensure the generality of the current work towards prospective lumbar spine models for clinical applications, a generic lumbar spine model is targeted in order to develop real-time simulation techniques that may allow many specific spine models to potentially be used in clinical applications. All aspects of FE spine model generation must be examined from various models in literature to develop an effective and real-time generic lumbar spine modelling methodology for clinical use. The general procedure of spine model development may be found in Figure 1-1.

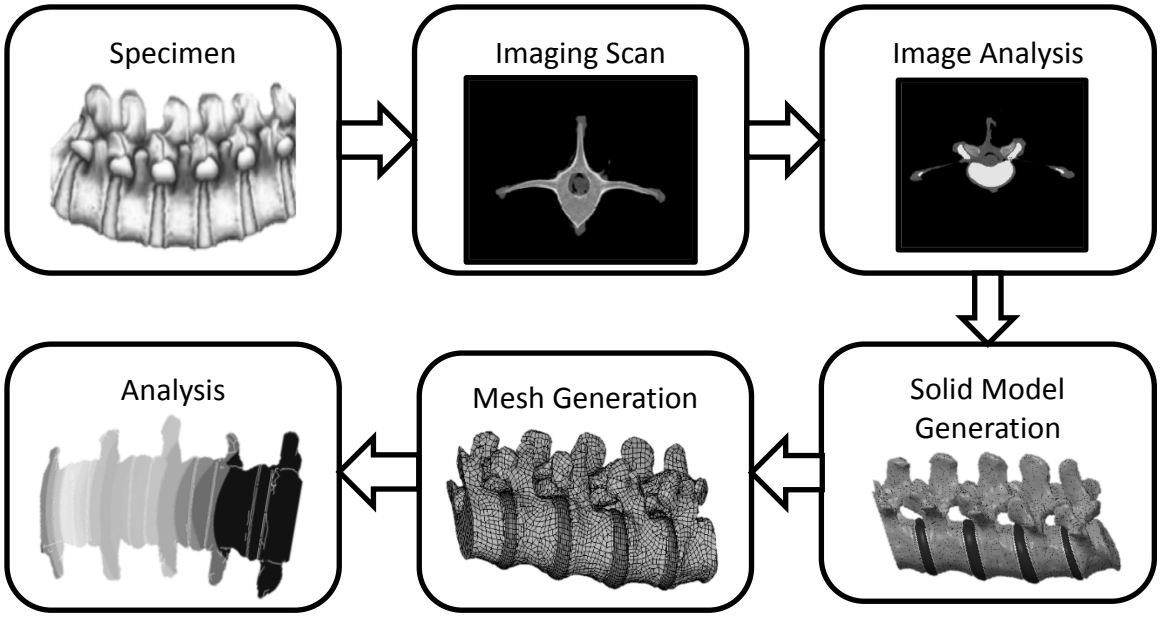

Figure 1-1: Development of an FE lumbar spine model. Specimen image acquired from [63]. The CT images depict a pig spine vertebra, while the model images show a human lumbar spine.

The FE lumbar models described in the ensuing paragraph are mostly from studies that investigated and were validated against general physiologic movements with applications to back pain, spine degeneration, or spinal implants. Kinematic models were not included here as they are applicable to only a small set of applications compared to FE ligamentous lumbar models. The generic modelling methodologies presented by these FE lumbar models are typically used as a basis to build application-specific models, as described in previously.

Most generic FE lumbar spine modelling methodologies involve generation of seven distinct structural components: cancellous bone, cortical bone, posterior elements, annulus fibrosus, nucleus pulposus, cartilaginous endplate, and seven ligaments. The cancellous bone encapsulated by the cortical bone comprised the vertebral body, and the annulus fibrosus

consisted of a solid matrix embedded with fibres. Some models included a bony endplate on the vertebral body as well [51], [64]. Typically, participants or cadavers underwent a computed tomography (CT) or magnetic resonance imaging (MRI) scan of their lumbar spine [65], and the resulting image stacks from the scans were analyzed using image analysis software (such as Simpleware or Mimics) to create three-dimensional solid (virtual) models of the vertebrae. During image analysis, the images were filtered, and the model was smoothed to ensure good quality FE elements during mesh generation. After generating the vertebrae, cartilage endplates of 0.5 to 1.0 mm thickness (plus bony endplates for some models) were created on the inferior and superior surfaces of the vertebral bodies [66], [67]. Then, the IVDs filled in the spaces between the vertebral bodies, where the nucleus pulposus was surrounded by multiple layers of annulus fibrosus. For some models, the nucleus pulposus was centered within the intervertebral disc [41], [68], but for other models, its center was moved some small distance (usually 3.5 mm) dorsally [44], [51]. For mesh generation, all models employed entirely hexahedral elements for the annulus fibrosus with mostly hexahedral elements mixed with some tetrahedral elements for the nucleus pulposus, endplates, vertebral bodies, and posterior elements [30], [46]. Most models used shell elements for the cortical bone [46] while others used solid hexahedral/tetrahedral elements [51]. The ligaments and fibers were usually generated with one-dimensional tension-only spring elements [30], while some recent models generated the ligaments using two-dimensional membrane elements [31], [46]. Since the critical biomechanical components of the spine are the IVD and facet joints [40], almost every model paid particular attention to these areas. In most models, the annulus fibrosus was modelled like a composite material with uniaxial reinforcement: spring elements for the fibers connected the corner nodes of the hexahedral elements that comprised the annulus matrix [30].

Alternatively, some models overlaid the hexahedral matrix mesh with fiber membrane elements, while others defined the combined annulus matrix and fibers with composite elements which were formulated using composite material continuum theory [69]. Effectively, modelling the fibers as separate elements results in the stiffness contribution of the fibers being superimposed on the stiffness of the matrix. With regards to fiber angles within the annulus matrix, some models implemented the fibers at approximately 30° [39], [68] while others varied the fibers ventrally from approximately 24° at the midplane to 46° at the dorsal side [51]. The stiffness contribution of the fibers also typically varied from stiffer in the outer layer to less stiff in the inner layer [41]. Contact modelling within the facet joints revealed greatest discrepancies between models. Some models used soft contact, although the normal contact stiffness varied between models, while others used hard contact [30]. A facet cartilage layer was sometimes included in only some models [66], [70]. Also, the initial gap between facet surfaces varied greatly between models since it is highly dependent on vertebral geometry, which was acquired from different participants. All models regarded the facet joints as frictionless [30]. Despite the importance of the facet joints, some models did not include contact formulations [43]–[45], but each of their specific applications did not require contact conditions.

Many studies acquired the material properties for each specific spinal structure in their lumbar models from different sources. Cancellous and cortical bones were usually defined as linear orthotropic [31], [35], but sometimes as linear isotropic in simpler models [33], [34], [36] or elasto-plastic in more complex models [46], [71]. A couple studies even generated bone stiffness values from CT density values (i.e. Hounsfield units) [72]. The posterior elements were typically defined as linear isotropic [30]. Yet, one model characterized each vertebra as

a rigid body connected by a deformable beam element at the pedicles [48], in which the beam element was comprised of the same material properties as the posterior elements. Cartilage endplates consisted of linear isotropic material with similar properties across most models [39], [51], and in models that include bony endplates, the bony endplate material was linear isotropic as well with stiffness similar to cortical bone [51], [73]. Nucleus pulposus material model differed widely from model to model: some were linear elastic [32], some were Mooney-Rivlin [51], and others were incompressible using hydrostatic elements [30]. All models of the nucleus pulposus were either nearly- or fully-incompressible [30]. Similarly, some spine models described the annulus fibrosus matrix as linear elastic [39] and others as Mooney-Rivlin [31], although the material properties were similar between models with a moderately high Poisson's ratio [30]. With regards to the annulus fibers, some studies specified a linear elastic stress-strain curve [44], [74], [75], while most models referred to the same curve defined by Shirazi-Adl [39]. Lastly, ligament stress-strain curves were usually acquired from the same two studies, one by Pintar [76] and the other one by Shirazi-Adl [39], although some studies approximated these curves as linear or multilinear elastic. However, regardless of element types or material models, all the successful models from literature were well-validated against experimental data, either collected in-house on the same cadaveric specimen as the model's geometry or from previous studies in literature. Especially, Dreischarf *et al.* [30] investigated numerous well-validated FE spine models from other researchers and compared them to *in vitro* and *in vivo* data [77], [78]. They not only discovered that the median value of the FE models provided a good approximation to experimental data, but also provided a means for which to validate future FE models. If a proposed FE model's results lie within the range of previously well-validated models presented by Dreischarf *et al*., then that model is effectively

valid for spine biomechanical response. Still, all FE spine models from the literature are complex and require significant amounts of computation time. Based on previous methodologies, and through comparison to previous studies, an accurate FE spine model may be developed for implementation into real-time clinical scenarios.

Recent improvements in computing hardware, specifically GPUs, and FE formulations exhibit great promise towards creating real-time spine simulations. NVIDIA has developed a C programming interface called CUDA (Compute Unified Device Architecture) [79], allowing programmers to easily parallelize their applications onto GPUs. Likewise, some studies have developed novel methods incorporating CUDA to increase computation speeds for nonlinear simulations of human tissues. Using these methods and through the development of novel techniques, the proposed research focuses upon improving the computation time of FE lumbar spine models to approach real-time speeds. Still, before assessing GPU programming and real-time simulation, the mathematical formulations must be derived and analyzed.

1.2.2 The Finite Element Method – Total Lagrangian Formulation

Finite element method has been used since the early 1960's as an effective numerical approximation of the equations of motion. Early work in nonlinear structural finite element analysis focused on two different implementations of the continuum mechanics incremental equations of motion: the Updated Lagrangian (UL) formulation and the Total Lagrangian (TL) formulation [80]. The primary conceptual difference between the UL and TL formulations is the equilibrium configuration that the stresses and strains refer to. During Newton-Raphson iterations, the displacement derivatives may be referred to any configuration for which equilibrium has already been calculated, see Figure 1-2.

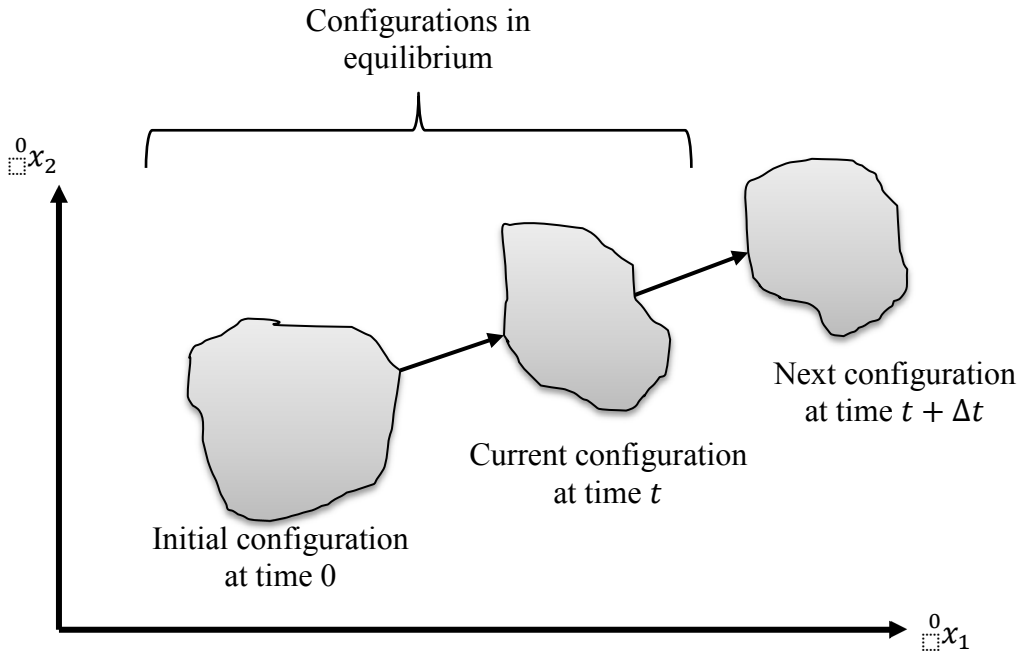

Figure 1-2: Configurations considered for the UL and TL formulations [80].

For the TL formulation, the initial configuration (usually unloaded) is the referenced configuration, while for the UL formulation, the current configuration (i.e. the most recently calculated) is the referenced configuration. Effectively, the UL formulation requires recalculation of the current configuration at each iteration but less memory to store the referenced configuration. On the other hand, the TL formulation requires more memory to hold the initial configuration throughout the analysis but fewer calculations per iteration. Hence, early finite element programs employed the UL formulation since memory was costly in earlier computers. The TL formulation has gained recent attention since computer memory is now less costly. Each configuration has advantages and disadvantages over the other, depending on the target application. Yet, both formulations include large deformation, large rotation, and large strain effects. The ensuing mathematical derivations follow Bathe's book [80] (please

see the reference for further details). Considering GPU implementation and corresponding to advancements in the field of real-time simulation, the proposed thesis uses the TL formulation. The following derivation describes how deformable models are represented within the TL framework, in addition to presenting the resulting FE matrices.

Based on the principle of virtual displacements (including the principle of virtual work), the governing equation of motion for nonlinear finite element analysis in the TL framework is seen in Equation (1-1).

$$\int_{{}^0V} {}^{t+\Delta t}_{0}S_{ij}\delta\, {}^{t+\Delta t}_{0}\epsilon_{ij} d\ {}^0V = \delta u_i\, {}^{t+\Delta t}R_i \qquad (1-1)$$

where ${}^{t+\Delta t}_{0}S_{ij}$ refers to components of the 2nd Piola Kirchhoff (PK2) stress tensor in the next configuration with reference to the initial configuration, ${}^{t+\Delta t}_{0}\epsilon_{ij}$ refers to components of the Green-Lagrange (GL) strain in the next configuration with reference to the initial configuration, ${}^0V$ is the volume in the initial configuration, u_i is the components of displacement, ${}^{t+\Delta t}R_i$ is the components of the external load vector, and δ refers to the virtual displacement. For clarification, the left subscript for each parameter corresponds to the referred configuration while the left superscript refers to the calculated configuration.

To solve the governing equation at the next time step, the GL strain, Equation (1-2), may be broken down into components.

$${}^{t+\Delta t}_{0}\epsilon_{ij} = {}^{t+\Delta t}_{0}e_{ij} + {}^{t+\Delta t}_{0}\eta_{ij} \qquad (1-2)$$

where ${}^{t+\Delta t}_{0}e_{ij}$ is the linear component and ${}^{t+\Delta t}_{0}\eta_{ij}$ is the nonlinear component.

The components of the GL strain may be inserted into the governing equation, see Equation (1-3).

$$\int_{{}_0V} {}^{t+\Delta t}_{0}S_{ij}\delta\left({}^{t+\Delta t}_{0}e_{ij} + {}^{t+\Delta t}_{0}\eta_{ij}\right) d\ {}^{0}V = \delta u_i\ {}^{t+\Delta t}R_i$$

$$\Rightarrow \int_{{}_0V} {}^{t+\Delta t}_{0}S_{ij}\delta\, {}^{t+\Delta t}_{0}e_{ij} d\ {}^{0}V + \int_{{}_0V} {}^{t+\Delta t}_{0}S_{ij}\delta\, {}^{t+\Delta t}_{0}\eta_{ij} d\ {}^{0}V = \delta u_i\ {}^{t+\Delta t}R_i \qquad (1-3)$$

Considering that the stresses and strains at the $t + \Delta t$ configuration are unknown, they must be decomposed into incremental components by:

$${}^{t+\Delta t}_{0}S_{ij} = {}^{t}_{0}S_{ij} + {}_0S_{ij} \qquad (1-4)$$

$${}^{t+\Delta t}_{0}e_{ij} = {}^{t}_{0}e_{ij} + {}_0e_{ij} \qquad (1-5)$$

$${}^{t+\Delta t}_{0}\eta_{ij} = {}^{t}_{0}\eta_{ij} + {}_0\eta_{ij} \qquad (1-6)$$

To apply the strain components to Equation (1-3), the variation must be applied by:

$$\delta\, {}^{t+\Delta t}_{0}e_{ij} = \delta\ {}_0e_{ij} \qquad (1-7)$$

$$\delta\, {}^{t+\Delta t}_{0}\eta_{ij} = \delta\ {}_0\eta_{ij} \qquad (1-8)$$

Note that $\delta\, {}^{t}_{0}e_{ij} = \delta\, {}^{t}_{0}\eta_{ij} = 0$ since the variation is taken at the $t + \Delta t$ configuration.

Inserting the strain decompositions into Equation (1-3), one can obtain:

$$\int_{{}_0V} {}^{t}_{0}S_{ij}\delta\ {}_0e_{ij} d\ {}^{0}V + \int_{{}_0V} {}_0S_{ij}\delta\ {}_0e_{ij} d\ {}^{0}V + \int_{{}_0V} {}^{t}_{0}S_{ij}\delta\ {}_0\eta_{ij} d\ {}^{0}V + \int_{{}_0V} {}_0S_{ij}\delta\ {}_0\eta_{ij} d\ {}^{0}V = \delta u_i\ {}^{t+\Delta t}R_i \qquad (1-9)$$

So far, no assumptions or simplifications have been made to the governing equation of motion. It has only been re-written in terms of incremental decompositions. Now, each term may be analyzed to determine whether simplifications (i.e. linearizing displacements) would help improve solvability. Hence, the linearity of each strain component must be determined. Referring to Equation (1-2), the GL strain is calculated from Equation (1-10) and is equal to:

$$ {}^{t+\Delta t}_{0}\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial {}^{t+\Delta t}u_i}{\partial\ {}^{0}x_j} + \frac{\partial {}^{t+\Delta t}u_j}{\partial\ {}^{0}x_i} + \frac{\partial {}^{t+\Delta t}u_k}{\partial\ {}^{0}x_i}\frac{\partial {}^{t+\Delta t}u_k}{\partial\ {}^{0}x_j}\right) \qquad (1-10) $$

where ${}^{t+\Delta t}_{0}u_{i,j} = \frac{\partial {}^{t+\Delta t}u_i}{\partial\ {}^{0}x_j}$.

Considering that ${}^{t+\Delta t}u = {}^{t}u + u$, Equation (1-10) may be decomposed into its increments:

$$ {}^{t+\Delta t}_{0}\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial\left({}^{t}u_i + u_i\right)}{\partial\ {}^{0}x_j} + \frac{\partial\left({}^{t}u_j + u_j\right)}{\partial\ {}^{0}x_i} + \frac{\partial\left({}^{t}u_k + u_k\right)}{\partial\ {}^{0}x_i}\frac{\partial\left({}^{t}u_k + u_k\right)}{\partial\ {}^{0}x_j}\right) $$

$$ \Rightarrow {}^{t+\Delta t}_{0}\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial\ {}^{t}u_i}{\partial\ {}^{0}x_j} + \frac{\partial\ {}^{t}u_j}{\partial\ {}^{0}x_i} + \frac{\partial\ {}^{t}u_k}{\partial\ {}^{0}x_i}\frac{\partial\ {}^{t}u_k}{\partial\ {}^{0}x_j} + \frac{\partial u_i}{\partial\ {}^{0}x_j} + \frac{\partial u_j}{\partial\ {}^{0}x_i} + \frac{\partial\ {}^{t}u_k}{\partial\ {}^{0}x_i}\frac{\partial u_k}{\partial\ {}^{0}x_j} + \frac{\partial u_k}{\partial\ {}^{0}x_i}\frac{\partial\ {}^{t}u_k}{\partial\ {}^{0}x_j} + \frac{\partial u_k}{\partial\ {}^{0}x_i}\frac{\partial u_k}{\partial\ {}^{0}x_j}\right) $$

$$ = \frac{1}{2}\begin{pmatrix} {}^{t}_{0}u_{i,j} + {}^{t}_{0}u_{j,i} + {}^{t}_{0}u_{k,i}\,{}^{t}_{0}u_{k,j} \\ + {}_{0}u_{i,j} + {}_{0}u_{j,i} + {}^{t}_{0}u_{k,i}\,{}_{0}u_{k,j} + {}_{0}u_{k,i}\,{}^{t}_{0}u_{k,j} + {}_{0}u_{k,i}\,{}_{0}u_{k,j} \end{pmatrix} \qquad (1-11) $$

Equation (1-12) may be determined by combining Equations (1-2) and (1-7) to (1-8). Then, comparing Equation (1-12) to Equation (1-11), derivations for ${}_{0}e_{ij}$ and ${}_{0}\eta_{ij}$ may be found:

$$ {}^{t+\Delta t}_{0}\epsilon_{ij} = {}^{t}_{0}e_{ij} + {}^{t}_{0}\eta_{ij} + {}_{0}e_{ij} + {}_{0}\eta_{ij} \qquad (1-12) $$

$$ {}_0e_{ij} = \frac{1}{2}\left({}_0u_{i,j} + {}_0u_{j,i} + {}^t_0u_{k,i}\ {}_0u_{k,j} + {}_0u_{k,i}\,{}^t_0u_{k,j}\right) \qquad (1-13) $$

$$ {}_0\eta_{ij} = \frac{1}{2}\left({}_0u_{k,i}\ {}_0u_{k,j}\right) \qquad (1-14) $$

By applying the variation to the increment in strain components, linearity of the terms may be determined by:

$$ \delta\ {}_0e_{ij} = \frac{1}{2}\left({}_0\delta u_{i,j} + {}_0\delta u_{j,i} + {}^t_0u_{k,i}\ {}_0\delta u_{k,j} + {}_0\delta u_{k,i}\,{}^t_0u_{k,j}\right) \qquad (1-15) $$

$$ \delta\ {}_0\eta_{ij} = \frac{1}{2}\left({}_0\delta u_{k,i}\ {}_0u_{k,j} + {}_0u_{k,i}\ {}_0\delta u_{k,j}\right) \qquad (1-16) $$

Since Equations (1-15) and (1-16) are linear and ${}^t_0S_{ij}$ is a known value from the current configuration, the ${}^t_0S_{ij}\delta\ {}_0e_{ij}$ and ${}^t_0S_{ij}\delta\ {}_0\eta_{ij}$ terms are known and linear, respectively, noting that the ${}_0\delta u$ terms are the virtual displacements and will cancel out when Equation (1-9) is expanded. Given that ${}_0S_{ij}$ is a function of strain, Equation (1-9) is highly nonlinear because of the terms ${}_0S_{ij}\delta\ {}_0\eta_{ij}$ and ${}_0S_{ij}\delta\ {}_0e_{ij}$. In order to linearize these terms, a Taylor expansion may be applied where we neglect any higher order terms:

$$ {}_0S_{ij}\delta\ {}_0e_{ij} = \left(\frac{\partial\,{}^t_0S_{ij}}{\partial\,{}^t_0\epsilon_{rs}}\ {}_0e_{rs} + higher\ order\ terms\right)\delta\ {}_0e_{ij} \cong\ {}_0C_{ijrs}\ {}_0e_{rs}\delta\ {}_0e_{ij} \qquad (1-17) $$

$$ {}_0S_{ij}\delta\ {}_0\eta_{ij} = \left(\frac{\partial\,{}^t_0S_{ij}}{\partial\,{}^t_0\epsilon_{rs}}\ {}_0\eta_{rs} + higher\ order\ terms\right)\delta\ {}_0\eta_{ij} \cong 0 \qquad (1-18) $$

where ${}_0C_{ijrs} = \frac{\partial\,{}^t_0S_{ij}}{\partial\,{}^t_0\epsilon_{rs}}$ is the tangential (first order) material property matrix calculated at the current configuration.

Finally, the linearized equation of motion may be derived by inserting Equations (1-17) and (1-18) into Equation (1-9) and moving known quantities to the right-hand-side:

$$\int_{{}_0V} {}_0C_{ijrs}\ {}_0e_{rs}\delta\ {}_0e_{ij}d\ {}^0V + \int_{{}_0V} {}_0^tS_{ij}\delta\ {}_0\eta_{ij}d\ {}^0V = \delta u_i{}^{t+\Delta t}R_i - \int_{{}_0V} {}_0^tS_{ij}\delta\ {}_0e_{ij}d\ {}^0V \quad (1-19)$$

As a result of the applied approximations, the governing equation of motion exhibits linearization error:

$$Error = \delta u_i{}^{t+\Delta t}R_i - \int_{{}_0V} {}_0^{t+\Delta t}S_{ij}\delta\,{}_0^{t+\Delta t}\epsilon_{ij}d\ {}^0V \quad (1-20)$$

The error in Equation (1-20) represents the "out-of-balance virtual work" and may be decreased by iterating the linearized governing equation of motion until certain convergence measures are satisfied, such as force or displacement error. Therefore, the TL formulation equation at each iteration is equal to:

$$\int_{{}_0V} {}_0C_{ijrs}^{k-1}\Delta\ {}_0e_{rs}^k\delta\ {}_0e_{ij}d\ {}^0V + \int_{{}_0V} {}_0^{t+\Delta t}S_{ij}^{k-1}\delta\Delta\ {}_0\eta_{ij}^k d\ {}^0V$$

$$= \delta u_i{}^{t+\Delta t}R_i - \int_{{}_0V} {}_0^{t+\Delta t}S_{ij}^{k-1}\delta\,{}_0^{t+\Delta t}\epsilon_{ij}^{k-1}d\ {}^0V \quad (1-21)$$

where $\Delta\ {}_0e_{ij}^k = f\left(\Delta u_i^k\right)$ and $\Delta\ {}_0\eta_{ij}^k = g\left(\Delta u_i^k\right)$ and ${}^{t+\Delta t}u_i^k = {}^{t+\Delta t}u_i^{k-1} + \Delta u_i^k$.

Closer inspection of Equations (1-19) and (1-20) reveal that Equation (1-21) represents the Newton-Raphson iterations for solving Equation (1-20), where *k* is the current Newton-Raphson iteration. See Figure 1-3 for a depiction of the Newton-Raphson solution to nonlinear equations.

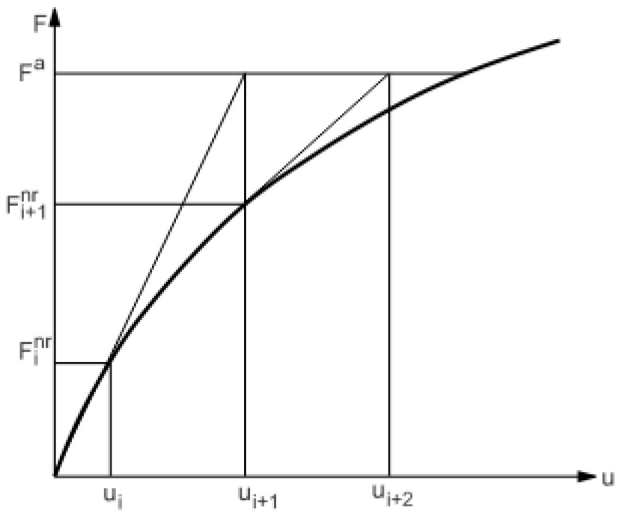

Figure 1-3: Depiction of Newton-Raphson iterations in solution to nonlinear equations.

In matrix form, Equation (1-21) becomes:

$$\left({}^{t+\Delta t}_{0}\boldsymbol{K}_{\boldsymbol{L}}^{k-1} + {}^{t+\Delta t}_{0}\boldsymbol{K}_{\boldsymbol{N}}^{k-1}\right){}^{t+\Delta t}\boldsymbol{\Delta u}^{k} = {}^{t+\Delta t}\boldsymbol{R} - {}^{t+\Delta t}_{0}\boldsymbol{F}^{k-1} \qquad (1-22)$$

where $\boldsymbol{K_L}$ is the linear stiffness matrix representing the first term of Equation (1-21), $\boldsymbol{K_N}$ is the nonlinear stiffness matrix representing the second term, $\boldsymbol{\Delta u}$ is the iteration increment in displacement, $\boldsymbol{R}$ is the external force vector, and $\boldsymbol{F}$ is the internal force vector representing the second term on the right-hand side.

Now that the general FE equation at each Newton-Raphson iteration has been derived for the TL formulation, matrices for each element must be generated. For the models used in the current study, two element types are used primarily: linear tetrahedral elements and linear tension-only spring elements. See Appendix A for the derivation of the FE matrices characterizing linear tetrahedral elements, and see Appendix B for the matrices characterizing

linear tension only spring elements. Also, see Appendix C for the derivation of the various material property matrices required for the proposed FE spine model.

1.2.3 GPU Computation and Programming

Traditionally, FE analyses were computed using central processing units (CPUs) and stored during computation on random access memory (RAM) of desktop computers (or network servers). As a result, simulations were slow, and better hardware (such as server clusters) required to speed up any simulation was expensive, yet they still did not approach real-time computational speed. On the other hand, recent advancements in GPUs are providing new possibilities for scientific computing and simulations. Driven by the gaming industry, GPU development has greatly increased computational power for certain applications at a reasonable cost compared to CPUs, see Figure 1-4 for a speed comparison between various GPUs and CPUs over recent years.

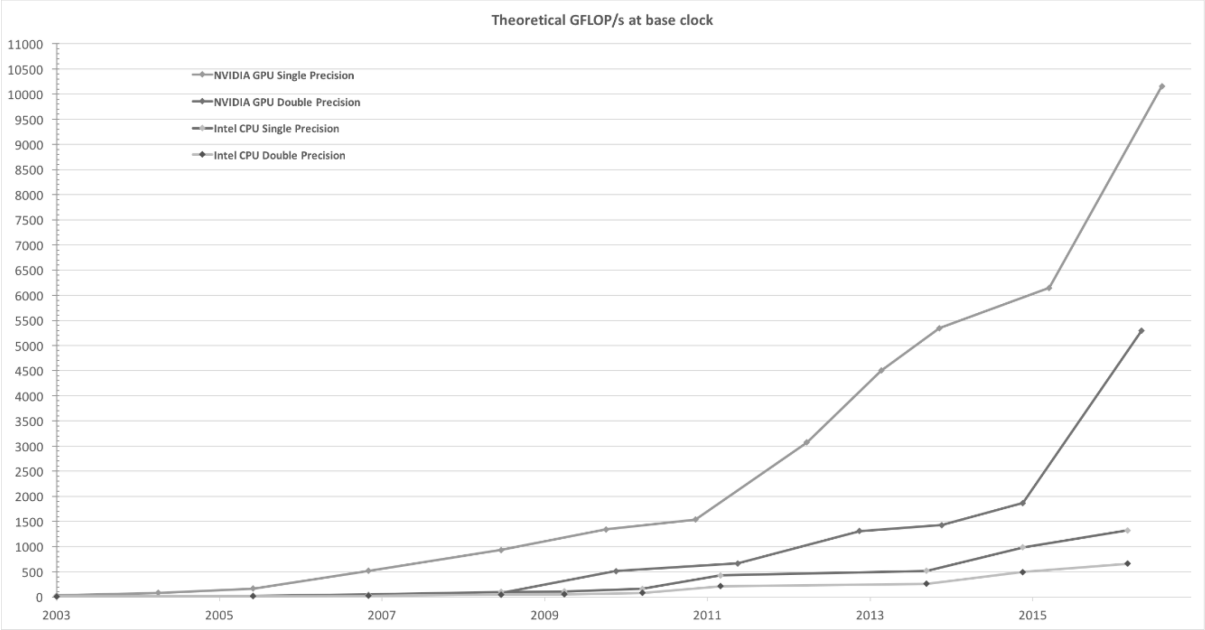

Figure 1-4: Theoretical speed-up of GPU compared to CPU. GFLOP/s stands for giga-floating operations per second. Image from [79].

GPUs perform the same computer instruction in parallel as opposed to serially as done by CPUs. In situations involving the same computations on separate memory spaces (such as FE analysis), GPUs can provide massive improvements in computation speed depending on the specific GPU used and application involved. Conversely, CPUs are significantly faster for computations on the same memory space. Using a GPU, iterations of a common computer instruction are all calculated simultaneously instead of sequentially. Hence, the power of a GPU lies in its architecture [79]: thousands of processors allow calculations to be computed in parallel, thereby decreasing total computation time for applications where identical calculations are processed with thousands of iterations. Furthermore, some of the programmable memory on the GPU resides close to the processors, which allows for very fast memory accessing times. In addition to improving GPU architecture, NVIDIA has developed

a C programming interface called CUDA, allowing programmers to readily parallelize (thus speed up) their applications. In all, GPUs provide immense potential to greatly increase computation speeds through advancements in architecture and programming interfaces, although the speed improvements depend on the parallelism of the coding application. An excellent description of GPU CUDA programming can be found on NVIDIA's website [79]. For the purposes of this thesis, a brief explanation highlighting important considerations relevant to the current studies is presented.

CUDA is a powerful programming tool for readily parallelizing computation-heavy programs. The CUDA programming model explicitly distinguishes between GPU (known as device) memory and CPU (known as host) memory. Likewise, parallel functions, or kernels, run on the device are distinctly distinguished from serial functions executed on the host. In application, the main function of a C program runs on the host and launches kernels to run on the device, see Figure 1-5.

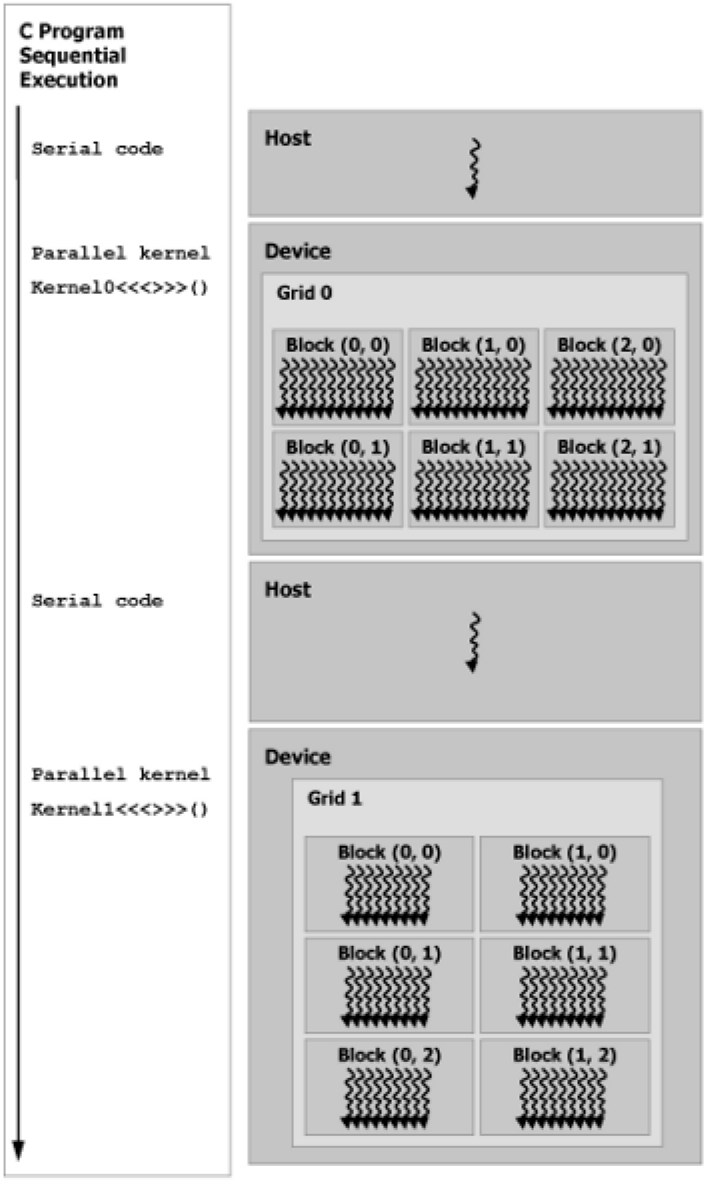

Figure 1-5: CUDA program execution depicting serial code execution on the host and parallel kernel execution on the device [79]. Note that the main function launches the kernels from the host side. In this and subsequent Figures, Block(i,j) refers to the Block identification within the grid array of two dimensions. Same convention for Thread(i,j).

CUDA's programming interface is built upon kernels, which are C functions that are called from within the main function of the program. With each kernel invocation, the number of

blocks and threads per block that the device will run must be specified. Threads represent each of the streaming multiprocessors (SMs) running the code instructions in parallel. A group of threads make up a block, where all threads in a block run simultaneously, and a group of thread blocks make up a grid, see Figure 1-6, which represents all threads running the same kernel code in parallel.

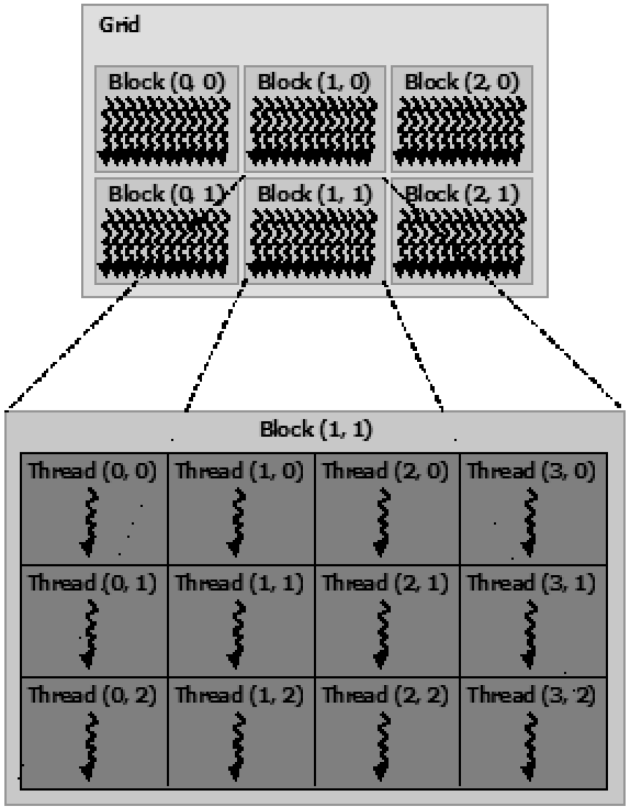

Figure 1-6: Threads, block, and grid organization [79].

Although each block is considered separate from other blocks for coding purposes, multiple blocks will run simultaneously depending on the number of SMs provided by the GPU; thus, the programmer must consider all blocks in a grid running simultaneously. Threads and blocks may be identified in up to three dimensions for convenience in specifying calculations, but for

the purposes of the current work, only one dimension was used. Within each kernel, any thread may be selected from the grid using:

$$index = blockDim.x * blockIdx.x + threadIdx.x \qquad (1-22)$$

where *index* is the resulting thread identification (ID) from the entire grid, *blockDim.x* is the block size (i.e. number of threads in the block), *blockIdx.x* is the block ID within the grid, and *threadIdx.x* is the thread ID within that block.

Kernels execute threads in equally-shaped thread blocks; hence the total number of threads run by a single kernel call is equal to the number of blocks times the number of threads per block. In conjunction with the hierarchy in kernel execution, device memory handling follows a hierarchy as well, see Figure 1-7.

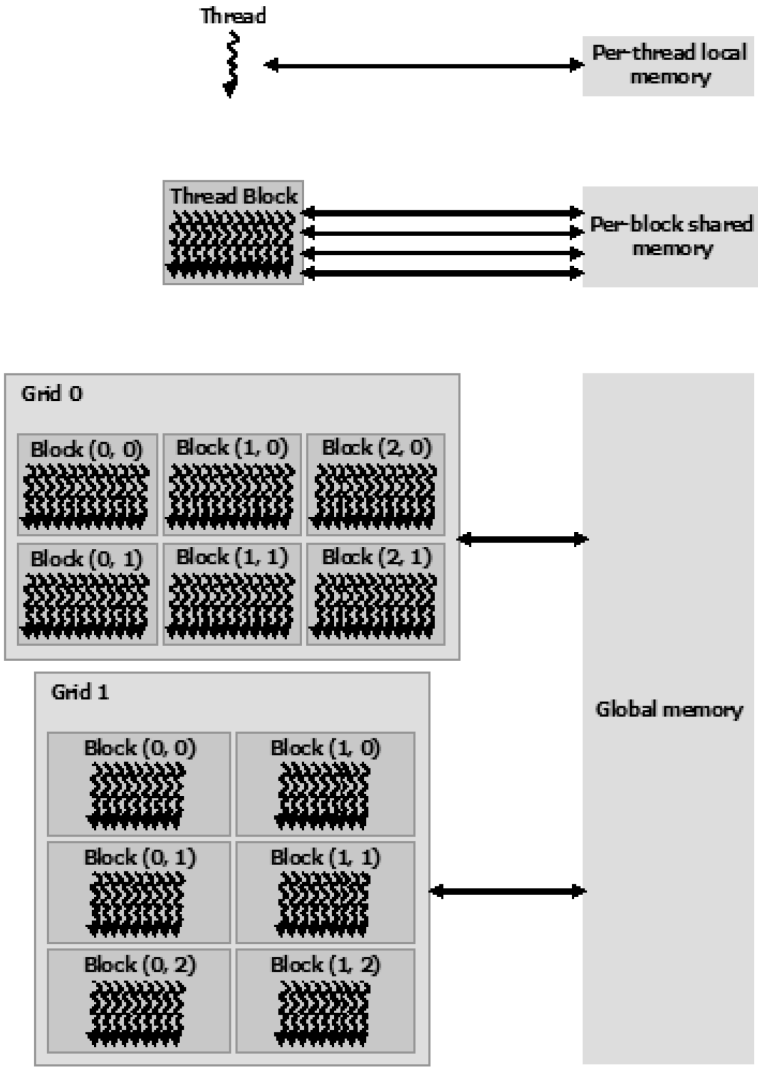

Figure 1-7: Device memory hierarchy [79].

Local and register memory is only available to each thread, while shared memory is available to the entire thread block and all other global memory is available to the entire grid, see Figure 1-8 which gives a helpful depiction of how CUDA programming interfaces with the physical device architecture.

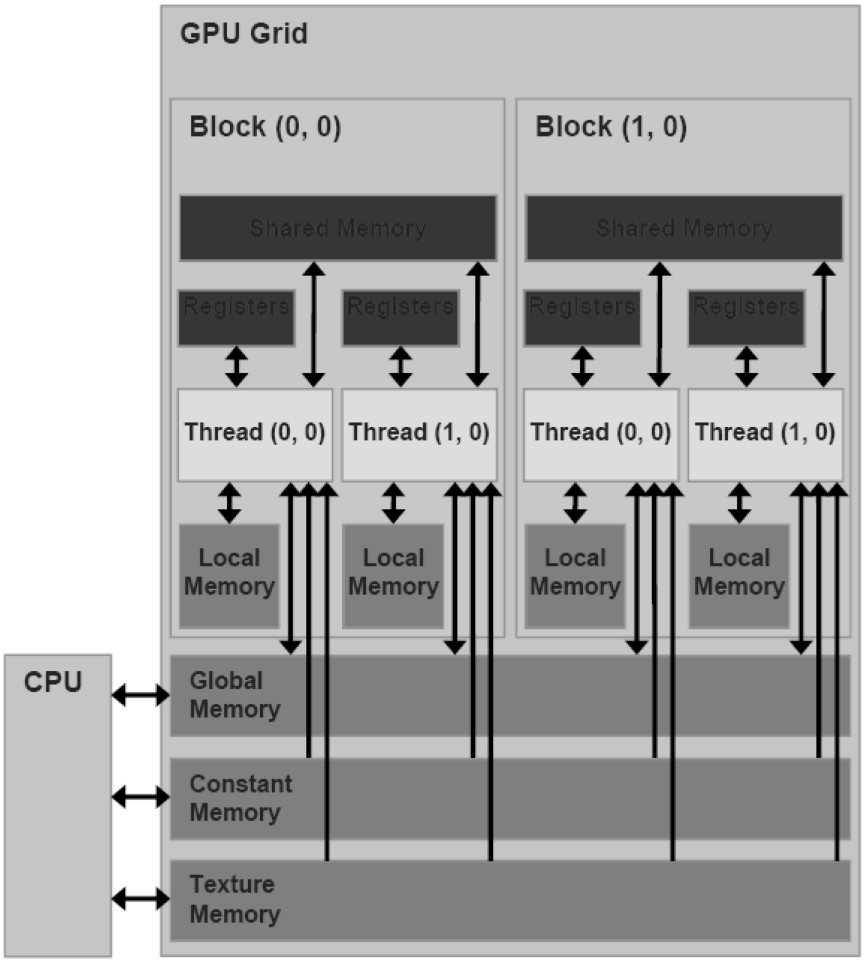

Figure 1-8: Memory handling on the GPU where dark grey blocks depict memory residing next to the SMs and medium-dark grey blocks depict DRAM [79].

In Figure 1-8, the dark blocks represent the memory types that reside next to the processors while medium-dark blocks represent dynamic random-access memory (DRAM) which holds most of the memory available to the entire GPU. According to the NVIDIA programming manual, computation on registers and shared memory requires only 24 processor cycles, whereas global memory requires 400 to 600 cycles [79]. Hence, CUDA codes that make efficient use of register/shared memory are exceptionally fast. In combination with effective memory handling, the number of threads per block (i.e. block size) is determined by balancing

the number of processors running simultaneously (known as occupancy) with the amount of registers and shared memory available to each thread. Increasing the number of threads per block increases GPU occupancy (i.e. the number of SMs running at once) but decreases the amount of register/shared memory for each thread. For most applications, an optimal number may be determined which ensures 100% occupancy, meaning all SMs effectively are running simultaneously, while allowing sufficient register/shared memory available for fast calculations, depending on the compute capability of the GPU. Thus, while CUDA provides a slick interface for parallel programming, the developer must consider the specific hardware that their programs are running on. Specifically, each GPU's compute capability determines its main capabilities and memory limitations, see Table 1-1.

Table 1-1: Specifications based on GPU compute capability [79].

Technical Specifications	Compute Capability									
	1.0	**1.2**	**2.0**	**3.0**	**3.5**	**5.0**	**5.3**	**6.0**	**6.2**	**7.0**
Maximum blocks per SM	8	8	8	16	16	32	32	32	32	32
Maximum threads per SM	768	1024	1536	2048	2048	2048	2048	2048	2048	2048
Maximum registers per SM	N/A	N/A	32K	64K	64K	64K	32K	64K	32K	64K
Maximum registers per thread	124	124	63	63	255	255	255	255	255	255
Maximum shared memory per SM (KB)	48	48	48	48	48	48	48	48	48	96
Amount of local memory per thread (KB)	16	16	512	512	512	512	512	512	512	512
Constant memory size (KB)	64	64	64	64	64	64	64	64	64	64

Based on the compute capability, the minimum block size for 100% occupancy may be determined by dividing the maximum number of threads per SM by the maximum number of blocks per SM. From there, the maximum amount of registers available per thread and shared memory available per block may be determined by dividing the maximum amount of registers per SM and the maximum shared memory per SM by the block size, respectively. For the Titan Black GPU (compute capability of 3.5) as an example, the minimum block size for 100% occupancy is 128. With 128 threads per block, only 32 registers and 24 bytes of shared memory

would be available for fast calculations within each thread, limited by the total amount of memory available per SM. If single precision was used to maximize the amount of values stored, only 38 fast memory spaces are available for each thread. Determining the maximum amounts of registers and shared memory are critical for speeding up applications since register/shared memory computations are considerably faster than global memory computations. Therefore, the amounts of register/shared memory available can significantly alter algorithms and memory handling within each kernel. If the block size specified for a kernel is less than the minimum block size, then less than 100% occupancy will be achieved (some SMs won't be running during kernel execution). On the other hand, the minimum block size may not provide enough registers or shared memory needed for fast memory access times within the kernel. If more register or shared memory than available for that block size is specified by the programmer, then the resulting code will run at less than 100% occupancy as well. Therefore, most applications will require a balance between GPU occupancy and fast memory computations depending on the amount of memory required for each kernel. Further discussion for FE analysis is found in the next section, Section 1.2.4.

Typical workflow for efficient memory handling using CUDA can be seen in Figure 1-9.

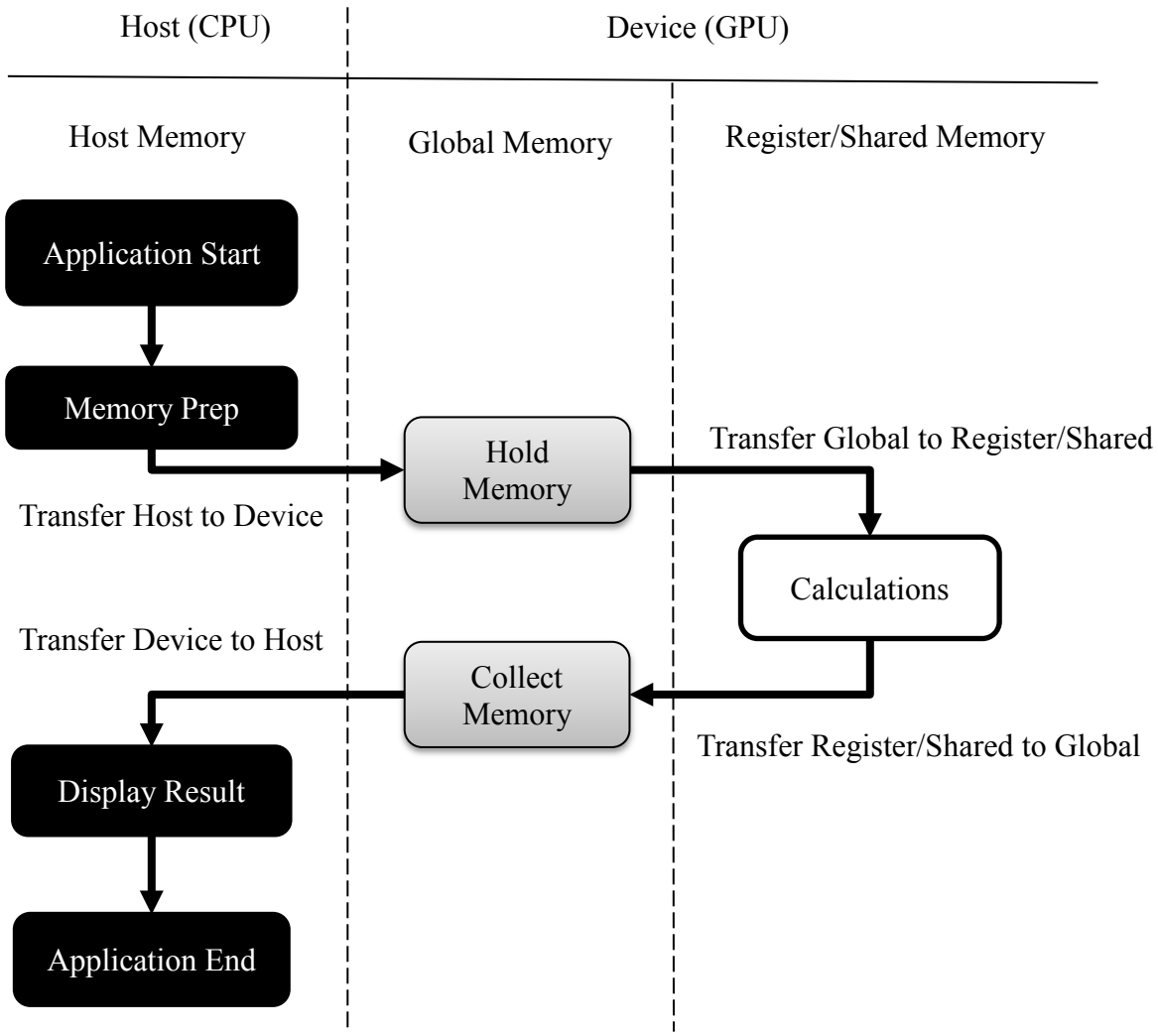

Figure 1-9: Efficient memory handling when launching kernels using CUDA.

1.2.4 GPU Computing for Finite Element Analysis

In the past, FE analysis was typically computed on CPUs using well-established programs, such as ANSYS and Abaqus. However, certain aspects of FE analysis are "embarrassingly parallel" [9]. As stated in Section 1.2.3, processes computed on separate memory spaces in similar manners are readily parallelizable. Thus, computations involving node location calculations and any element calculations, including stiffness building and stress/strain

calculations, can be readily run in parallel. On the other hand, some processes are considerably less parallelizable, including stiffness assembly and matrix solving, where calculations occur using the same memory space. See Figure 1-10 for workflow of typical structural FE analysis highlighting processes that are readily parallel. Regardless, parallelizing those node and element calculations of FE analysis would greatly improve computation speed, and current linear direct sparse matrix solving methods remain more efficient through CPU computation.

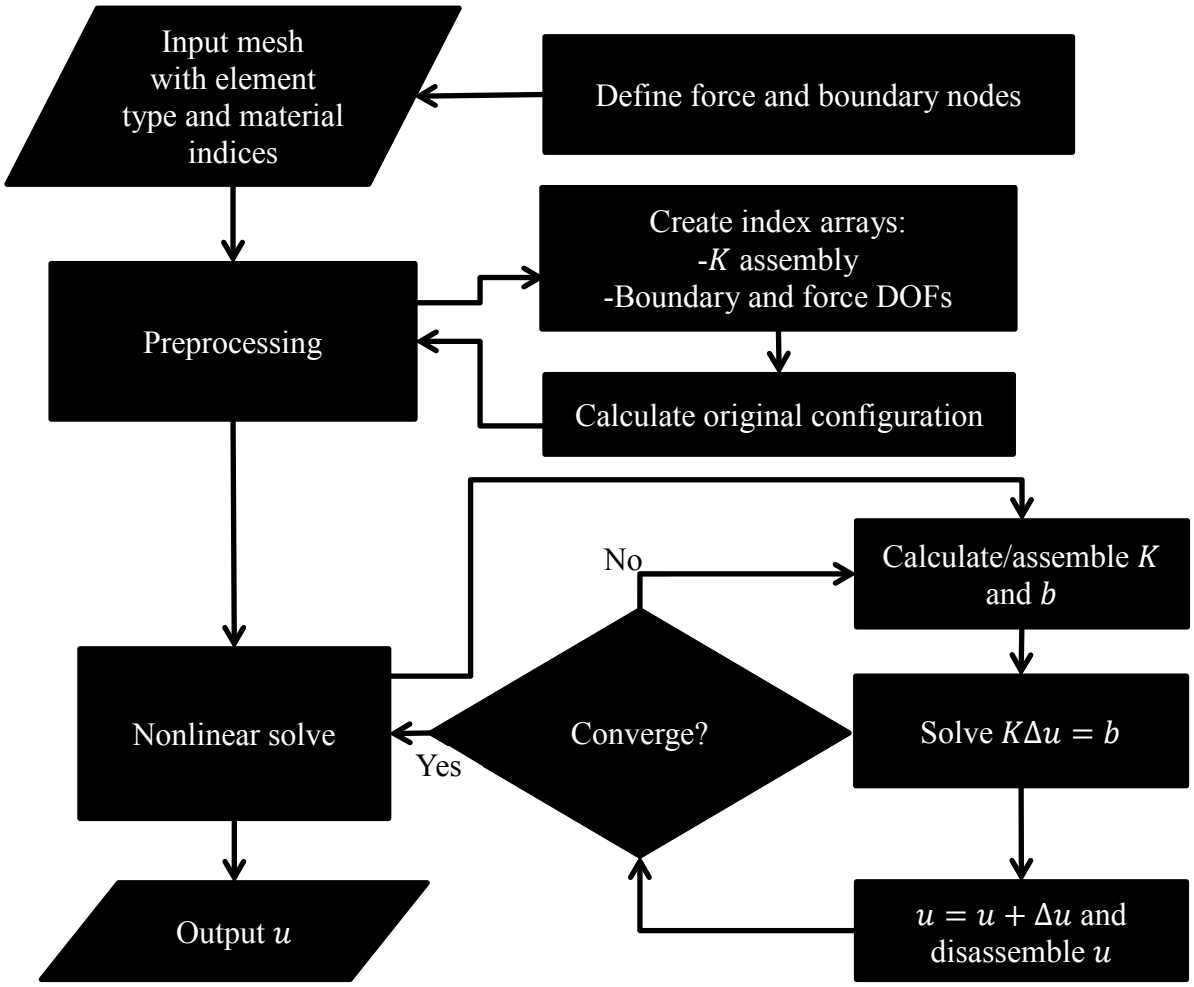

Figure 1-10: General workflow of nonlinear FE analysis

As for any program developed using CUDA, GPU computing of FE models requires certain considerations.

For even simple element types like the linear tetrahedral, see Appendix A, sizeable amounts of memory are required to efficiently store calculations during stiffness matrix building. For the current discussion, stiffness matrix building (linear tetrahedral element) is used as an example since it exhibits the greatest memory requirements during FE analysis, other than linear matrix solving. As seen from Equation (A-34) for building the linear tetrahedral stiffness ($\mathbf{K}_L$); the strain displacement matrix ($\mathbf{B}_L$) requires 72 floats, the material property matrix ($\mathbf{C}$) requires 36 floats, and the overall stiffness matrix ($\mathbf{K}$) requires 144 floats, in addition to other intermediate arrays including stress and strain calculations. These arrays may be reused for calculating the nonlinear stiffness matrix ($\mathbf{K}_N$). Consequently, element calculations ideally require 252 floats, which is significantly greater than available fast memory. As discussed in Section 1.2.3, increasing the registers per thread results in decreased occupancy. Clearly, to implement FE models onto GPU computing, either the occupancy must be decreased to allow for more fast memory available per thread or slower global memory must be utilized to ensure 100% occupancy. The optimal combination of occupancy and memory handling depends on utilization of the fast memory, which would require vastly different approaches to efficient memory handling; and therefore, considerable amounts of study to determine the best approach. For the purposes of the current study, a sufficient balance should be determined through FE knowledge and expertise. Other FE components implemented on the GPU should be generated with similar considerations, and in cases where a balance resulted in significant inefficiencies, the CPU will need to be used. Another example that requires GPU architecture considerations is vector assembly. Vector assembly involves collecting element vectors onto

the global nodal vectors; in other words, adding contributions from separate memory spaces (i.e. nodal calculations for each elements) onto the same global memory space (i.e. nodal calculations for the entire model).

A couple previous strategies for vector assembly include the scatter and gather methods [2], but considering the potential race conditions present in each strategy, other strategies have been developed considering GPU computation. See Section 1.2.5 regarding GPU strategies for vector assembly. As well, see Chapter 3 for further discussion regarding GPU computing of spine models.

1.2.5 Real-Time Simulation

Over the past thirty years, some researchers have started to realize the value of real-time simulation for biomedical applications. As such, novel algorithms were developed to improve the computation speed of deformable models, specifically focusing on surgery simulations. Most initial algorithms involve simplified deformation models that were less accurate than FE models but computationally faster. The most successful algorithms include: spring-mass models (including volumetric extensions) and ChainMail models [81]. Spring-mass models were generated by springs connecting discrete mass points comprising a mesh of the geometric body [81]. In accordance with Newton's second law of motion, the force equilibrium for each node of the mesh is described by:

$$m_k \frac{d^2 \boldsymbol{x}_k}{dt^2} + c_k \frac{d\boldsymbol{x}_k}{dt} + \boldsymbol{F}_k = \boldsymbol{R}_k \qquad (1-23)$$

where k is the node, m_k is the mass, c_k is the damping coefficient (representing viscoelasticity), $\boldsymbol{x}_k$ is the current coordinates, $\boldsymbol{F}_k$ is the internal force, and $\boldsymbol{R}_k$ is the external (applied) force.

The internal force is determined by springs connecting the nodes via Hooke's law:

$$\boldsymbol{F}_k = \sum_{l \in N_k} \boldsymbol{k}_k \boldsymbol{s}_k^l \qquad (1-24)$$

where N_k are the adjacent nodes, $\boldsymbol{k}_k$ is the spring stiffness determined from the material properties of the body and $\boldsymbol{s}_k^l$ is the difference between the initial and current lengths of the spring between nodes l and k.

Equation (1-23) may be discretized in time and solved using a fourth order Runge-Kutta method [81]. As seen from Equations (1-23) and (1-24), spring-mass models are simple in methodology and implementation with a reasonable degree of accuracy. However, the mesh exhibits difficulty propagating the deformations, especially for relatively stiff systems, where oscillations often occur as a natural result of the spring-type mesh. Moreover, spring-mass models demonstrate noticeable limitations in volumetric deformation since they lack volumetric considerations. Building upon the volumetric limitation of mass-spring models, ChainMail models were developed [82]. ChainMail models consist of a mesh of nodes connected by links (or chains), like spring-mass models. However, unlike spring-mass models, the ChainMail algorithm is vastly different. In general, ChainMail is built upon the idea of relative movement constraints for each node surrounding a moving node. The moving node generates constraints in the form of bounding regions for its surrounding nodes, where they must lie within the bounding regions of the moving node. In this manner, node movements are

propagated through the mesh, much like in a chain. In application, the ChainMail algorithm is broken down into two stages: first the propagation stage, then the relaxation stage. The propagation stage is defined by node movement through the mesh, in which one node (or a set of nodes) is moved, then that movement is propagated through the mesh via the bounding regions of each node until all constraints are satisfied. Then, the relaxation stage occurs where equilibrium, like Equation (1-23), is iteratively established through an energy minimization process. The system's energy is reduced by adjusting node positions and calculating the resulting system energy based on material properties until equilibrium is reached. Altogether, the ChainMail algorithm improves the stability and deformation propagation limitations of spring-mass models. In addition, recent work has improved the ChainMail method to conserve volume and strain energy, and to allow for the modelling of anisotropy and inhomogeneity [83]. Although promising for real-time simulation of biomedical models, the ChainMail method still exhibits considerable limitations. The volumetric capabilities may have been improved through recent work [83], but the method is not yet proven effective for complex models including materials exhibiting near-incompressibility. In all, spring-mass and ChainMail models provided greatly improved computation speed over traditional FE models, but their decreased accuracy and current lack of capabilities have proven ineffective for general real-time biomedical simulation.

In recent years, some algorithms that provide computational improvements for FE models have surfaced. Some linear formulations were developed to improve computation speed [5], [84], but these simulations did not meet the large deformation requirement of typical biomedical models. Consequently, Miller et al [10] invented and developed the Total Lagrangian Explicit Dynamics (TLED) algorithm for real-time simulation of biomedical tissues, specifically

applied to neurosurgery (i.e. brain models). The TLED algorithm applies an explicit centered differencing numerical scheme to the equation of motion while calculating the internal energy using finite elements. Over several publications, Miller's group has refined the TLED algorithm to incorporate various features, including contact [85], nonlocking tetrahedra [86], hourglass control [87], and more [21], [88], [89]. Based on their publication history, Miller et al developed the base TLED algorithm first [10], then built advanced FE features on top of that algorithm as they moved real-time simulation closer to their neurosurgery simulation application. The contact algorithm developed by Miller's group [85] revealed numerical contact formulations as considerable challenges that depend heavily on the base FE algorithm. Likewise, the original TLED algorithm has been tested in numerous other applications as well [90], [91], which shows that their first publication revealing the TLED algorithm proved useful for applications other than the primary motivation behind the study. Furthermore, they reformulated the TLED algorithm into the Meshless TLED [92] algorithm to better handle large deformations in biological tissues. Although fast for well-behaved materials, the TLED algorithm is severely limited by its critical time step, see Equation (1-25), which is highly sensitive to near-incompressibility.

$$\Delta t_{crit} = \frac{L_e}{c} \qquad (1-25)$$

where L_e is the characteristic length related to element size and c is the dilatational wave speed of the speed which has significant relation to bulk modulus of the material. Hence, the maximum time step decreases with increased bulk modulus.

Alternatively, Liu *et al.* [93] developed a different meshless algorithm based on smooth particle hydrodynamics (SPH). Traditionally, SPH is very computationally costly, so Liu proposed a

localized SPH method that reduced the number of nodes participating in the computation to a local interaction area. This novel SPH method increased computation speed considerably, but it was limited to local deformation; and therefore, it cannot accurately predict global deformations. In consideration of these limitations, more recent work has focused on using GPU technology to improve computation speeds.

As GPU hardware has improved, various researchers have developed parallel algorithms to improve FE computation speed. Miller's group developed parallel versions of the TLED algorithm that exhibited great increases in computation speed over the original TLED algorithm [21], [94], [95]. Although exceptionally fast per iteration, the parallel versions still suffer the same limitations as the original TLED algorithm. Other GPU-based work includes: a co-rotational FE method [11]; other meshless methods including parallel implementations of SPH and element-free Galerkin [93], [96], [97]; a parallel ChainMail algorithm [98]; an element-by-element preconditioned conjugate gradients algorithm [2]; and statistical-type methods including parallel proper generalized decomposition [99]. Courtecuisse *et al.* developed a comprehensive GPU-implemented simulation system that included deformation, contact, cutting, and haptic feedback [3], [11]. Their system is built upon a co-rotational formulation [100] for the deformable model solved via a backward Euler time integration (an implicit numerical method) using a conjugate gradient linear solver. The co-rotational formulation improves computational efficiency by considering only linear elasticity but adjusts for the large deformations by adding rigid body motion of the elements. Thus, the co-rotational formulation allows for large displacements and rotations but small strains, which means that small meshes sizes are required to ensure small strains within each element during large deformation. Likewise, small strains can limit the method's applicability considering the large

strains expected within biomedical tissue. An alternative method that models large strains better than the FE method is meshless methods [101]. Since meshless methods do not rely upon elements (and thus element quality) for strain energy calculation, they handle large deformations with ease. However, meshless methods are not as readily parallel as FE methods and they require significantly more computations. Karatarakis *et al*. [96] proposed a node pair-wise approach that was significantly more parallel (and thus faster) than the Gauss point-wise approach. However, the method was created for small deformation and has not yet been tested in large deformation and nonlinear material scenarios. Meshless methods have great potential for real-time simulations of biomedical models, but they are currently in their infancy compared to FE methods. On the other hand, Rodriguez *et al*. [83], [98] developed a parallel ChainMail method that demonstrated near real-time response for wrist surgery simulation. Regardless, the parallel version of the ChainMail still experiences the same shortcomings as the regular ChainMail algorithm. Lastly, Mafi & Sirouspour [2] developed a suite of GPU algorithms and techniques for parallelizing nonlinear FE models, including an element-by-element parallel implementation of a Jacobi-preconditioned conjugate gradient (PCG) solver. They demonstrated a strategy for coalesced memory storage along with a GPU-implemented vector assembly method that takes advantage of shared memory. Their coalesced memory storage strategy exhibited considerable improvements in efficiency over linear memory storage, in which global memory bandwidth was optimized by allowing parallel threads to access consecutive locations instead of block-size separated locations characterized by the linear pattern.

For their vector assembly method, shared and global memory indexing arrays are pre-computed to direct the atomic operations onto the correct memory spaces. Then during

computation, element vectors are collected on shared memory using atomic operations, and the shared memory vectors are then added to global memory using atomic operations. This vector assembly strategy also demonstrated considerable speed-up compared to previous strategies. Furthermore, their element-by-element PCG linear matrix solver demonstrated great computation speed-ups for well-behaved biomedical models. However, iterative solvers, like PCG, are dependent on the conditioning of the linear matrix; and therefore, it must be tested against direct sparse solvers for the targeted application. The PCG solver requires that the linear matrix be symmetric and positive-definite, which is not always the case during FE analysis. In all, although real-time simulation has been an area of considerable study over the past twenty years, no complete solution exists to the real-time problem and each method must be weighted on its pros and cons for each application. Otherwise, new methods must be developed for new applications. Further discussion is found in Chapter 3.

1.3 Thesis Objectives

In Sections 1.1 and 1.2, I outlined a need and focus for the current work and presented the current state of literature regarding this multidisciplinary undertaking. In summary, many FE lumbar spine models have been constructed and validated for various applications; in addition, numerous real-time FE techniques have been developed that take advantage of parallel GPU computing capabilities. Yet, no current study has attempted to build a real-time FE lumbar spine model by combining generic spine modelling methodologies with real-time FE techniques, in addition to developing novel FE techniques specifically for spine models. Considering that the proposed thesis represents an initial and fundamental exploration into real-time FE simulation of lumbar spine models, the scope shall be limited as such. To construct a foundation from which to generate prospective real-time lumbar spine models, the

target purpose for the proposed lumbar spine model will be general, in which the proposed model will undergo validation for gross physiologic movements. Furthermore, given the challenge of real-time facet contact simulation revealed by previous work in addition to potential applications without contact conditions, the proposed thesis will focus on developing a real-time FE lumbar model without facet contact, while acknowledging this significant limitation. Facet contact will an area of future work that will build upon the foundation provided by the current work; and thus, facet contact is out-of-scope for the proposed thesis. **Therefore, the primary objective of the proposed thesis is to create a FE lumbar spine model without facet contact using generic methodologies that exhibits close to real-time computation speeds while preserving accurate biomechanical response for gross physiologic movements.** The proposed thesis represents original progress towards the development of real-time FE lumbar spine models for specific applications, in which many potential models may be built upon the foundation provided by the current work. To achieve this primary objective, three specific aims are realized:

1. Develop generic modelling methologies then build, within a currently available commercial FE program, and validate a FE lumbar spine model that is specifically designed for parallel computing.

2. Test current real-time FE techniques in a custom CUDA program using spine material models and develop novel real-time FE techniques specifically designed with spine models in mind.

3. Create and test a generic FE lumbar spine model without facet joint forces in a custom CUDA program that exhibits validated biomechanical response and significant increases in computation speed compared to conventional models.

1.4 Thesis Outline

The proposed thesis is divided into five Chapters. Motivation for the proposed work and a literature review was presented in Chapter 1. In Chapter 2, a novel generic FE lumbar spine modelling methodology is proposed and compared to conventional models, with the consideration that the proposed model will be used for parallel computation. Validation of the proposed model is presented, and comparison to the conventional model is made in terms of accuracy and computation speed. The model presented in Chapter 3 aims to meet criteria implied by specific aim 1. Previous real-time FE techniques and linear solvers are then evaluated for their usefulness regarding real-time computation of spine models in Chapter 3. Particularly, GPU methods for FE analysis are evaluated. Through evaluation of previous techniques, the proposed study used the techniques that produced that fastest result with good accuracy. In areas where previous techniques were insufficient, novel GPU techniques are presented and evaluated. Additionally, the best performing linear matrix solver is chosen for use with spine models. Also in Chapter 3, a real-time FE program is presented and tested against a conventional FE program, ANSYS. In Chapter 4, implementation of the proposed lumbar spine model, with facet contact removed, from Chapter 2 into the real-time FE program from Chapter 3 is presented. Some novel FE techniques specific to lumbar spine models are presented including their integration into the real-time FE program. Evaluation of the real-time lumbar spine model against the conventional model is demonstrated, and improvements for future development are suggested. Chapter 5 collects the results from Chapter 2 through

Chapter 4 and presents an overall discussion of the proposed work, including conclusions and future work.

Each of the specific aims from the thesis objectives (Section 1.3) are addressed and solved in each paper presented within the proposed thesis. Hence, Chapter 2 directly addresses Specific Aim 1; then Chapter 3 directly addresses Specific Aim 2; and finally, Chapter 4 builds upon Chapter 2 and Chapter 3 to resolve Specific Aim 3. Upon completion of each specific aim, the results are considered together and compared to the Primary Objective of the thesis in the Chapter 5. Conclusions (Section 5.1) and implications for Future Work (Section 5.2) are drawn from the discussion, which concludes the proposed thesis.

Chapter 2 Towards the Development of a Faster Generic Lumbar Spine Model Using Parallel Computing Considerations

A version of this chapter is under review as:

Maeda, N.K., Boulanger, P., Carey, J.P., Development of a Faster Lumbar Spine Model with Parallel Computing Considerations, Computer Methods in Biology and Medicine (2018).

2.1 Introduction

Numerous finite element (FE) models of the lumbar spine have been developed using both generic and problem-specific methodologies over the past 40 years. Some of the first models were developed to accurately portray general physiologic movements of the lumbar spine while providing insight into the biomechanical causes of back pain [38], [39], [102] or into spinal instrumentation [103]. Through early FE models, investigators were able to conduct simulations on loading scenarios, plus provide novel stress and strain data, that could not be achieved through experimentation [102]. From that point, prospective lumbar spine models built upon those early models by adding enhanced features, such as visco-elasticity [104] or muscle effects [105], and applying those models towards investigations of biomechanical properties [71], [104], [105] or specific clinical studies [31], [58]. Some models focused on implant and orthotic design [31], while others explored impact loadings [104]. Nevertheless, spine models have improved over time and current spine modelling methodologies may be derived from early investigations. Although FE modelling of the spine has effected improvements to clinical devices [31], [58], [106], few studies have aimed for direct clinical

integration of spine models [81], where the clinician may interact with the model to evaluate their proposed spinal intervention biomechanically and thus improve clinical outcomes. One study successfully applied an FE scoliotic spine model in the scoliosis clinic [58], where the FE model improved brace design. Although promising, long term outcomes are yet to be determined. Clinical integration of FE spine models is rare since spine models have a considerable limitation: current simulations require significant amounts of time to compute the spinal response (i.e. hours) [107], making them too slow for many practical applications. Even for scoliosis bracing, faster solve times (i.e. less than five seconds) would allow orthotists to adjust their designs quicker for faster brace fabrication. As another potential example (among others), lumbar spine models with computation times less than a second would open the door towards patient-specific evaluation of spinal implants during surgery. Hence, the clinical integration of FE spine models exhibits great potential; thus, researchers must address the barriers of clinical integration, especially computation speed, to realize this potential. To improve the computation time of FE lumbar spine models, every aspect of model development and solving must be considered.

Various FE lumbar spine models have been developed and validated against experimental data [30]. Although differences exist between many models, most FE lumbar spine models were developed using similar methodologies. The annulus fibrosus is often comprised of hexahedral elements embodying the matrix with crisscrossing tension-only spring elements for the fibers [51]. Also, the vertebrae are usually broken down into vertebral bodies and posterior elements [48]. Some studies have taken the bone density into account [72], but most simply split vertebral body into cancellous and cortical bone parts, where the cortical bone is often generated using shell elements. Studies often characterize the nucleus pulposus as either

nearly-incompressible (i.e. mixed displacement/pressure) solid elements [32], [49] or fully-incompressible hydrostatic elements [39], [108]. *In vivo*, the nucleus acts like a nearly-incompressible gel-like substance with properties close to water [23]. With regards to spine modelling, either representation is used and both exhibit similar results [30]. Ligaments are typically generated as tension-only spring elements [33], [48], [68], although some recent models have constructed them as shell elements [31], [46]. Most validated models include contact between the facet joint surfaces [30], [31], [48], [71], in which the contact is defined as frictionless and the effect of cartilage is sometimes included [109], [110]. Yet, some investigations have not included contact in their models depending on their applications [43]–[45]. Considering the low facet joint forces observed in flexion and lateral bending [30], some investigators also did not investigate facet joint forces for model validity in those loading conditions [32]. Altogether, most studies built their FE lumbar spine models using these generic methods. To ensure that their models represented accurate spine biomechanics, model results were typically compared to *in vitro* data collected from experiments with the same geometry. For most investigations, previous lumbar spine models conducted validation for physiologic loading conditions given their clinical applications of spinal implants or biomechanical investigations related to back pain. Recently, Dreischarf *et al.* [30] collected results from various well-validated spine models to show that the median of these models better predicts *in vitro* data. Hence, the data presented by Dreischarf could be useful for validating prospective spine models through comparison to a range of well-validated models.

Almost all successful models are meshed primarily with hexahedral elements to ensure model accuracy at a reasonable model size, especially considering the presence of nearly-incompressible materials. Yet, based on FE theory, tetrahedral elements require significantly

less computations than hexahedral elements [80]. Although tetrahedral elements are more efficient per element, each hexahedral element breaks down into five to six tetrahedral elements [111]. Reduced integration (with one integration point) greatly reduces the number of computations for hexahedral elements, but hourglass control is required to ensure that the hexahedral elements do not lock spurious modes [112]. Hourglass control introduces more computations making the reduced integration hexahedral element slightly less efficient. On the other hand, FE programs such as ANSYS add an enhanced strain field to the tetrahedral elements in order to improve stabilization and prevent locking [112]. Regardless, no study yet has compared a spine model built entirely from tetrahedral to conventional models built primarily from hexahedral elements, especially in the context of improving computation speed. Moreover, considering the nonlinearity inherent to spine biomechanical behaviour, solving FE spine models requires nonlinear analyses which is typically performed using the Newton-Raphson method [112]. Each Newton-Raphson iteration in the nonlinear analysis is broken down into the stiffness 'matrix build' phase and the 'matrix solve' phase [112]. Depending on element formulations, each model will exhibit different computation speeds within each phase of the nonlinear solve. Therefore, the effects of model parameters and formulations on each solving phase must be considered when developing a computationally improved FE spine model.

For defining the cortical bone surrounding the vertebral body, most models use shell elements [30]. Although more accurate in thin-walled scenarios, these elements can be relatively unstable and require significantly more calculations than the underlying solid elements [80]. In spine biomechanics, the vertebrae are orders of magnitude stiffer than the intervertebral disc and surrounding tissues [48]. Some musculoskeletal dynamic models have taken advantage of

this fact for applications involving orthopedic design [113], ergonomic design [114], [115], and rehabilitation investigations [116]–[118]. Using OpenSim and AnyBody software to simplify model development and solving, musculoskeletal dynamic models have provided insight into spinal muscle forces and vertebral displacements during various postures and movements. Although potentially useful for clinics such as scoliosis brace development and spinal manipulation therapy, no clinical setting has integrated such spinal models – other than for certain orthopedic and ergonomic design purposes. As shown by Shirazi-Adl [48], the vertebral bodies and posterior elements tend to act as rigid bodies connected at the pedicles by beam elements. Likewise, treating the vertebrae as rigid bodies would greatly reduce the problem size since the vertebrae account for a significant proportion of elements in the entire model. Rigid bodies are computationally more efficient than solid elements due to the decreased number of elements and overall reduced degrees of freedom (DOFs). However, the introduction of rigid bodies may reduce accuracy further in addition to causing numerical instabilities in the form of an ill-conditioned stiffness matrix. Mathematically, the entire stiffness of the vertebra would be collected into the surface nodes of the vertebral endplates, decreasing model stability. Also, material failure and yielding could not be investigated by this type of model, reducing its usefulness. Similarly, musculoskeletal dynamic models cannot calculate the intradiscal pressure, further reducing their usefulness. Regardless, the use of musculoskeletal dynamic models, plus models that treat the vertebrae as rigid bodies, could greatly increase the computation speed and is a promising avenue of future work.

Recent developments in computer memory and graphics processing units (GPUs) have opened the door to significant speedups for FE simulations. GPUs allow scientific computations to readily be computed in parallel on thousands of processors simultaneously instead of

sequentially as on central processing units (CPUs), allowing great speedups for applications involving repetitive computations, such as FE [2]. Some studies successfully applied real-time FE techniques on GPUs to other biomedical systems such as brain and soft tissue models [11], [88]. Yet, no studies have attempted to apply real-time FE techniques to lumbar spine models. Considering no work to date has focused on developing real-time FE spine models, novel efforts are required to move FE spine models toward real-time clinical scenarios as done for other biomedical simulations. Further development and application of current real-time FE techniques could allow for integration of spine models into clinical scenarios. However, before investigating real-time techniques for spine models, a novel generic methodology for generating an FE lumbar spine model designed specifically for GPU computation must be created.

This study takes the first step towards computational improvement of FE lumbar spine models through development of a novel model with better computational efficiency that allows for real-time computing options. By following previous FE lumbar spine models [30] and allowing for the wide potential applicability of the proposed model, a general validation against physiologic loading scenarios is pursued, in which detailed validation would be required in further study for specific applications. The primary objective of the current study is to develop a generic FE lumbar spine modelling methodology and validate a lumbar spine model for physiologic loading scenarios with the following criteria: increased computation speed over conventional models, and improved parallelism for a prospective GPU implementation. The research question that this work addresses is: would a novel FE lumbar spine modelling methodology produce a faster lumbar spine model than conventional models while preserving validity for physiologic loading scenarios?

2.2 Methods

An unmarked computed tomography (CT) scan (1.5 mm slices) was obtained of a full lumbar spine (L1 to L5). A solid model of the spine's bony components was created from the CT images using Simpleware (Simpleware Version 7.0, Synopsys, Exeter, UK) and saved as an IGES file. Then, using SolidWorks (SolidWorks 2015 SP4.0, Dassault Systems, Waltham, MA) on the same IGES file, intervertebral discs were created as slightly bulging volumes between each vertebra with a nucleus pulposus volume ratio of ~60% [38] and 1 mm cartilage endplates [44]. The mesh was generated using Hypermesh (Hyperworks 2017, Altair Engineering Inc., Troy, MI) from the solid model, in which the annulus layers were map meshed and the rest of the model was free meshed to form the irregular spinal geometry. Ligament attachment points were defined in Hypermesh based upon previous models [30] and the mesh was exported into ANSYS (APDL Version 17.0, ANSYS Inc., Canonsburg, PA) for analysis. All analyses were run on a desktop computer with 4 cores of an Intel Xeon E5-2630 v2 CPU (Intel Core, Intel, Santa Clara, CA) and 128 gigabytes of RAM. For all analyses, large displacement analysis was applied with a full Newton-Raphson non-linear solver and a sparse direct linear solver at each Newton-Raphson iteration. Convergence criteria for displacement and force was set to the ANSYS defaults [112], which for the current analysis was approximately 0.5 to 3.5 N for force convergence (using the L2 norm), 0.35 to 0.5 mm for displacement convergence (using the L2 norm), and approximately 12 to 40 Nmm for moment convergence (using the L2 norm), all depending on the model and loading involved.

2.2.1 Conventional Model

The conventional model was developed in a manner similar to previous models that utilized generic modelling methodologies [30]. As such, the annulus fibrosus was defined as collagen

fibers embedded in a ground matrix [50]. Linear hexahedral brick elements (SOLID185) with reduced integration comprised the annulus matrix while the fibers were generated using 2-node tension-only spring elements (COMBIN39) connecting the corner nodes of the annulus matrix brick elements. The annulus matrix mesh was generated such that the fibers were created as 8 alternating concentric layers [49] at alternating angles of ~30 and ~150 degrees from the transverse plane as done in previous FE lumbar models [68]. Although *in vivo* lumbar intervertebral discs typically have 15-20 layers [119], the current study's model generation is consistent with previous models [38], [49] that balanced modelling simplicity with sufficient accuracy, considering that the difference in gross biomechanical results between modelling 8 layers versus 15-20 layers would be negligible. Annulus fiber stiffness was varied from the most stiff outer layer to the least stiff innermost layer according to previous studies [39], and the volume weighting of the fibers with respect to the annulus matrix was varied from 23% at the outermost layer to 5% at the innermost layer following literature [44]. All remaining components, excluding the ligaments and cortical bone, were generated from linear hexahedral, tetrahedral, prism, and pyramid elements to form the irregular geometries. To handle near-incompressibility and improve model stability, pressure nodes were added to the nucleus pulposus, thus the nucleus was comprised of mixed displacement-pressure (u/p) elements. To model the different material properties of the vertebral bodies and posterior elements, each vertebra was split at the pedicles with shared nodes between the components. Further, the cortical bone was defined as shell elements (SHELL181) surrounding the cancellous bone of the vertebral body. Like the annulus fibers, the ligaments were defined as 2-node tension-only spring elements. Finally, facet contact was symmetrically specified between the superior and inferior facets of adjacent vertebra. The augmented Lagrange method

[112] defined the contact behavior between the facets with softened stiffness to represent the joint cartilage. Material properties for each component are shown in Table 2-1.

Table 2-1: Material Properties

Component	Model	Properties	Values
Cortical bone [51]	Proposed/Conventional	E_{xx}, E_{yy}	11,300 MPa
		E_{zz}	22,000 MPa
		G_{xy}	3800 MPa
		G_{yz}, G_{xz}	5400 MPa
		ν_{xy}	0.484
		ν_{yz}, ν_{xz}	0.203
Cancellous bone [51]	Proposed/Conventional	E_{xx}, E_{yy}	140 MPa
		E_{zz}	200 MPa
		G_{xy}, $G_{yz,}$ G_{xz}	48.3 MPa
		ν_{xy}	0.450
		ν_{yz}, ν_{xz}	0.315
Cartilage Endplates [51]	Proposed/Conventional	E	23.8 MPa
		ν	0.4
Annulus matrix	Conventional [51]	C_{10}	0.18 MPa
		C_{01}	0.045 MPa
		K	4.35 MPa
	Proposed [49]	C_{10}	0.56 MPa
		C_{01}	0.14 MPa
		K	14.0 MPa
Nucleus pulposus	Conventional [51]	C_{10}	0.12 MPa
		C_{01}	0.03 MPa
		K	1499.9 MPa
	Proposed	E	1.0 MPa
		ν	0.49958

Symbols are given as follows: E is the Young's modulus, G is the shear modulus, and ν is the Poisson's ratio in the x, y, or z directions for the orthotropic materials (independent of direction for isotropic materials); C_{10} and C_{01} are the material constants and K is the initial bulk modulus (related to incompressibility) for the Mooney-Rivlin material model. MPa represents megaPascals

Ligament properties were calibrated such that the model response matched previous literature data; see Figure 2-1 for a comparison of the current study's ligament response curves compared

to previous work [51]. Note the ligament stiffnesses shown in Figure 2-1 are different since each study utilized different geometry and the ligament stiffnesses were calibrated to match validation criteria. Likewise, the facet contact surface offset was set to 0.5 mm and the stiffness factor was 0.018 to represent the facet cartilage, in which the exact values resulted from calibration of the model to experimental data. Optimal convergence was desired to get the fastest model for comparison, so a trial-and-error procedure was conducted to calibrate these contact parameters. Similar to previous models [30], the model was calibrated to match experimental data by adjusting contact parameters for optimal convergence. The gap between the articulating surfaces was a result of the facet joint geometries. See Figure 2-2 for a depiction of the conventional model created for this study. The conventional model was comprised of 216 253 elements and 52 826 nodes in total.

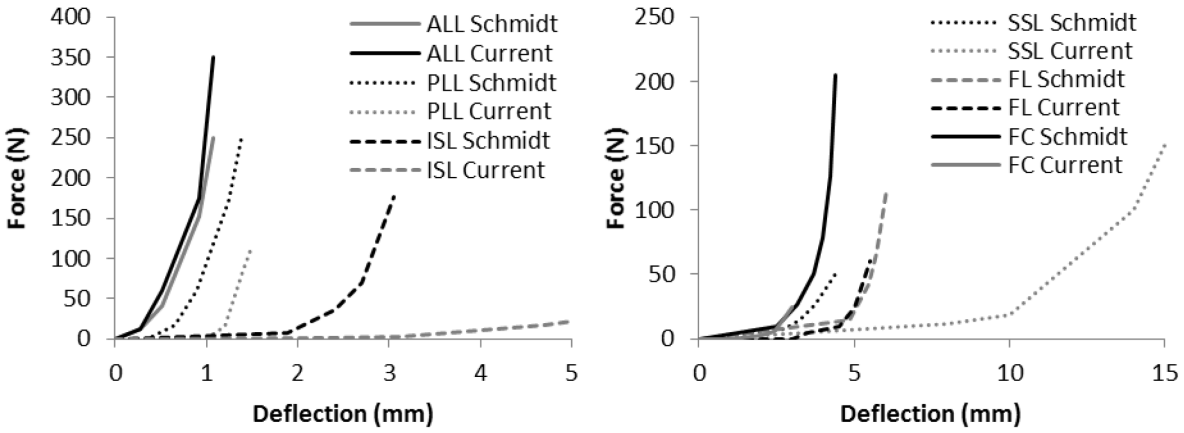

Figure 2-1: Ligament stiffness curves used for both the proposed and conventional models (i.e. "Current") compared to Schmidt's [51] ligament stiffness curves. (on the left) The anterior longitudinal ligament (ALL), posterior longitudinal ligament (PLL), and interspinous ligament (ISL) are shown on the left. (on the right) The supraspinous ligament (SSL), ligamentum flavum (FL), and facet capsulary ligament (FC) are shown on the right.

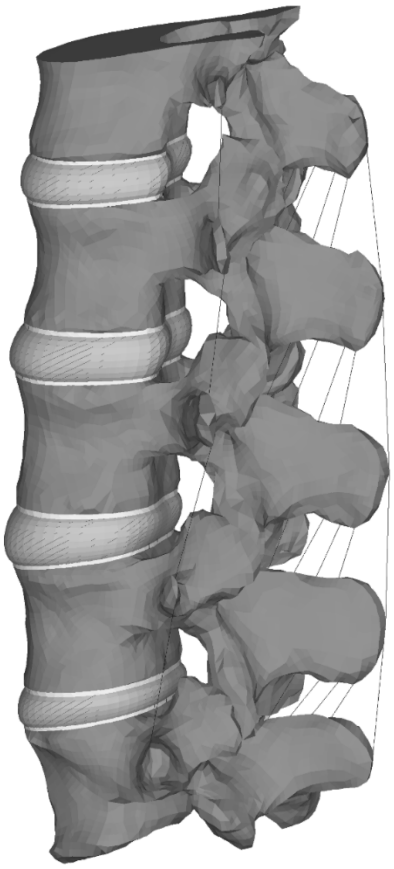

Figure 2-2: Depiction of the conventional spine model.

2.2.2 Proposed Model

The proposed model aimed to improve computational efficiency by decreasing the number of computations while preserving the accuracy of the conventional model for gross physiologic movements. Therefore, the proposed model exhibited four significant differences compared to the conventional model. Firstly, all solid components were composed of linear tetrahedral elements instead of a mixed linear hexahedral and tetrahedral mesh. Secondly, the u/p elements encompassing the nucleus pulposus were replaced with pure displacement elements. Thirdly, the cortical bone was defined as the outer layer tetrahedral elements encompassing the cancellous bone as opposed to shell elements. Lastly, the material properties were adjusted to improve accuracy and convergence of the model, see Table 2-1. The two material property differences between the conventional and proposed models were within the annulus fibrosus and nucleus pulposus. For the proposed model, the nucleus pulposus was defined as linear elastic, as opposed to Mooney-Rivlin for the conventional model, and the material properties were adjusted by decreasing the Poisson's ratio (starting at 0.4999) until good model convergence was achieved for all loading scenarios. Considering the slight decrease in bulk modulus in the intervertebral disc due to the nucleus pulposus, the annulus fibrosus material properties were adjusted to increase the bulk modulus, which improved model accuracy. To create the tetrahedral mesh, the conventional mixed mesh was split into purely tetrahedral elements following a previous algorithm for splitting hexahedral, pyramid, and wedge elements into a consistent tetrahedral mesh [111]. Thus, the number of nodes remained the same as only the elements were split, and the annulus fibers remained connected to the same nodes as from the conventional model mesh. Contact surface offset was adjusted slightly to improve convergence: 0.4 for the left L3 facet and 0.43 for the right L3 facet. Contact

parameters for the proposed model were adjusted through a trial-and-error procedure, as done with the conventional model. The proposed model has a similar depiction as the conventional model, as seen in Figure 2-2, since visually the only difference is hexahedral elements to tetrahedral elements. The proposed model was comprised of 274 680 elements and 52 826 nodes in total. Note the same number of nodes as in the conventional model. Mesh convergence was deemed not necessary for the current study as long as the models passed validation, and especially, considering that the study focuses on direct comparison between modelling methodologies rather than model application.

2.2.3 Model Validation

Both the conventional and proposed models underwent validation procedures to ensure that both models were accurate for basic physiologic loadings before comparison. Loading scenarios for validation were chosen as basic for allowing the development of prospective spine models following the proposed methodology, especially considering that most clinical spinal responses break down into basic physiologic movements. Moreover, the standard loadings typically conducted for spinal implant testing are pure moments [24], which are representative of many potential clinical scenarios. Model validation was achieved following Dreischarf's paper [30] comparing numerous well-validated FE lumbar models from literature. The tested loading scenarios included pure moments of 7.5 Nm in flexion, extension, lateral bending (left and right), and axial rotation (left and right), in addition to pure compression of 1000 N using the follower load method [68]. As per Dreischarf, moments were applied at the cranial end of the L1 vertebral body using a remote point attached to all nodes on the superior surface, and the follower loadings were applied at the node closest to the centroid of each vertebral body. The L1-L5 vertebral rotations resulting from the applied moments were

computed as the rotation of that remote point. Boundary conditions for all loading scenarios involved fixing the inferior surface of the L5 vertebral body in all DOFs. Intradiscal pressure was determined by calculating the average hydrostatic pressure of all elements comprising the nucleus. Each model was considered valid if its output – L1-L5 vertebral rotation for moment loadings and L4-L5 nucleus pressure for compression loading – fell within the range of outputs generated by previously well-validated models; see Dreischarf's paper [30]. Note that the applied loadings and expected responses are pseudo-static and thus do not include the hysteresis typical of *in vitro* models [24].

2.2.4 Model Comparison

Upon validation, both models were compared in terms of speed and accuracy. To conduct the comparison, only moment loadings were applied to each model with the same boundary and loading conditions as the validation stage. Compression loading was not investigated for comparison since it requires small time steps to satisfy the follower load, and thus, it would not be useful considering that the long-term focus of the current study is real-time solving and that it is another considerable challenge for real-time solving that would require a separate study. The number of substeps for each model under each loading scenario was optimized through preliminary convergence trials. For accuracy, displacements were compared between each model. For speed, the number of iterations for convergence, time per iteration, and total computational time were recorded from each model and compared. Considering that all simulations were run on the same workstation under the same conditions (including background processes) and geometry, differences in total computation time are attributed to the differences in element and material definitions between the models.

2.3 Results

2.3.1 Validation

The results from each loading case for each model satisfied the validation criteria, see Figure 2-3.

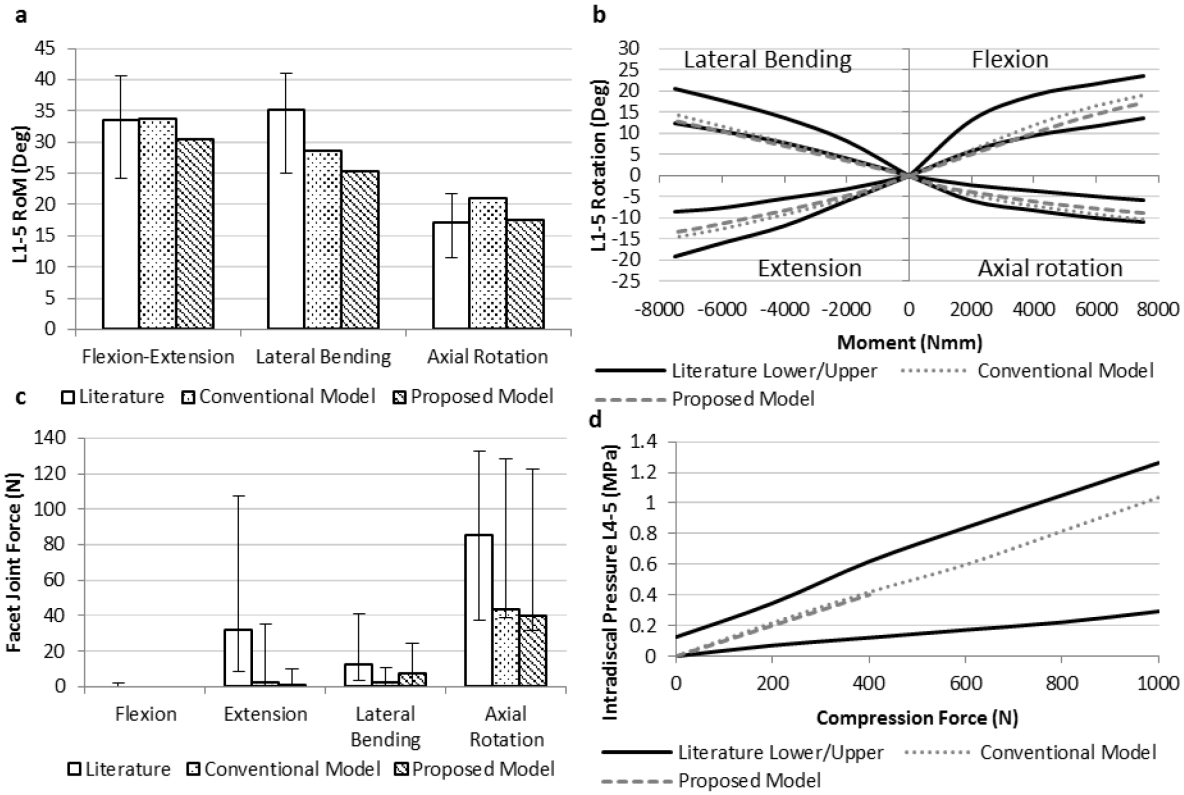

Figure 2-3: (a) L1-5 RoM of the conventional and proposed models compared to previously well-validated models from literature, as published by Dreischarf [30]. The red bar represents the median of previous models while the error bars represent the range. (b) L1-5 vertebral body rotation of conventional and proposed model compared to the most and least stiff well-validated models shown by Dreischarf. (c) Facet joint forces of the proposed and conventional models compared to Dreischarf's data. The median of all facets is shown with the maximum and minimum shown by the error bars. (d) Intradiscal pressure resulting from the L4-5 intervertebral nucleus pulposus under compressive follower load of the conventional and proposed models compared to the models exhibiting the highest and lowest pressures from Dreischarf's data.

The conventional model matched close to the median of previous FE data [30] for flexion, lateral bending, and compression, and within the range of previous well-validated FE lumbar spine models for extension and axial rotation. On the other hand, the proposed model better predicted the median of FE data for extension and axial rotation, and was within the range of previous well-validated models [30] for flexion and lateral bending. However, the proposed model did not converge past 400 N for the compression loading. This is likely due to locking of the tetrahedral elements under considerable hydrostatic pressure in the nearly-incompressible nucleus pulposus, although further investigation is necessary to determine the true cause. Also, facet joint forces of both the conventional and proposed models were ~30 N lower in extension than previous models, although the conventional model was closer. In general, both the conventional and proposed models were considered valid in representing a generic lumbar spine model's response to basic physiologic loadings, see Figure 3-1; and therefore, Model Comparison (Section 2.3.2) may be performed.

2.3.2 Model Comparison

See Table 2-2 for accuracy and speed comparisons between the conventional and proposed models. Also, see Figure 2-4 for a depiction of the speed comparison.

Table 2-2: Accuracy and speed comparison between proposed and conventional models.

Loading Case	Model	RoM (degrees)	Sub-steps	Iterations	Time (s)	Time/ Iteration (s)
Flexion	Proposed	17.97	4	21	263.1	12.5
	Conventional	19.11	4	20	378.2	18.9
Extension	Proposed	13.66	3	17	212.1	12.5
	Conventional	14.60	3	20	430.1	21.5
Lateral Bending Right	Proposed	13.53	3	16	204.4	12.8
	Conventional	14.13	1	9	192.1	21.3
Lateral Bending Left	Proposed	13.92	3	16	203.8	12.7
	Conventional	14.43	1	10	193.4	19.3
Axial Rotation Right	Proposed	8.90	3	16	198.04	12.4
	Conventional	10.60	3	20	413.1	20.6
Axial Rotation Left	Proposed	8.85	3	20	242.3	12.1
	Conventional	10.43	4	24	491.6	20.5

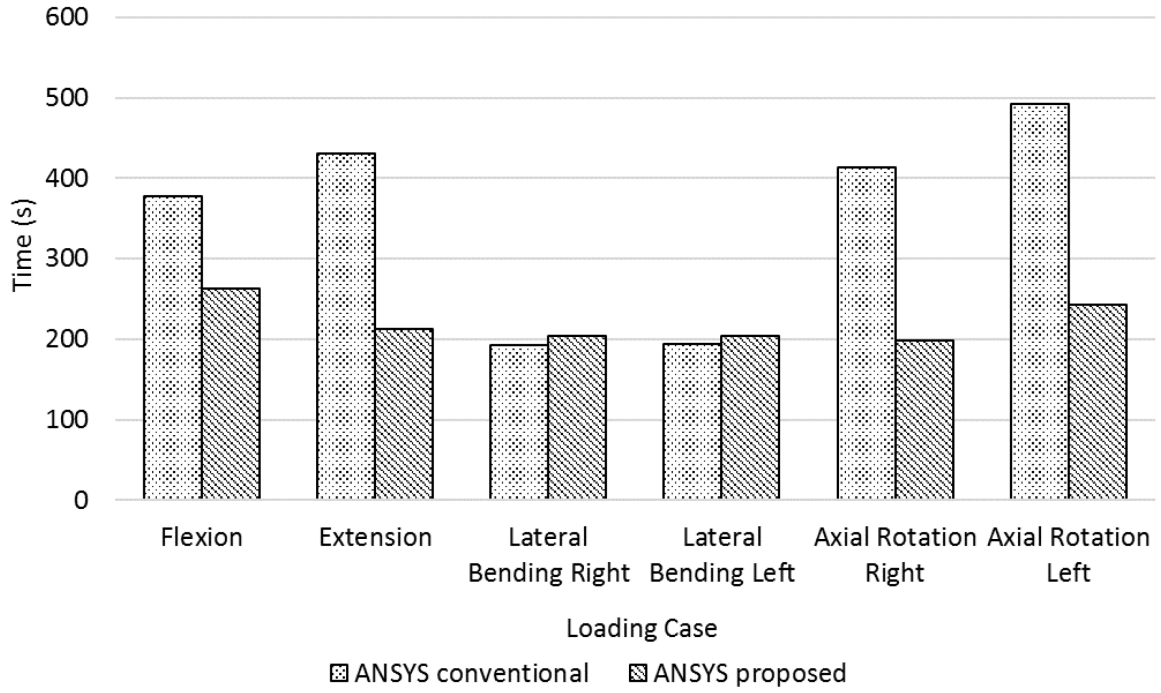

Figure 2-4: Speed comparison of the conventional and proposed models.

As shown in Table 2-2, the proposed model exhibits an average of 8.6% lower rotation than the conventional model for moment loadings, hence the proposed model is stiffer than the conventional model. Although the applied loadings are pure physiologic moments, out-of-plane rotation of approximately 1.8 degrees (at maximum) occurred due to minor asymmetry in the geometry. Yet, this coupled motion was minimal and consistent between both models, and therefore, not a concern for the accuracy and speed comparisons of the models. The proposed model had an average speed-up of 1.6X compared to the conventional model for all moment loadings. The greatest speed-up was 2.1X for the right axial rotation loading case, while the proposed model was slightly slower (0.94X) for the lateral bending loading cases primarily due to the number of iterations until convergence. Accordingly, the overall speed-up directly results from the increased speed per iteration since similar convergence was noticed between the two models except for lateral bending, see Table 2-2. Also, the CPU time for the matrix build phase of the conventional and proposed models were 77 and 94 seconds, respectively, showing that increased speed of the proposed model can be attributed to the linear matrix solve phase of the analysis rather than the matrix build phase. In other words, the conventional model was faster for the matrix build phase but much slower for the matrix solve phase, making the proposed model faster overall.

2.4 Discussion

Two FE lumbar spine models – a conventional and a proposed model – were validated for basic physiologic loading conditions against previous models and compared to determine whether simpler modeling techniques could improve the computational efficiency of spine models generated from generic methodologies. From Figure 2-3b, the gross vertebral rotation curves appear to lack some of the expected nonlinearity within the loading range, but the

tangent stiffness' match closely to the literature curves. The perceived lack of nonlinearity compared to other spine models can largely be attributed to geometric differences of the current model, especially within the facet joints, and partly attributed to the linear nucleus pulposus in addition to the increased stiffness of the annulus matrix. Although the moment-rotation curves appeared somewhat linear, the strain energy is nonlinear given the number of iterations until convergence exhibited during the model solving stage for each sub step. One concern for model validation was the facet joint forces in extension. Both conventional and proposed models were not within range of previous models (~10 N to ~110 N with a median of ~30 N); however, the median of facet joint forces from previous models is relatively low (~30 N) so this small difference between the current study and previous studies is not a significant concern. Moreover, the geometry used in the current study is different from previous models. The facet joint surfaces may be situated (i.e. the gap between the surfaces) such that contact doesn't occur in certain loading conditions including extension, which accounts for the lower facet joint forces for the current study.

Upon completion of validation, the proposed study investigated the impact of simpler elements and materials on model accuracy and speed. Through model comparison, the proposed study determined that the model simplifications were effective in decreasing computation time alongside an 8.6% difference in accuracy, in which the difference in accuracy is less of a concern given that the proposed model was validated. One noticeable oddity in the Table 2-2 is the substantial difference in convergence iterations (and thus computation speed) between left and right axial rotation. Some lack of symmetry in the geometry (since the geometry came from an unmarked human CT data set) between the right and left facets are a likely explanation for the convergence disparity, in which this point is supported by the consistency of this

convergence disparity between the conventional and proposed models. Nonetheless, the speed increase from the conventional to proposed model was modest but consistent for most of the tested loading scenarios, see Figure 2-4. The only concern was the lateral bending loading, where the proposed model was slightly slower. Effectively, slower model convergence caused longer computation of the proposed model for that loading scenario, in which contact conditions resulted in slower convergence. The median facet joint force for lateral bending in previous models is low at ~10 N and is highly dependent on geometry [30]. Hence, a model can still have valid kinematic response (moment-rotation curve) and intradiscal pressure without contact conditions for flexion and lateral bending loadings. Some researchers do not consider facet joint forces a major concern for their results in lateral bending [32]. Also, slower convergence is a result of 'near' state of contact as the model slows down significantly at the point of contact due to the sudden stiffness change in the model, which is all irrespective of the amount of contact force. On the other hand, the proposed model's response in flexion, where no contact occurred, and axial rotation, where contact occurred early along the loading curve, demonstrated that the proposed model is faster than the conventional model. Therefore, the methodology proposed by the current study exhibits considerable potential for the development of prospective lumbar spine models requiring improved computational speed. Model differences resulting in the decreased computation time are broken down into Element Formulations (Section 2.4.1) and Material Definitions (Section 2.4.2), while model Parallelization (Section 2.4.3) was a concern when building the proposed model considering the long-term focus of the current study.

2.4.1 Element Formulations

The main element formulation differences between the conventional and the proposed models were: an overall mixed mesh versus pure tetrahedral mesh, shell elements versus solid elements for the cortical bone, and mixed u/p elements for the nucleus pulposus versus pure displacement elements.

Tetrahedral elements tend to be stiffer than hexahedral elements in the case of large deformation. Notwithstanding, the overall difference in stiffness between the proposed and conventional models was small. Considering that the hexahedral elements of the conventional model were computed using reduced integration with hourglass control, they were still slightly more efficient to build than the tetrahedral elements with enhanced strain of the proposed model, as determined from the CPU formulation time. However, tetrahedral elements were focused on for the proposed model since they are more suited for GPU processing. Although the proposed model has significantly more elements, the overall efficiency of tetrahedral elements outweighs the increased element count. Furthermore, the difference in accuracy by using tetrahedral elements is minimal compared to the gains in parallelism and speed, especially considering that the proposed model passed validation criteria. Regardless, further stability considerations, such as material and shell element changes, were necessary since tetrahedral elements tend to lock in large deformation scenarios.

With regards to the cortical bone, the difference in overall biomechanical accuracy is minimal when switching to a thicker cortical layer of tetrahedral elements in the proposed model from a thinner layer of shell elements in the conventional model. The overall rigidity of the vertebral body compared to the rest of the model accounts for the preservation of accuracy in switching

from shell elements to solid elements for the cortical bone. Clearly, using a thicker cortical layer of tetrahedral elements is significantly more efficient than the shell elements alone. Still, further study is required to determine the exact improvements in efficiency and stability. Additional improvements in efficiency may be revealed by defining the vertebrae as rigid bodies [48], but losses of stability and utility are issues. Again, the use of rigid bodies for the vertebrae could greatly increase the computation speed and is a promising avenue of future work.

The last difference between the conventional model and proposed model is the use of mixed u/p elements to define the nucleus pulposus. Mixed u/p formulation helps stabilize elements comprised of nearly-incompressible material [80]. However, u/p elements add pressure DOFs to the model, increasing the model size, and thus, increasing the solve time per iteration. Also, the u/p element formulation results in a non-positive definite stiffness matrix so a slower matrix solver (such as a QR solver) was required to compute each iteration [120] for the conventional model. Similar results would be revealed for other models from literature using hydrostatic elements (HSFLD242 in ANSYS) to describe each nucleus [31], [68], [108] since they require extra pressure DOFs and result in a non-positive definite stiffness matrix as well. Therefore, pure displacement elements are preferred where stability is not a concern. As determined by the current study and given linear material for the nucleus pulposus, u/p elements are not required for stability since the tetrahedral elements appear sufficiently stable and accurate for spine biomechanics, given the current mesh size.

In all, although splitting the conventional model into tetrahedral elements resulted in significantly more elements and slightly increased model formulation time, the other element

formulation improvements resulted in improved matrix solve times. Hence, overall model solve times were faster for the proposed model. Nonetheless, other models could be generated that may be more efficient than the proposed model. In particular, element formulation times could be optimized by meshing with reduced integration hexahedral elements and solid elements for the cortical bone (instead of shell elements), while solve times could be preserved by using pure displacement elements instead of u/p elements in order to take advantage of faster matrix solvers. However, the focus of the current study was preparing spine models for real-time applications by taking advantage of parallel computing, therefore tetrahedral elements were built into the proposed model. Given the results of the current study, the element formulation time difference between the conventional and proposed models was small, and therefore, the improved overall computation time of the proposed model by using pure displacement reduced integration hexahedral elements instead of tetrahedral elements would be small as well. Still, further investigation would be necessary.

The current study proved that faster, yet accurate spine models can be developed using simpler elements, but further improvements might still be possible. Other optimizations may have potential, but one formulation exhibits considerable promise. ANSYS uses the updated Lagrangian Jaumann (ULJ) formulation [112] to build its elements, yet the total Lagrangian (TL) formulation [80] is more efficient since the Jacobian doesn't need to be recalculated for each iteration in TL. Still, further work is necessary to develop and evaluate the use of the TL formulation for real-time spine models.

2.4.2 Material Definitions

In addition to the differences in element formulations between the proposed and conventional models, the proposed model also changed two material definitions to improve model efficiency. The two material changes were: linear material for the nucleus pulposus; and stiffer Mooney-Rivlin constants describing the annulus matrix.

Linear material properties generate faster solving models since the material stress-strain matrix does not require updating between iterations. Moreover, linear material properties tend to be more numerically stable for the same reason. However, they are not accurate when approximating the response of spinal tissue, particularly in large deformation scenarios. In spine biomechanics, the axis of rotation passes through the intervertebral disc since the nucleus possesses a large bulk modulus [23]. Hence, large strains are not expected within the nucleus pulposus so linear material properties are sufficient for accurate spine displacements in response to physiologic moments, as noticed by the current study and others [32], [39].

The condition number of the material property matrix has a considerable effect on model stability [80], [120]. To improve stability of the proposed model, the Mooney-Rivlin constants were altered to increase stiffness. By increasing the stiffness of the annulus matrix, the stiffness gradients between the vertebrae and intervertebral discs were decreased. This resulted in improved numerical stability which improved model convergence. Although increasing the annulus stiffness altered the proposed model's accuracy, the improved convergence and overall model speed overlap the current study's goals considering that the model still meets the validation criteria.

2.4.3 Parallelization

The proposed model was developed with real-time solving considerations in mind since the long-term focus is on increasing the capability of spine models adapted and developed specifically for clinical scenarios. With other biomedical FE models, the use of GPUs for model solving has significantly increased the computational speed [11], [21]. Hence, parallelization of model building and solving at the model development stage was a critical consideration when creating the proposed model. For CPU computation, the increased element counts of the proposed model increased the nonlinear model build time per Newton-Raphson iteration since the elements are built sequentially on the CPU. On the other hand, GPU computation allows the elements to be built in parallel, thereby decreasing computation time by building thousands of elements at the same time [79]. Therefore, although the proposed model has substantially more elements than the conventional model (274 680 vs 216 253, an increase of 27%), the proposed model is better suited for GPU computation. In comparison with the conventional model, the proposed model boasts fewer computations per element over a larger number of elements, thus splitting the calculations over a larger number of processors in GPU computation. Hence, more elements are computed at a single time, greatly increasing the computational speed for building the stiffness matrix and load vector. Consequently, the proposed tetrahedral model is the primary focus of future GPU development towards real-time computation.

2.4.4 Limitations

Although generally successful, a significant limitation exists in the proposed model: it is unstable in pure compression loading via the follower loading method. Hence, the applicability of the proposed model is limited to scenarios without compression loading where the patient

must be lying down and no large axial compression loadings are applied to the lumbar spine [121]–[124]. Applicable scenarios include SMT or spinal surgery. Nonetheless, the proposed generic modelling methodology provides a basis and a guide for developing other computationally improved models, whether they use all or some of the proposed improvements. The current study design has limitations. Since both models were built from the same lumbar spine geometry, the effect of differing geometries on the resulting computation speeds was not investigated, even though differences in geometry between models account for significant differences in model response to loading [125]. Still, considering that the conventional and proposed models are geometrically exact, the results are independent of geometry and the proposed methodology should produce a similar result when applied to other lumbar spine geometries. Also, contact conditions have a significant effect on model convergence, but their effect was not thoroughly examined. The contact parameters effectively changed both the contact stiffness (i.e. the effect of joint cartilage) as the facets come into contact and the point when the facets come into the contact (i.e. the initial gap between the contact surfaces). Contact parameters were optimized to achieve good convergence for all the tested loading scenarios (which required a balance between each of the loading scenarios) considering that the goal of the study was computation speed with validity. The overall modelling modifications to the proposed model caused increased overall stiffness which changed the point when contact occurred between the facet surfaces. For the case of lateral bending, contact for many of the facet surfaces occurred just before 7.5 Nm making convergence difficult for the proposed model (similar for the conventional model where some of the surfaces are just coming into contact at 7.5 Nm). Additionally, some of the surfaces are in contact while others are on the verge of contact, which affects the convergence behavior as

well. Considering that the proposed model exhibited slower convergence than the conventional model, the proposed model likely had more surfaces in a near contact state than the conventional model. Regardless, the qualitative comparisons of computation speed are affected minimally since the contact conditions only changed *when* contact occurs along the loading curve and not the loading curve itself. In any case, the inclusion of contact was still necessary considering that: contact is critical in accurate spine models, and the presence of contact demonstrates that the proposed model is still faster with contact included (i.e. axial rotation where contact forces are significant). Further study may determine the quantitative effect of contact on model convergence between the conventional and proposed models. Regardless, the current study ensured ideal convergence and accomplished better comparison that focused on the element formulation and material differences between the two models, although different geometries with different contact situations (gap and stiffness) could be investigated further. Lastly, the current study only considered physiologic loadings up to 7.5 Nm as done for validating other lumbar models against pure moments [24], [30], but other pseudo-static loading scenarios would likely produce similar results. The tangent stiffness curves at 7.5 Nm match close to previous literature, see Figure 2-3, so similar results are expected for loadings past 7.5 Nm. Also, spinal response to other pseudo-static loading scenarios, such as the postero-anterior loadings of spinal manipulation therapy, principally break down into the physiologic loadings (for each motion segment) explored by the current study, although further investigation and validation would be required to apply any such model to clinical scenarios. One final consideration must be realized: the proposed results are qualitative in that only one geometry was compared. Hence, the quantitative results would differ for different geometries, but the overall comparison between the proposed and conventional modelling methodologies

would remain the same. In all, despite its limitations, the current study is useful as a guide for spine model development bearing in mind real-time applications.

2.5 Conclusion

As one step towards developing faster spine models using generic methodologies, the primary objective of the current study was to develop a generic spine modelling methodology that exhibited improved computational efficiency over conventional methodologies while considering parallelization. As shown, the proposed model outperformed the conventional model in the tested loading conditions since it runs faster for the same problem size while preserving accuracy and validity for basic physiologic loading conditions. Improved speed was attributed primarily to linear matrix solving times. Numerous changes from the conventional model were made to generate the proposed model based on other works and theories in the literature, but the effects of each change were not investigated. The aim focused on first identifying if such modifications were of value and then to guide future work. Thus, an optimal model was not acquired through the current study, although it could be through further investigation of the methods used, specifically with the material changes in addition to pure displacement elements for the nucleus and cortical bone. Regardless, the use of tetrahedral elements improved the parallelization of the proposed model while still exhibiting increased computational performance over the conventional model. In meeting the current study's aims, the proposed spine modelling methodology can be used to bring lumbar spine models closer to real-time clinical applications, such as SMT, spinal surgery, or spinal implant design during intervention. From this starting point as a building block and by using the proposed methods, computationally improved lumbar spine FE models may be developed for potential integration

into clinical scenarios. Nonetheless, further work is necessary before the goal of accurate, real-time spine simulation is realized.

Chapter 3 Evaluation and Development of Real-Time Finite Element Techniques to Simulate Spine Model Deformations

A version of this Chapter is under review as:

Maeda, N.K., Boulanger, P., Carey, J.P., Evaluation and Development of Real-Time Finite Element Techniques to Simulate Spine Model Deformations, Computers and Structures (2018).

3.1 Introduction

Finite element (FE) lumbar spine models have contributed significantly to the study of lumbar spine biomechanics [30], [31], [41], [126] over the last 40 years, in addition to improvements in implant and orthotic design [31], [59]. Although potentially useful in clinical and training scenarios [107], [127], the development of real-time FE lumbar spine models is lacking in current literature. Current simulations of FE lumbar spine models are too slow for real-time clinical scenarios, such as scoliosis bracing or spinal surgery. Although generic lumbar spine model development methodologies have not yet been analyzed for computational speed, real-time FE techniques must also be developed that consider the inherent complexity of spine materials.

Most current spine models were computed using readily-available FE modelling software [30], such as ANSYS and Abaqus. Hence, their models were built upon non-linear element formulations within the software. For ANSYS and Abaqus, all non-linear structural elements are generated using the updated Lagrangian Jaumann (ULJ) formulation [112]. In comparison

with the total Lagrangian (TL) formulation, the ULJ formulation requires less memory during calculation and allows direct calculation of the true (Cauchy) stresses, which is advantageous for developing mixed displacement-pressure formulations. Since ANSYS and Abaqus were developed in the early 1990s, when memory was costly, they focused on the ULJ formulation. On the other hand, the TL formulation requires less computations than the ULJ formulation for basic elements. Hence, the TL formulation is theoretically faster than the ULJ formulation for basic structural analyses [2] given serial computation speed per element as a requirement for real-time FE methods. Another formulation showing promise due to recent developments is meshless techniques [101], specifically smoothed particle hydrodynamics (SPH) [93]. Meshless techniques were created in the 1970s as an alternative to FE, but gave way to FE as the primary numerical technique since FE satisfies continuity within the model (meshless is discontinuous) and requires less computations [9]. With the emergence of real-time biomedical simulations, meshless techniques have arisen as a viable alternative to FE. Meshless techniques are advantageous for biomedical tissues since they are not sensitive to element shape quality, and thus, are more stable in large deformation scenarios. On the other hand, they require significantly more calculations per node and are not as readily parallelizable as FE techniques since calculations within each node are interconnected [93]. Considering recent developments in memory size and computing power of graphics processing units (GPUs), parallel computability of simulation methods is a keen consideration for the development of potential real-time FE methods [128]. Some researchers have attempted to develop real-time GPU meshless techniques [89], [96], [97], but they are in their infancy relative to FE techniques. Still, meshless techniques provide a potential avenue for future research of real-time spine models. Lastly, mass-spring methods offer a fast alternative to FE, but they tend to be

inaccurate [129] depending on the model. Accuracy is another consideration for potential real-time FE methods.

Recently, improvements in GPU technology and development of a programming library known as CUDA have opened exciting possibilities for real-time simulation. CUDA allows programmers to readily parallelize their applications by taking advantage of GPU architecture and memory design, increasing computation speeds substantially by conducting many calculations at the same point in time instead of sequentially as done in computation on the central processing unit (CPU). Using CUDA, some researchers have focused efforts on developing real-time FE models for specific clinical applications [2], [11], [21]. Miller *et al.* focused on developing and refining a novel Total Lagrangian Explicit Dynamics (TLED) algorithm [10] on GPUs for application to brain surgery [21]. Further, the TLED algorithm has gone through considerable development including contact methods [85], hourglass control [87], and non-locking tetrahedral elements to handle nearly-incompressible materials (Poisson's ratio equal to 0.49) [86] within the TLED framework. TLED methods were successful in achieving significant speed-ups over conventional FE programs, but they were not "real-time" and they exhibited some limitations in application. The TLED algorithm was designed primarily for brain models, in which spine models encompass materials with considerably higher bulk modulus. For spine models, the incompressibility of the nucleus pulposus must be considered when evaluating potential real-time FE methods. TLED is limited by a maximum time step resulting from its explicit nature [10], in which the time step would be unrealistically small for a spine model considering geometric complexity and nearly-incompressible material (Poisson's ratio greater than 0.499 for spine models) [2]. On the other hand, Mafi designed a preconditioned conjugate gradient (PCG) matrix solver for the TL

formulation considering the capabilities and architecture of GPUs [2]. Mafi also outlined a coalesced memory storage strategy, see Equation (1-1), that demonstrated increased efficiency over linear storage strategy. Mafi's coalesced memory strategy ensured that parallel threads (of the same block) running the same instruction accessed consecutive global memory locations, which optimized global memory bandwidth. Additionally, Mafi presented a vector assembly strategy that performed atomic operations on faster shared memory instead of global memory, which exhibited speed-ups over previously developed scatter and gather vector assembly methods. The CUDA memory indexing scheme is:

$$index = size * blockDim.x * blockIdx.x + i * blockDim.x + threadIdx.x \quad (1-1)$$

where *index* refers to the target memory space within the global array, *size* refers the array size, and *i* refers a vector or matrix component in non-coalesced global memory.

Mafi's memory handling strategies are useful for developing other real-time FE programs, regardless of solver used. However, PCG solvers are highly dependent on the linear matrix's condition number, so Mafi's solver may not be successful for spine models, although investigation is necessary especially considering numerical stability as a concern for potential real-time FE methods for spine models. Another real-time FE algorithm was developed by Courtecuisse [3], where he refined a co-rotational linear strain algorithm for GPU implementation [11]. Nonetheless, a co-rotational FE formulation would be inaccurate since large strain effects are prominent within spine models. Other possible methods for real-time FE analysis include statistical-type methods [99], [130] based on model reduction techniques and principal component analysis. Although promising for certain applications, extensive pre-processing of the FE models is required for statistical methods [130], thereby reducing its

general utility to clinical scenarios that don't require immediate generation of patient-specific shape models. In all, although previous work was successful for each of these authors' specific applications, their methods are mostly not applicable to spine models. Therefore, novel efforts are required to move FE spine models toward real-time clinical scenarios, starting with the usage of spine material models to evaluate the suitability of real-time FE methods.

This study represents steps toward real-time simulation of FE spine models by implementing current and new real-time FE techniques on spine material models. Although some of the previous methods work well for other biomedical material models [2], [3], [21], spine material models are more complex (especially the nucleus pulposus) [30], [38], [51]. Therefore, to assess the value of the potential GPU methods for real-time FE spine models, spine material models must be tested first. Testing and development of real-time FE techniques on spine material models would open the door towards implementing real-time FE techniques onto FE spine models. In the current study, GPU methods specifically designed for spine material models were investigated and developed.

The research question that this work addresses is: how can real-time FE techniques improve the computation speed of spine material models? Hence, specific aims for this study include: (1) generate fast FE formulations for the least and most computationally difficult spine materials; (2) parallelize the formulations for GPU implementation; and (3) choose the fastest linear solver from readily available solvers.

3.2 Methods

A custom program was developed using Visual C++ (Visual Studio 2013, Microsoft Inc., Redmond, WA) and CUDA (Version 6.5, NVIDIA Corporation, Santa Clara, CA) that

implemented FE computations on a desktop workstation featuring a NVIDIA Titan Black GPU (NVIDIA Corporation, Santa Clara, CA) in conjunction with a commercial CPU (i5-2500 CPU, Intel Core, Intel, Santa Clara, CA). The CUDA program's speed and accuracy were compared to ANSYS (ADPL Version 17.0, ANSYS Inc., Canonsburg, PA) on the same workstation for the same model. For comparisons to ANSYS and between linear solvers, a cube model was used with a mesh of 41 526 nodes (124 578 DOF) and 231 650 linear tetrahedral elements, which is the approximate problem size as a typical lumbar spine model [36]. See Figure 3-1 for a depiction of the cube model including mesh and see Section 3.2.4 for further cube model description.

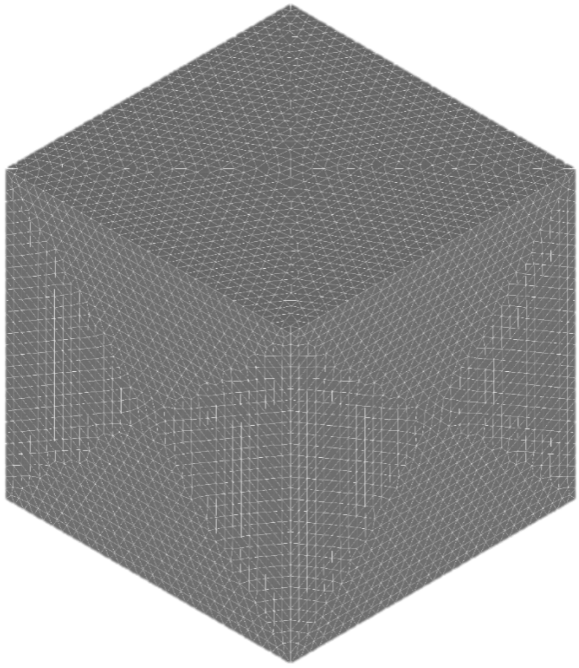

Figure 3-1: Depiction of the cube model with tetrahedral mesh used to test the CUDA program and linear solvers.

3.2.1 General Mathematical Framework

Nonlinear finite element formulations were required considering the large deformation effects expected when applying loads to tissues. A commonly used and effective method for real-time simulation [2], [10] is the TL formulation developed by Bathe [80] from which all elements of the current model were built, see Equation (3-2) for the principle of virtual displacements using the TL formulation. During computation, all variable derivatives are referred to the original configuration as opposed to the current configuration (as in the ULJ formulation):

$$\int_{{}^{0}V} {}_{0}^{t+\Delta t}S_{ij}\,\delta\,{}_{0}^{t+\Delta t}\epsilon_{ij} = {}^{t+\Delta t}R \qquad (3-2)$$

where R is the external force vector, V is the volume, $S = C\epsilon$ is the 2nd Piola-Kirchoff stress tensor, $\epsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i} + u_{k,i}u_{k,j})$ is the Green-Lagrange strain, $u_{i,j} = \frac{\partial u_i}{\partial x_j}$ is the gradient of displacement, u_i is the nodal displacement in the *i* direction, and x_j is the original nodal coordinates in the *j* direction. Subscripts refer to the referenced configuration from which the current derivatives are calculated, while the superscripts refer to the calculated configuration. The subscript or superscript "0" refers to the original configuration, while "t" refers to the current configuration, and "$t + \Delta t$" refers to the next configuration. For example, ${}_{0}^{t+\Delta t}\epsilon_{ij}$ is calculated using ${}_{0}^{t+\Delta t}u_{i,j} = \frac{\partial\,{}^{t+\Delta t}u_i}{\partial\,{}^{0}x_j}$ and u_i is calculated from the "$t + \Delta t$" configuration and x_j is calculated from the "0" configuration.

Unlike ULJ, the TL formulation eliminates the need to re-compute the configuration (i.e. the Jacobian) at each Newton-Raphson iteration. Instead, computation of the initial displacement effect is required for the strain-displacement matrix derived from the Green-Lagrange strain,

Equation (3-3), at each iteration. Still, the overall number of computations for the initial displacement effect is significantly less than re-computing the current configuration:

$$ {}_0e_{ij} = \frac{1}{2}\left({}_0u_{i,j} + {}_0u_{j,i} + {}^t_0u_{k,i}\ {}_0u_{k,j} + {}_0u_{k,i}\,{}^t_0u_{k,j}\right) \qquad (3-3) $$

where ${}_0e_{ij} = {}_0\epsilon_{ij} - {}_0\eta_{ij}$ is the linear strain component (η is the nonlinear strain component and ε is the overall Green-Lagrange strain), and ${}_0\epsilon_{ij} = {}^{t+\Delta t}_0\epsilon_{ij} - {}^t_0\epsilon_{ij}$. A subscripted variable with no superscript refers to the difference between the current configuration (t) and the next configuration (t+Δt).

Shape functions for the linear tetrahedral elements were the same as for the ULJ formulation in ANSYS [112], by:

$$ u = ru_1 + su_2 + tu_3 + (1 - r - s - t)u_4 \qquad (3-4) $$

where r, s, t are the isoparametric coordinates of the element.

All derivatives are calculated with respect to the original configuration at time 0 as per the TL formulation. The rest of the derivation followed Bathe's work [80].

The full Newton-Raphson nonlinear solver method was used to compute the overall nonlinear equation:

$$ \int_{{}^0V} {}_0C_{ijrs}\Delta\ {}_0e_{rs}\delta\ {}_0e_{ij}d\ {}^0V + \int_{{}^0V} {}^{t+\Delta t}_0S_{ij}\delta\Delta\ {}_0\eta_{ij}d\ {}^0V $$
$$ = {}^{t+\Delta t}R - \int_{{}^0V} {}^{t+\Delta t}_0S_{ij}\delta\,{}^{t+\Delta t}_0\epsilon_{ij}d\ {}^0V \qquad (3-5) $$

where ${}_0C_{ijrs}$ is the material property tensor and Δ refers to the increment from the Newton-Raphson iteration. At each iteration, ${}^{t+\Delta t}_{0}S_{ij}$ and ${}^{t+\Delta t}_{0}\epsilon_{ij}$ are updated and Δu is solved for to get the next increment, then $u = u + \Delta u$ and the process is repeated.

Considering the size of typical lumbar spine models [36] and the significant computation time expected per iteration, fast convergence was required to minimize the number of iterations. Also, no sudden stiffness curve changes are expected for spine models in response to gross physiologic movements (i.e. flexion, extension, lateral bending, and axial rotation), as observed in Dreischarf's work [30], so line-search or secant methods would not improve convergence (and thus computation speed). These considerations are necessary given the potential application of the given work on spine material models to full lumbar spine models.

3.2.2 Program Framework

The FE formulations were implemented onto the GPU using CUDA [79]. To reduce the number (and time) of GPU/CPU memory transfers, all model data was initially transferred to the GPU global memory and kept on the GPU during the solve phase, then the result was transferred back to the CPU. If the linear solver of the solve phase was implemented on the CPU (as opposed to the solve phase staying on the GPU), the stiffness matrix and load vectors were transferred to the CPU and the resulting displacement vector was transferred back to the GPU. Hence, all other data processing was completed on the GPU to ensure that minimal time was lost to data transfers. Also, the assembled stiffness matrix and unassembled force vector, including stress and strain updates, were created within a single stiffness building kernel to minimize the number of kernel launches. Moreover, fast-math and optimized CUDA libraries, such as cuBLAS [79], were used. To increase computation speed further (with a slight loss of

accuracy), all computations in the CUDA program were single precision as opposed to double precision, which is used by ANSYS. See Figure 3-2 for the flowchart comprising the program framework.

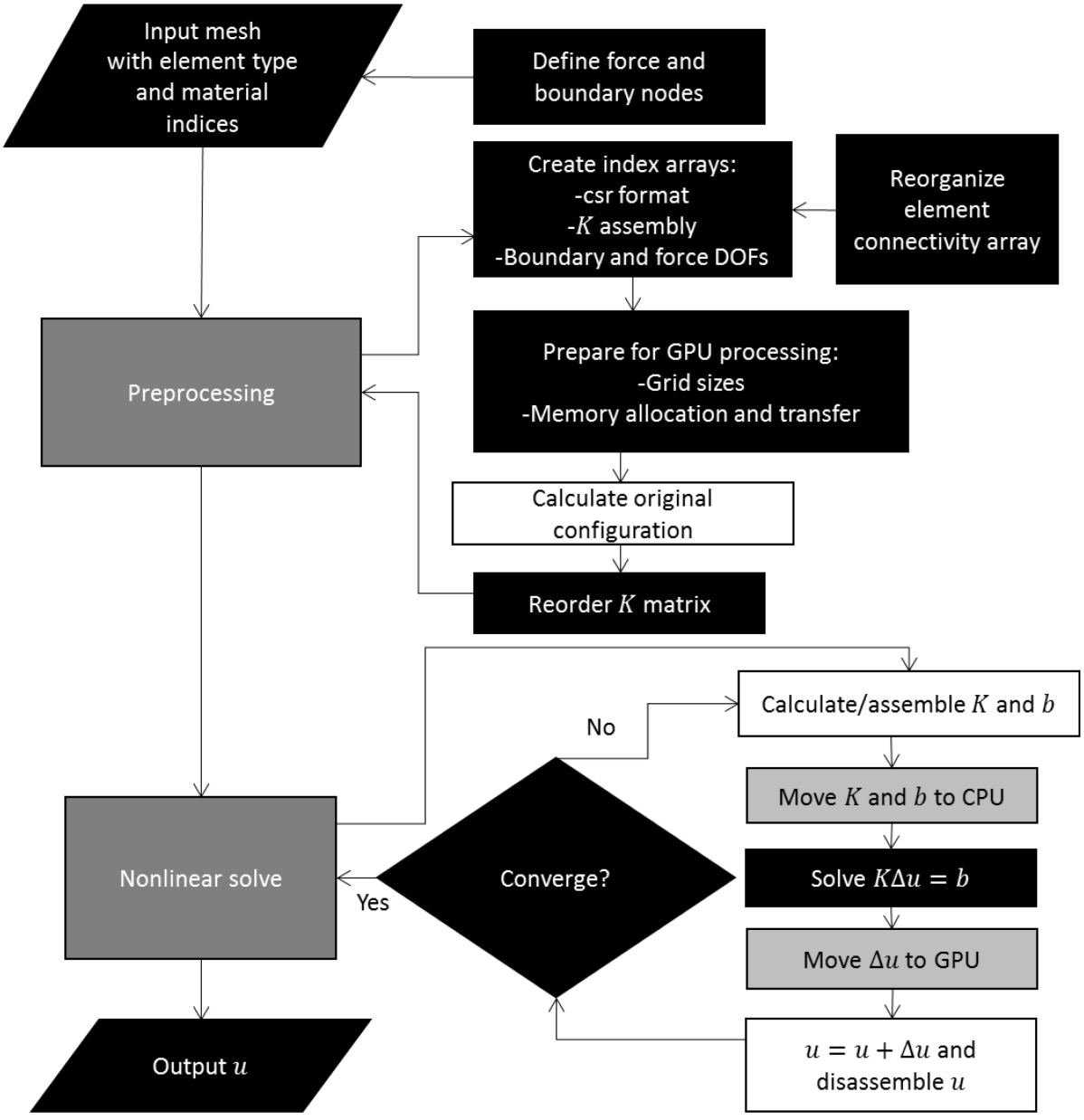

Figure 3-2: Flowchart for the CUDA program with CPU implemented linear solvers. CPU processes are depicted by black boxes with white text, GPU processes are white boxes with black text, processes run on both are grey boxes with white text, and data transfer processes are grey boxes with black text. $\boldsymbol{K}$ is the tangential stiffness matrix at each Newton-Raphson iteration, $\boldsymbol{b}$ is the residual force vector at each iteration, $\boldsymbol{R}$ is the external force vector, $\boldsymbol{\Delta u}$ is

the incremental displacement vector, and $\boldsymbol{u}$ is the total displacement vector. For the PCG solver, CPU/GPU data transfer processes did not occur since the linear solve was performed on the GPU (i.e. the "Solve $\boldsymbol{K\Delta u = b}$" process would be in a white box with black text instead and the grey data transfer processes would be absent).

3.2.3 GPU Memory Handling Strategies

GPU memory handling strategies were developed by building upon Mafi's work [2]. Coalesced global memory for element data reads and writes were performed as described by Mafi, including the vector assembly kernel that conducts atomic operations on the shared memory instead of the global memory. Shared memory was also coalesced according to the following indexing equation:

$$index = i * blockDim.x + threadIdx.x \qquad (3-6)$$

To achieve the maximum possible computation speed provided by the GPU, the use of occupancy, shared memory, and register memory were balanced. For the given Titan Black GPU with a CUDA compute capability of 3.5 [79], a thread block size of 128 threads was used to generate a balance between GPU occupancy and register/shared memory available to each thread. 128 threads per block is the minimum block size for 100% GPU occupancy, but that only leaves 32 registers and 24 bytes of shared memory corresponding to only 36 single-precision floats available per thread. However, element building requires significantly more memory. Therefore, to balance occupancy with memory usage, 128 threads per block, 128 registers per thread, and 384 bytes of shared memory (96 floats) per thread were specified. This corresponds with 6.25% GPU occupancy, which appears low but allows for significantly more efficient memory usage. Considering the massive memory requirements of FE analysis, memory usage takes higher priority than GPU processor usage. Even at 6.25% occupancy,

effectively 128 elements are running simultaneously per streaming multiprocessor representing massive improvements over serial element computation. Since each tetrahedral element stiffness matrix requires 144 floats, they were stored in global memory within the assembled model stiffness matrix. Within each kernel function, computation speed was optimized by first copying global memory onto shared or register memory, performing computations on shared or register memory, then writing the result back into global memory. This memory strategy, in addition to the increased usage of register/shared memory at the expense of GPU occupancy, minimizes the number of slow global memory accesses within the kernel function.

Stiffness matrix assembly was conducted directly into global memory promptly upon calculation within the stiffness building kernel. During computation, a pre-calculated indexing array used scatter method to add the stiffness matrix values onto global memory in compressed sparse row (csr) format [79], which immediately prepared the stiffness matrix for the linear solving stage and reduced the amount of memory required to store the stiffness matrix. With regards to an efficient matrix assembly strategy, preprocessing of the element connectivity array ensured that all elements within any single block did not share a single common node, a technique known as graph coloring [131]. Since the model was relatively large (~250 000 elements), this lack of node sharing was ensured, and race conditions were minimized within each block. Vector assembly was completed using Mafi's strategy [2].

3.2.4 Linear Solver Comparison

During each Newton-Raphson iteration, the FE program must solve a linear system of equations. Depending on the model, the stiffness matrix will have certain characteristics related

to solvability for certain linear solvers. Hence, the efficiency of linear solvers in FE analysis is highly dependent on the model. To determine the most efficient solver for spine models specifically, various solvers were tested on a simplified cube model of similar size to a prospective generic lumbar spine model. The solvers tested by the current study are the GPU-implemented Jacobi preconditioned conjugate gradient (PCG), Cholesky factorization for symmetric matrices (LLT solver) and for non-symmetrical matrices (LDLT solver), structurally symmetric QR factorization (SQR solver), non-symmetric QR factorization (NQR solver), and a GPU implemented Cholesky factorization (CUDA Chol). The PCG solver was written onto GPU memory following Mafi's assembled stiffness algorithm (as opposed to Mafi's element-by-element method) [2]. Likewise, the LDLT, LU, QR, and NQR solvers were implemented onto the CPU via the Intel MKL Pardiso library (Intel MKL, Intel, Santa Clara, CA), which is a suite of highly efficient and optimized solvers. Also, the CUDA Chol solver was implemented onto the GPU via NVIDIA's cuSolver library (CUDA, NVIDIA Corporation, Santa Clara, CA).

For analysis, boundary conditions involved fixing the bottom surface in all degrees of freedom while forces were applied directly on the nodes of the top surface as a pressure (total force evenly divided amongst the nodes of the top surface). Two linear isotropic materials common in spine models were tested: posterior elements (Young's modulus of 3500 MPa and Poisson's ratio of 0.25) and nucleus pulposus (Young's modulus of 1 MPa and Poisson's ratio of 0.49958). To ensure non-linear deformation in the analysis, forces of 1 MN and 600 N were applied for the posterior elements material and nucleus pulposus material, respectively. Preliminary testing of the nucleus pulposus material revealed severe ill-conditioning of the stiffness matrix; therefore, multiple material models within the cube model were not tested as

severe ill-conditioning represents the toughest test for the linear solvers. In other words, multiple materials within the same model would produce similar computational results to the nucleus pulposus material alone since both models generate stiffness matrices possessing large differences between eigenvalues. Furthermore, contact conditions were not investigated for similar reasons: contact for spine models is typically chosen as augmented Lagrange [30] resulting in a symmetric stiffness matrix; sudden stiffness changes resulting from contact often generate an ill-conditioned stiffness matrix; and contact behaviour for any specific model is a result of geometry so it can only be effectively tested on the target geometry, in which the current study uses a simplified geometry (i.e. a cube instead of a spine) to explore the research question. Therefore, considering that the most computationally difficult component of spine models is the nucleus pulposus and that ill-conditioning of the stiffness matrix is the primary concern, the proposed analyses are sufficient to determine the appropriate linear solver for prospective spine models. Accuracy and speed of the CUDA program were also tested against ANSYS by comparing average displacement of the top surface for each loading case and overall solving speed, respectively. The Newton-Raphson convergence criteria for all simulations was 1 mm for displacement and 20 N for force (both are L2 norms), and the convergence criteria for the PCG linear solver was set to 1.0e-8.

3.3 Results

See Table 3-1 for the displacement and time results and see Figure 3-3 for a speed comparison of the solvers tested, including ANSYS.

Table 3-1: Displacement and speed of the cube model for each program and solver.

Model	Solver	Absolute Displacement (mm)	Number of Iterations	Total Time (s)		
				Formulation	Solve	Total
Posterior Elements: $E = 3500\ MPa$, $\nu = 0.25$	PCG	2.9674	2	0.10	0.80	0.90
	LLT	2.9674	2	0.09	6.51	6.59
	LDLT	2.9674	2	0.08	6.63	6.72
	SQR	2.9674	2	0.12	14.70	14.82
	NQR	2.9674	2	0.12	16.17	16.30
	CUDA Chol	fail				
	ANSYS sparse	2.838	2	-	-	26.55
Nucleus Pulposus: $E = 1.0\ MPa$, $\nu = 0.49958$	PCG	5.4292	4	0.21	26.08	26.29
	LLT	5.4292	4	0.16	13.27	13.43
	LDLT	5.4292	4	0.16	13.36	13.52
	SQR	5.4292	4	0.24	30.64	30.88
	NQR	5.4292	4	0.24	29.13	29.37
	CUDA Chol	fail				
	ANSYS sparse	4.5333	4	-	-	46.31

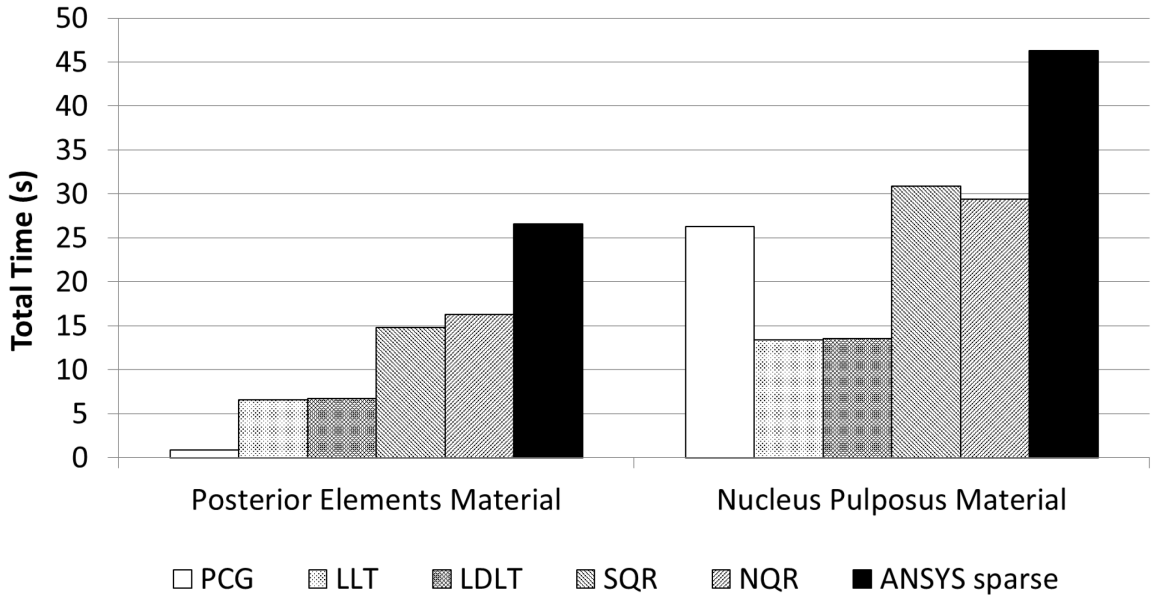

Figure 3-3: Total simulation time compared between the different solvers and ANSYS.

The accuracy between all linear solvers implemented in the CUDA program was equal, as expected. For the posterior elements material, the PCG solver exhibited the fastest time at 0.9 seconds followed by the LLT solver with a time of 6.6 seconds, while the slowest solver was the NQR solver with a time of 16.3 seconds. As seen in Table 3-1, the model of posterior elements material required two Newton-Raphson iterations to converge, while the model of nucleus pulposus material required four Newton-Raphson iterations to converge. To solve the linear system of equations at each Newton-Raphson iteration, the PCG solver required varying amounts of iterations until convergence: 562 and 735 iterations for the posterior elements material; and 12080, 5464, 12835, and 12655 iterations for the nucleus pulposus material. For the nucleus pulposus material, the LLT demonstrated the fastest time at 13.4 seconds while the SQR solver was the slowest with a time of 30.9 seconds, and the PCG solver exhibited a time of 26.29 seconds. Also, the GPU/CPU data transfer time process per iteration of the CPU-implemented linear solvers was minimal at ~0.0045 seconds.

Accuracy was compared between the CUDA program and ANSYS, see Figure 3-4. The CUDA program revealed a difference of 0.13 mm (4.6%) and 0.90 mm (19.8%) with the ANSYS program for the posterior elements and nucleus pulposus material, respectively, in the same loading scenario.

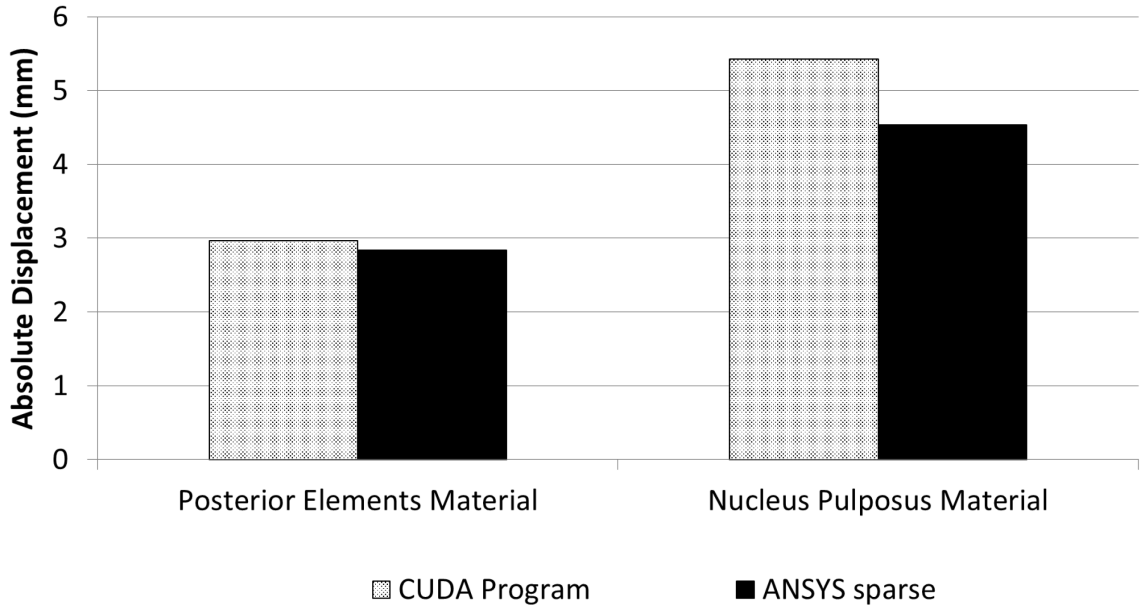

Figure 3-4: Comparison of the cube's top surface displacement between the CUDA program and ANSYS.

3.4 Discussion

A custom FE program was developed in CUDA for eventual use in real-time spine material model computing and then tested on a simplified cube model to determine the optimum linear solver plus viability for application to lumbar spine models. Although each real-time FE technique or CUDA program component of the current study was not tested individually, the overall CUDA program demonstrated substantial improvements in computation time over ANSYS with small differences (less than 1 mm) in accuracy for spine materials under substantial deformation. The mathematical basis behind the CUDA program is discussed in Section 3.4.1 and the GPU-implementation techniques, including comparisons to previous real-time FE methods, is discussed in Section 3.4.2. Choosing the Linear Solver (Section 3.4.3) presents the rationale and discussion towards the end choice based on the results. Lastly, the

use of the CUDA program for solving spine models is examined in Application to Spine Models (Section 3.4.4).

3.4.1 Finite Element Formulations

The large deformation formulations used are geometrically accurate since they account for the quadratic terms in strain displacement, although they are considerably slower than small deformation formulations. However, small deformation is not a reliable assumption for analyzing biological tissue; therefore, a large deformation formulation must be used. Theoretically, the TL and ULJ formulations are equivalent and stress/strain measures from either formulation may be calculated from the other formulation (i.e. second Piola-Kirchoff to Cauchy stress and Green-Lagrange to logarithmic strain). Still, small differences in accuracy between the CUDA program and ANSYS are noticeable during significant force application, as seen in Figure 3-4, specifically for the nucleus pulposus material under large deformation. Differences between ANSYS and the CUDA program may be attributed to: element strain enhancement in ANSYS versus no strain enhancement in the CUDA program; single precision computation in the CUDA program versus double precision in ANSYS; small errors accumulating during recalculation of the Jacobi (ULJ formulation); approximations in calculating the logarithmic strain (ULJ formulation); and calculation of the initial displacement effect (TL formulation). Other potential sources of error may exist, but further study is necessary to determine the most significant error source. Nonetheless, the accuracy difference between the CUDA program and ANSYS was under a reasonable limit of 20% considering the deformation range of ~40% exhibited by previous well-validated models [30]. Also, significant deformation in the nucleus pulposus is not expected during spine simulations, in which the current study demonstrated a deformation limit of the CUDA program with nearly-

incompressible materials that requires further investigation. Regardless, any prospective spine model implemented in the CUDA program would need to undergo validation procedures depending on the application. The results presented by the current study point toward the potential of generating a real-time FE spine simulation using the proposed methods. Therefore, the TL formulation is preferable since it requires fewer computations per element than the ULJ formulation.

Most other finite element methods, such as the arbitrary Lagrange-Euler method [132], are not well-suited for the large deformation solid mechanics required by spine models. On the other hand, meshless methods [101] provide a promising avenue of future work for real-time spine models. Meshless methods do not rely upon mesh quality so they tend to be more stable for large deformation scenarios [89]. However, meshless methods are less developed than FE methods, especially for real-time GPU applications since they are less suitable for parallelism [96]. Still, as researchers continue to develop meshless methods, they may become a promising option for future work of real-time spine simulations. Also, although they were not tested by the current study, mass-spring methods may provide substantial increases in computation speed over the current FE methods and could be a possible avenue of future study. Yet, given previous work in that area [81], mass-spring methods would likely prove too inaccurate for spine models.

3.4.2 GPU Implementation

The proposed GPU implementation established a balance between occupancy with register/shared memory usage for the given high-performance GPU, in addition to efficient usage of coalesced global and shared memory. Achieving the optimal thread block size and

making efficient use of GPU memory within the architectural limitations were critical to maximizing computation speed. Although GPU occupancy was low and not optimized, slow global memory loads were reduced significantly by efficient use of register/shared memory. Yet, the current study did not investigate the effect of using differing amounts of register/shared memory, including the balance of GPU occupancy in general, on the resulting computation speed. Further investigation is necessary to determine the optimal register/shared memory usage that increases GPU occupancy while preserving efficient memory usage. Mafi's highly efficient vector assembly algorithm was used for force vector assembly, but not for stiffness matrix assembly. Instead, the scatter method was used, in which race conditions of matrix assembly were minimized by pre-calculating an indexing array that ensured no two nodes were shared between elements of the same thread block. Although slower than Mafi's method, using the scatter method within the stiffness building kernel requires considerably less global memory, in which memory is an issue on the GPU for the given problem size, and is still sufficiently fast for the matrix building stage. In general, the current study uses the most efficient state-of-the-art techniques to attain the maximum possible computation speed. Clearly, the proposed methods are highly efficient since the matrix building process is fast with a time of 0.04 seconds per iteration for symmetric stiffness matrix building and 0.06 seconds for non-symmetric stiffness matrix building.

Another well-documented and popular method for real-time FE is the TLED algorithm. Although fast, the explicit nature of the TLED algorithm reduces its efficiency by limiting its time step. The maximum time step is dependent on the highest natural frequency of the system [10], which is influenced by both the bulk modulus of the material and the mesh quality. Higher bulk modulus and lower mesh quality of a model result in a lower maximum time step. For the

nucleus pulposus of spine models [30], the bulk modulus is in the range of $1x10^9$, which is orders of magnitude greater than the brain tissue used to test the TLED algorithm [21]. Also, spine models have irregular geometries where mesh qualities are not ideal, lowering the maximum time step further. On the other hand, implicit methods, such as sparse direct solvers coupled with full Newton-Raphson iterations, allow for considerably larger time steps for spine models making the overall solving stage faster than the TLED algorithm. Although the maximum time step for the TLED algorithm was not calculated for the current study nor was the TLED algorithm compared to the proposed algorithm, implicit methods would likely be faster than the TLED algorithm for the reasons discussed. Still, further study is required to definitively prove these claims.

Other methods have been developed for real-time GPU computation that were not explicitly tested by the current study, such as the co-rotational method [11] and the element-by-element (EbE) PCG linear solver method [2]. The co-rotational method demonstrates promise in scenarios of large deformation, but it is only valid in the low strain region since it is not a large strain formulation. On the other hand, the EbE PCG method is a fast implicit linear solver that takes advantage of the GPU and the matrix multiplication step of the conventional PCG method. Yet, it has severe limitations: the EbE PCG solver is only faster than the conventional PCG solver (including matrix assembly) under ten iterations (approximately) [2], in which a Jacobi PCG solver requires significantly more than ten iterations for a typical solve; and storing the stiffness matrix as its separate elements (as required by the EbE PCG solver) requires considerably more memory than a directly assembled matrix in csr format. Therefore, only the GPU implemented conventional PCG solver, which is still incredibly fast, was tested in the current study.

In comparison with ANSYS, the CUDA program demonstrated slightly different displacement but significant improvements in computation time. One factor contributing to the improvements is the use of single precision in the CUDA program instead of double precision as used by ANSYS. Using single precision decreases numerical accuracy but doubles (approximately) the computation speed since calculations are performed on only half the data. Considering that the overall speed-up is 3.4X for the nucleus pulposus material, other improvements in efficiency are attributed to the TL formulation, GPU implementation, and choice of linear solver. Still, significant speed-ups would be achieved if ANSYS computed the model in single precision instead of double precision.

3.4.3 Choosing the Linear Solver

Various linear solvers were tested to determine which solver would be fastest for FE spine material models (and thus prospective lumbar spine models) on the GPU. Considering that the assembled stiffness matrix was symmetric positive-definite, each of these solvers was expected to successfully solve the given model with equal accuracy. The main performance differences between the solvers would be primarily dependent on conditioning and sparsity pattern of the stiffness matrix. An ideal solver would be integrated on the GPU to eliminate time lost to data transfers between the GPU and CPU. Thus, an iterative solver, like Mafi's GPU implemented PCG solver [2], would be ideal since iterative methods exhibit better parallelism by nature, but iterative solvers are highly dependent on the condition number of the linear system of equations. In the current study, the PCG solver demonstrated real-time (less than a second) solving for the posterior elements material since the resulting stiffness matrix was well-conditioned. However, it exhibited substantially more iterations and time for the ill-conditioned stiffness matrix resulting from the nucleus pulposus (even though it was

symmetric positive definite), and thus, it would not be useful for spine models. Other iterative solvers, such as the PCG with a different preconditioner, may perform better than the PCG, but given the severe ill-conditioning of the stiffness matrix, they would likely not perform better than direct methods, although more study is required to prove this statement. On the other hand, direct methods require considerably more memory for solving and they lack parallelism, so they are not well developed for GPU implementation. Likewise, the CUDA Chol solver showed promise for the current study since it was implemented entirely on the GPU, but it did not converge for the given model size. Still, for the size of model in the current study (~124 578 DOF), transfer times between the GPU and CPU were not significant (~0.0045 seconds) and thus not a main concern. Given that the resulting stiffness matrix was symmetric positive-definite, the best performing solvers were the LLT and LDLT solvers, which demonstrated similar results, as provided by the Intel MKL Pardiso library and implemented on the CPU. Another advantage of using the LLT or LDLT solver is lower memory requirement since only the symmetric part of the matrix is stored, which is evident from the matrix build times where they were lower for the LLT and LDLT solvers than any of the other solvers. Other available solvers were not tested by the current study and may provide better performance with further study, but given the efficiency of the Intel MKL solvers, only small improvements would be expected, if any. Still, once developed, a GPU-implemented sparse direct solver may provide faster solve times for spine models. Typically for nonlinear structural analysis, most time in spine model FE analysis is spent during the linear solve phase of each Newton-Raphson iteration [112]. Therefore, substantial importance is placed on the choice of linear solver. Since it demonstrated the fastest computation times, the chosen solver for

prospective FE spine model solving, based on the current study for spine material models, is the LLT solver.

The nonlinear solver choice is important as well but given the insignificant matrix formulation times at each iteration, differences between the modified and full Newton-Raphson methods are minimal. Also, other line-searching nonlinear solvers would be inefficient considering the smoothness of the spine model stiffness curve in response to gross physiologic loading conditions.

3.4.4 Application to Prospective Spine Models

The current study was focused on developing a program that parallelizes and solves spine material models on a cube geometry as a test for application to prospective real-time FE spine models. Considering that the solid element and material formulations of typical FE lumbar spine models [30] are similar to the current study's formulations, the methods developed by the current study are well-suited and will significantly increase computation speed over conventional methods. The CUDA program readily parallelizes spine model materials for GPU implementation, and thus, it speeds up the solve time considerably. Yet, there is still a significant consideration for lumbar spine models not investigated by the current study: contact at the articulating facets. Real-time FE contact algorithms for other biomedical models have been investigated previously [3], [85], but they are not compatible within the current work given their different FE formulations. Implementation of contact conditions into the proposed CUDA program would require further exploration and development of real-time FE contact algorithms, and thus, it is an area of future work that would build upon the current study. Hence in future study, a prospective lumbar spine model without contact may be input to the CUDA

program, although its validity would depend greatly on its application. Some spine model applications require contact conditions [30], [31] while others do not [43], [45]. Upon input of a typical lumbar spine model without contact, the CUDA program will be ready to analyze the model with significant increases in computation speed and to output displacements (and rotations) of the model's response. Nonetheless, further investigation is necessary to determine the actual performance of the CUDA program for a typical lumbar spine model in comparison with conventional programs, which is the next step towards the development of a real-time FE lumbar spine model in addition to the development of contact conditions.

3.5 Conclusion

A custom CUDA program was successfully developed for spine material models using previous work in GPU implementation of FE methods along with novel techniques. In comparison to other FE formulations, the TL formulation was determined ideal for fast computation of spine material models. Some previous and some novel GPU programming techniques were successfully applied to parallelize the TL formulation, and implicit nonlinear methods proved superior to explicit and linear schemes. Results of testing on a simplified cube model demonstrated that the proposed real-time FE methods were accurate and more than three times faster than ANSYS for spine material models. Also, the LLT solver proved to be the fastest linear solver for applications involving spine material models. Although fast, the model solve is not yet at real-time speeds for the cube model. Still, the proposed methods can be readily adapted to faster GPUs with more cores and higher compute capabilities along with faster CPUs, which could approach real-time computation speeds in the future. Regardless, the CUDA program presented by the current study is ideal for the potential integration of prospective real-time FE lumbar spine models without contact, where significant increases in

computation speed are expected in future investigation. In conclusion, real-time FE techniques improved the computation speed of spine material models by not only parallelizing the element build computations, but also through a faster FE formulation algorithm and a faster linear solver.

Chapter 4 Hybrid GPU/CPU Computing of a Fast Finite Element Lumbar Spine Model

A version of this Chapter is in preparation to be submitted as:

Maeda, N.K., Boulanger, P., Carey, J.P., Hybrid GPU/CPU Computing of a Fast Finite Element Lumbar Spine Model, International Journal for Numerical Methods in Biomedical Engineering (2018).

Also, the Novel Composite Tetrahedral Element presented in Section 4.2.2 is in preparation for submission to an appropriate FE spine modelling conference in the near future.

4.1 Introduction

Spinal interventions for treating spinal disorders are becoming increasingly common [12], [15], but their outcomes reveal varied success in treating the spinal issues [13], [16], [17]. The mixed success rates could largely be explained by misunderstanding of biomechanical displacements and strains for a specific patient, in which the actual tissue displacements and strains may not match the clinician's intended outcome. During a spinal procedure, the clinician cannot effectively visualize the spine's biomechanical response to the applied forces, which may result in an adverse event or prevent a successful outcome. For example, the success of scoliosis bracing in preventing progression of the scoliotic curve is highly dependent on an orthotist's skill [16], but a real-time FE spine model would allow an orthotist to ensure spine straightness as they position the pads during bracing. As another example, a clinician could ensure that the vertebral movement does not cause considerable tissue strains (i.e. arterial dissection) during the force application of SMT. In either of these and other spinal intervention examples

(including surgery), the biomechanical feedback would need to be instant for efficacy at improving outcomes because: force application happens quickly so immediate feedback is required to adjust the applied forces; the intervention doesn't always go as planned and quick analysis is necessary to make the best decision in the moment; and less time per intervention allows the clinician to perform more interventions in a shorter period of time. Therefore, the application of a real-time simulation into clinical settings would allow clinicians to accurately visualize how the spine responds to the intervention, and through visualization, improve the outcomes of spinal interventions. However, current simulations of FE spine models require large amounts of time to compute spinal response, making them too slow for practical application. Consequently, as part of the first steps towards eventual implementation of spine models into real-time clinical scenarios, the computation speed of spine models needs to be increased.

Many problem-specific lumbar spine models have been developed and validated against experimental data [30] for various applications: biomechanical causes of back pain [41]; implant design [31]; scoliosis brace design [59]; and many more. The generic methodologies used to develop most of these problem-specific models are derived from earlier lumbar models [30], [38], [49]. Yet, no researchers have attempted to develop a lumbar spine model for real-time clinical scenarios, using generic methodologies. In Chapter 2, a generic FE lumbar spine model was developed and optimized with GPU implementation consideration. This proposed model was built, and then validated, using tetrahedral elements and relatively stable material properties for the annulus fibrosus and nucleus pulposus. Through comparison to a conventional spine model using ANSYS, the proposed model outperformed the conventional model through increased computation speed while preserving accuracy. Although GPU

implementation was a primary factor during the development of the proposed model, some further considerations are necessary. In the annulus fibrosus, the matrix and fibers share the same volume but were defined as separate elements. Element efficiency could be increased by combining the matrix and fibers in the same element. As observed in previous real-time simulation studies [85], facet contact presents a significant problem with regards to real-time computing since it often causes instability and requires smaller time steps. Accordingly, model efficiency and stability could be substantially increased by removing contact elements, although it would be at the significant cost of reduced accuracy in and applicability to many loading scenarios. While most lumbar spine model applications require facet contact for validity [30], [31], [48], [71], some models have investigated spine biomechanics without facet contact [43]–[45]. In the models without facet contact, their investigation focused on the vertebral column, in which facet contact was not necessary for model validity to their respective applications including biomechanical investigations of vertebral growth patterns in scoliosis [43], balloon kyphoplasty [45], and disc bulging [44]. Likewise, model validity did not require facet joint forces in flexion and lateral bending for some investigations [32] considering their low values in these loading conditions [30]. In each of these situations, real-time contact-less lumbar spine simulation would still provide instant valuable strain and displacement feedback to the clinician, which would allow the clinician to adjust their intervention during application, such as: adjusting the brace for scoliosis treatment and ensuring good biomechanical response to flexion and lateral bending movements for spinal surgical implants (including balloon kyphoplasty) based on their resulting placement during the surgery. Regardless, real-time simulation of contact is another considerable challenge and requires a separate investigation [85] that depends upon the formulations and results of the

current study. Contact modelling algorithms, especially real-time simulation, build upon the mathematical framework and other element formulations present within the base model. Other investigators of real-time biomechanical simulation initiated their work by first developing mathematical formulations for their respective applications [10], then adding contact considerations to their mathematical framework in subsequent studies [85]. Hence, the addition of contact algorithms is an area of future work that would build upon the current study. Regardless, the proposed model from Chapter 2 represented a step forward towards real-time clinical applications, and with some adjustments alongside GPU implementation, it could reach closer to real-time computation speeds.

Recently, some researchers have focused efforts on developing real-time FE models for specific clinical applications [2], [10], [11]. The development of CUDA by NVIDIA has revealed incredible ease and potential for the development of real-time scientific applications. Although successful for their specific applications, their methods were proven not applicable for spine models in Chapter 3. The TLED algorithm [10] was determined ineffective for nearly-incompressible materials (Poisson's ratio greater than 0.499) because its maximum time step would be inefficiently small. On the other hand, a GPU implemented preconditioned conjugate gradients solver was determined inefficient for nearly-incompressible materials owing to the ill-conditioned stiffness matrix. Other methods, such as the co-rotational algorithm [11] or statistical methods [130], were deemed insufficient in large strain or patient-specific scenarios, respectively, which are the focus of the current study. In Chapter 3, various linear solvers were tested for applicability and efficiency in application to spine material models. Through comparison of various linear solvers, the Cholesky factorization sparse direct solver exhibited the best performance for spine model applications. The investigation conducted during Chapter

3 also proved that the proposed GPU implementation of FE code using CUDA was much more efficient than CPU implemented FE programs, specifically ANSYS.

Chapter 4 focuses on combining the work completed during Chapter 2 and Chapter 3, which is applying a generic lumbar spine model (with facet contact removed) into a GPU-implemented FE program (i.e. CUDA program) in order to gain massive increases in computation speed at a reasonable computation cost. The research question that the current study addresses is: how much faster is a GPU-implemented FE lumbar spine model than conventional models in the absence of facet contact? Considering the overall goal of the current study as a step towards real-time simulation of a generic lumbar spine model, specific aims for this paper include: (1) implement the proposed lumbar spine model from Chapter 2 into the CUDA program with facet contact removed; (2) adjust the CUDA program for the proposed lumbar spine model; and (3) validate against contact-less general physiologic scenarios then test the combined CUDA model against the proposed and conventional lumbar spine models (with contact removed) implemented in ANSYS. Given the complexity of FE contact modelling and considering that the current study represents progressive work towards real-time simulation of FE lumbar spine models (including a basis for prospective contact modelling), facet contact is not a modelling consideration for the current study (it is instead a future consideration).

4.2 Methods

In Chapter 2, a lumbar spine mesh was created from a computed tomography scan (0.5 mm slices) of a full lumbar spine (L1 to L5). See Chapter 2 for more details on model development. In Chapter 3, a custom program was developed using Visual C++ (Visual Studio 2013,

Microsoft Inc., Redmond, WA) and CUDA (Version 6.5, NVIDIA Corporation, Santa Clara, CA) that conducted FE computations using the NVIDIA Titan Black GPU (NVIDIA Corporation, Santa Clara, CA) in conjunction with a commercial CPU (i5-2500 CPU, Intel Core, Intel, Santa Clara, CA). See Chapter 3 for more details on CUDA program development. For the current study, the lumbar spine mesh from Chapter 2 was adjusted and imported into the CUDA program from Chapter 3 for analysis. For comparison to conventional methods, the CUDA program's speed and accuracy was compared to ANSYS (ADPL Version 17.0, ANSYS Inc., Canonsburg, PA) implementations of the proposed and conventional models developed in Chapter 2. See Chapter 2 for details of model development for the proposed and conventional models.

4.2.1 CUDA Lumbar Spine Model

Two lumbar spine models were developed using generic methodologies in Chapter 2: a conventional model (Model 1) and a proposed model (Model 2). Model 1 was developed based on typical lumbar spine modelling methodologies from literature [30], while Model 2 improved the efficiency of Model 1 by using simpler elements and materials. Following Model 2, the lumbar spine model developed by the current study (i.e. CUDA model or Model 3) was comprised of linear tetrahedral elements for the vertebrae and intervertebral disc components along with tension-only spring elements for the ligaments, see Chapter 2 for details regarding generation of the spine model components. Following the objectives and scope of the current work, contact elements were removed from Model 1 and Model 2, and they were not implemented on Model 3. As done with the cube model from Chapter 3, the mesh of tetrahedral and ligament elements was imported directly from ANSYS, where the data underwent a pre-processing step to generate the necessary indexing arrays before analysis. See Figure 4-1 for a

flowchart of the CUDA program and see Appendix E for the CUDA program code including model preparation code. All material properties were the same for the proposed model (Model 2 in the current paper) from Table 2-1 in Chapter 2. Figure 4-2 illustrates a depiction of Model 3's mesh as taken using ANSYS.

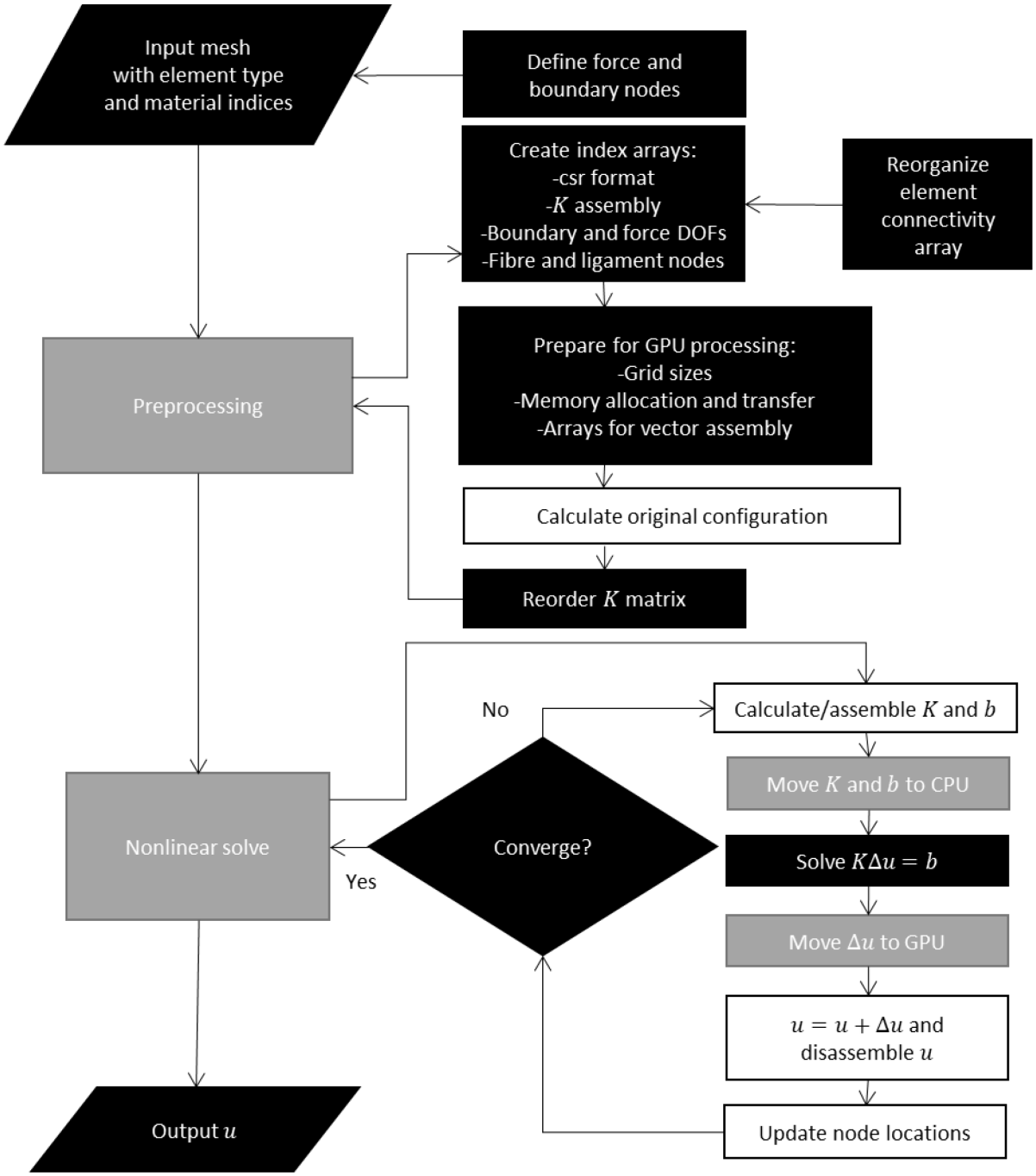

Figure 4-1: Flowchart for the CUDA program with CPU implemented linear solvers. CPU processes are colored in blue, GPU processes are run in green, processes run on both are colored in orange, and data transfer processes are colored in purple. Symbols are the same as in Figure 3-2.

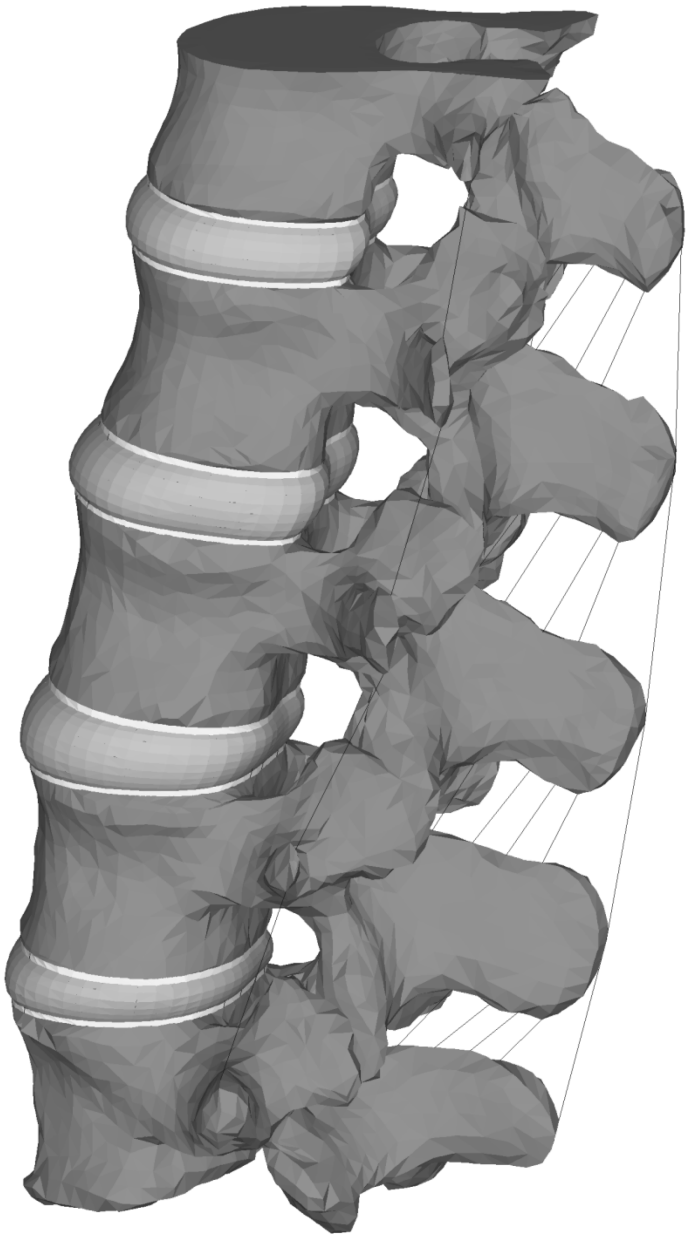

Figure 4-2: Spine model geometry used for Model 3. The mesh is from Model 2 in ANSYS but with the fiber elements removed since Model 3 used a composite element within the tetrahedral element geometry.

Two main differences exist between Model 2 and Model 3 implemented in the current study:

use of the TL formulation for all elements instead of the ULJ formulation from ANSYS; and

a novel combined-composite tetrahedral element for the annulus fibrosus instead of separate tetrahedral elements for the matrix and spring elements for the fibers. Use of the TL formulation for the spine model was presented in Chapter 3 and it was extended to formulate the tension-only spring elements, see Appendix B. Considering its novelty, the composite tetrahedral element is highlighted in Section 4.2.2, below.

The full Newton-Raphson nonlinear solver method was used to compute Model 3. Through preliminary analysis, the number of iterations to convergence was optimized by using three load steps. The convergence criteria were set at 21 mm and 200.0 N for displacement and force L2 norm convergence, respectively.

4.2.2 Composite Tetrahedral Element

For the new composite tetrahedral element type comprising the annulus fibrosus in Model 3, formulation of the composite tetrahedral element involved combining the annulus matrix and fibres into a single element type. The main assumptions behind the mathematical construction was iso-strain and that the fibre forces only acted longitudinal to the fibre direction (transverse stiffness was negligible). Hence, the strain and stress calculations were the same as for the basic tetrahedral element, but fiber stiffness was added to the material property tensor, by:

$$\boldsymbol{C} = (1 - V_F)\boldsymbol{C_B} + V_F\boldsymbol{C_F} \qquad (4-1)$$

$$\boldsymbol{S} = (1 - V_F)\boldsymbol{S_B} + V_F\boldsymbol{S_F} \qquad (4-2)$$

where $\boldsymbol{C}$ and $\boldsymbol{S}$ are the material property matrix and stress vector (second Piola-Kirchoff) of the element, $\boldsymbol{C_B}$ and $\boldsymbol{S_B}$ are the material property matrix and stress vector of the bulk annulus material, $\boldsymbol{C_F}$ and $\boldsymbol{S_F}$ are the material property matrix and stress vector of the annulus fiber

material in the element coordinate system (which coincides with the global coordinate system), respectively, and V_F is the volume fraction of the annulus fiber to bulk material.

These equations show the calculation of the material property matrix and stress vector, respectively, from the annulus bulk and fiber material via the rule of mixtures in the longitudinal direction [133]. To ensure the fibers were in the correct direction within the element coordinate system (which coincides with the global coordinate system), the material property matrix and stress vector for each annulus fiber was calculated based on the fiber force-deflection curve and fiber direction:

$$\boldsymbol{C_F} = E_F \boldsymbol{T}^T \boldsymbol{T} \qquad (4-3)$$

$$\boldsymbol{S_F} = F_F \boldsymbol{T} \qquad (4-4)$$

where E_F and F_F are the fiber stiffness and force from the fiber force deflection curve, respectively, and $\boldsymbol{T}$ is the transformation matrix from the fiber coordinate system (situated along the fiber direction) to the element coordinate system, see Appendix D for the transformation matrix.

At each Newton-Raphson iteration, the deflection of the fiber nodes (corresponding to that tetrahedral element as determined from a fiber indexing array) was used to update the tangential stiffness matrix and stress vector with reference to the fiber force-deflection curve. During solving, the fiber nodes are updated and the transformation is recalculated at each iteration since the fiber stiffness and force directions change at each iteration. Note that this composite element is numerically exact to separate tetrahedral and spring elements with one

slight difference: volume weighting of the components which is a closer representation of reality.

Although novel, the proposed composite tetrahedral element type was not tested on its own since it is simple and numerically similar to the previous formulation involving separate elements for the annulus matrix and fibres. Yet, prospective testing on this novel composite element for modelling of the IVD will be performed in the near future and the results presented at a conference, including publication in conference proceedings.

4.2.3 GPU Implementation

Parallel implementation of FE spine models was discussed, and spine material models were tested using a cube model, in Chapter 3. Based on the results, the proposed CUDA program was established as the best option for spine simulation. Also, the Cholesky factorization for positive-definite matrices, as accessible through the Intel MKL library, was chosen for solving Model 3 after testing a variety of linear matrix solvers in the CUDA program. Although the main components of the CUDA program were proven ready for implementation of Model 3, some adjustments were made to accommodate Model 3.

To account for the different numbers of DOF between each element type (twelve for the tetrahedral elements and six for the spring elements), global memory for each array was allocated as though all elements had the largest DOF, which was twelve. Thus, for the spring elements, only the first six memory spaces of the allocated twelve were used, leaving the remaining six memory spaces for each element blank, see Figure 4-3.

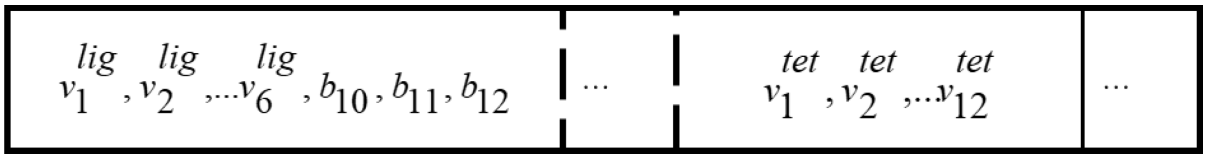

Figure 4-3: Global memory storage strategy for multiple element types where *tet* stands for tetrahedral element, *lig* stands for ligament element, and b_n represents blank memory spaces. The figure does not show the coalesced pattern for ease of explanation. Maximum DOF is 12 for tetrahedral element.

Memory storage in this manner allows for efficient global memory coalescing. Furthermore, to reduce the number of kernel launches, threads were explicitly selected within each kernel corresponding to each element and material type, and computations for that element were only applied within those selected threads. Additionally, extra indexing arrays were generated for the fiber nodes of the novel composite elements and nodal coordinates were updated for both the fiber and ligament nodes.

Therefore, with some adjustments, Model 3 was readily implemented into the CUDA program for analysis. Please refer to Chapter 3 for further details regarding GPU implementation. In correspondence with Chapter 3, the Cholesky factorization solver, as available through the Intel MKL library, was used as the linear solver.

4.2.4 Validation

Before computational speed comparison, Model 3 was validated in response to certain physiologic loading scenarios. Model 1 and Model 2 were validated against literature data [30] in Chapter 2 (as the conventional and proposed models, respectively). However, the removal of facet contact requires that the models be re-validated for the current study. Furthermore,

formulation and element differences between the proposed and CUDA model require that Model 3 undergo validation procedures as well.

In the absence of contact elements, validation of the facet joint forces was not conducted for Model 3, in addition to deformation response in axial rotation loading scenarios considering the significant presence of facet joint forces. Moreover, considering the difficulty of Model 2 in handling pure compression, validation for pure compression was not conducted as well. Hence, only RoM and L1 centroid rotation curves in response to flexion, extension, and lateral bending were validated against Dreischarf's study [30]. Model validation in this manner severely limits the applicability of Model 3, but considering scope of the current study, the proposed validation meets the specific aims. Validation simulations were conducted by fixing the inferior surface of the L5 vertebral body in all degrees of freedom while applying moment loads of 7500 N mm to the superior surface of the L1 vertebral body. For Model 1 and Model 2 in ANSYS, the moment loads were applied using a remote node that was rigidly connected to the L1 superior surface. To improve stability issues in Model 3, moment loads were applied as coupled forces across the L1 superior surface such that the sum of forces was equal to zero. Loads were applied in ten sub steps to trace the deflection curve of the spine as the load was applied. Upon validation of all three models for the current study, Model 3 was ready for computation speed comparisons to Model 2 and Model 1.

4.2.5 Comparison to ANSYS

Model 3, implemented in the custom CUDA program, was compared to Model 2 and Model 1, implemented in ANSYS, to determine whether Model 3 met the primary objective of the current study: substantial increases in speed with minimal losses in accuracy. As for model

validation, the inferior surface of the L5 vertebral body was fixed in all degrees of freedom while moment loads of 7500 N mm in flexion, extension, and lateral bending were applied to the superior surface of the L1 vertebral body. Loads were also applied as in the validation section for each model. Although the methodologies of load application are numerically different between Model 3 and the ANSYS models (Models 1 and 2), the overall validation and speed results are expected to be similar considering that sparse direct matrix solvers were used in each case. The number of sub steps for each model and each loading case were adjusted for optimal convergence (i.e. lowest number of iterations) in a preliminary analysis step. Timed simulations were run on the same workstation (CPU and GPU) to determine the accuracy and computation speed of each model. Accuracy was evaluated by comparing the L1 centroid rotation of each model. Upon collecting the results, Model 3 was compared to each ANSYS model (Model 1 and Model 2) in terms of accuracy and computation speed.

4.3 Results

4.3.1 Validation

Validation results can be seen in Figure 4-4 and Figure 4-5, in which each model falls within the range of previously well-validated models for the tested loading scenarios.

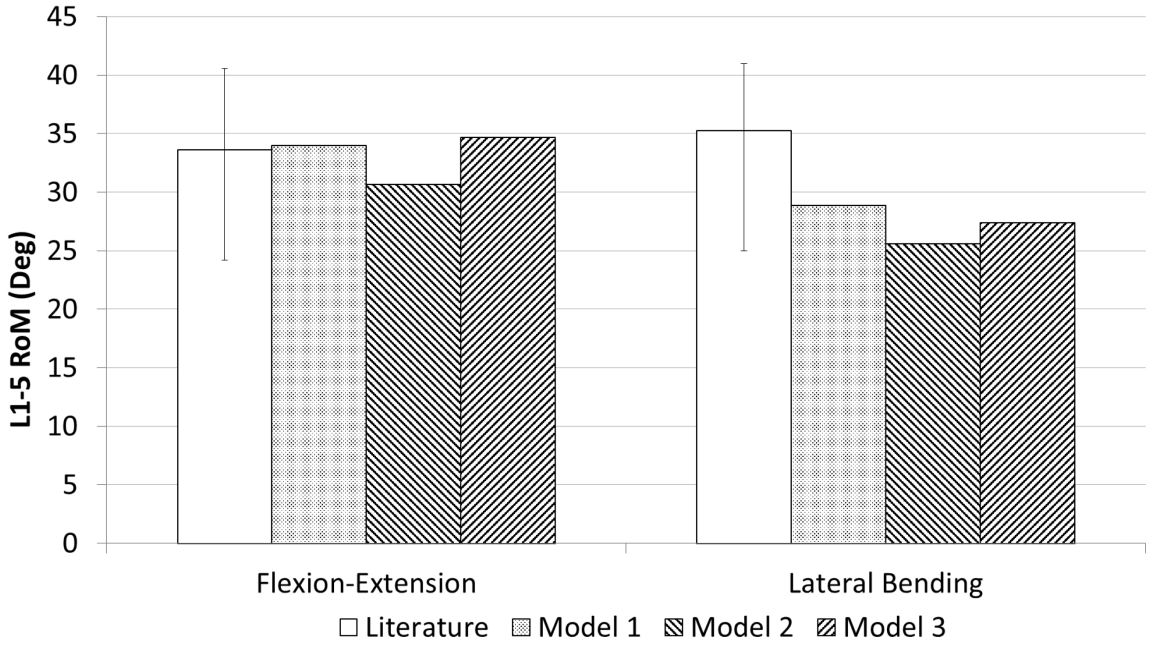

Figure 4-4: L1-5 range of motion (RoM) of the conventional and proposed models compared to previously well-validated models from literature, as published by Dreischarf [30]. The red bar represents the median of previous models while the error bars represent the range.

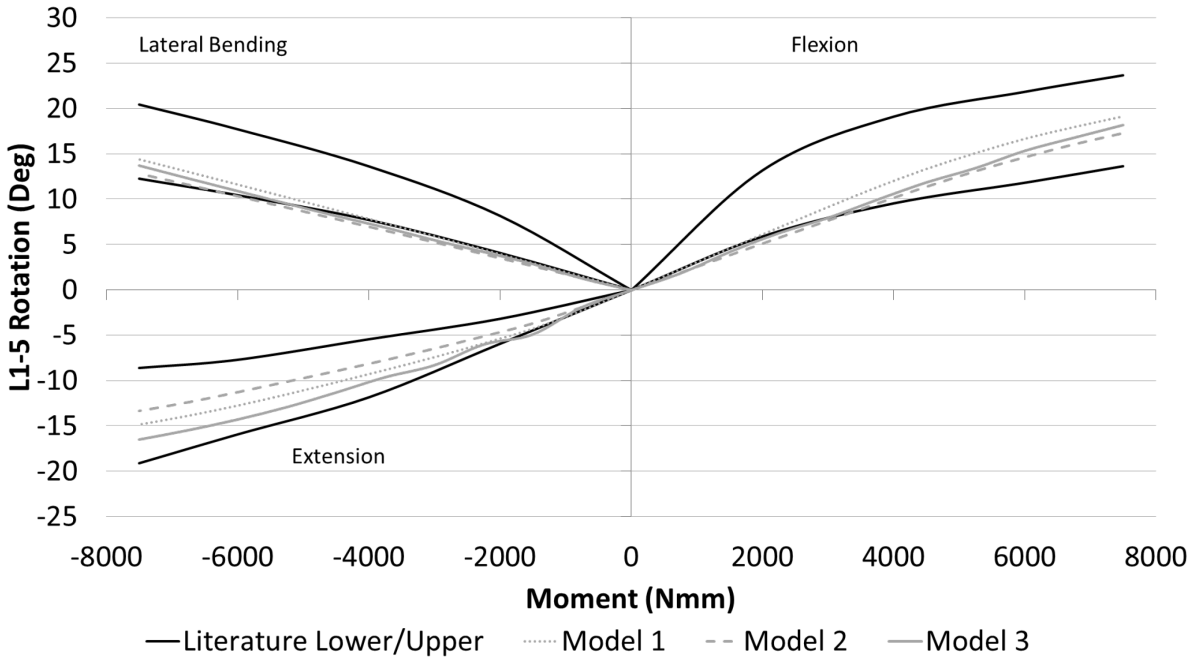

Figure 4-5: L1-5 vertebral body rotation of conventional and proposed model compared to the most and least stiff well-validated models shown by Dreischarf [30].

Therefore, all models are valid for displacement curves and RoM in flexion, extension, and lateral bending. Additionally, Model 3's displacement curves and RoM closely matched Model 2's responses.

4.3.2 Comparison

See Figure 4-6 and Figure 4-7 for computation time and L1 centroid RoM comparisons, respectively, between the CUDA and ANSYS (conventional and proposed) models. Also, see Table 4-1 for the collection of all accuracy and speed results, including a breakdown of the simulation time into time per iteration along with matrix build (formulation) time and linear matrix solve time.

Table 4-1: Speed comparison between all models

Loading Case	Model	RoM (degrees)	Substeps	Number of Iterations	Time (s)	Time/Iteration (s)
Flexion	Model 1	19.1030	4	20	282.2000	14.1100
	Model 2	17.9685	4	21	185.7700	8.8462
	Model 3	17.9536	3	13	13.2517	1.0194
Extension	Model 1	14.8591	3	16	221.8200	13.8638
	Model 2	13.8409	3	15	135.5800	9.0387
	Model 3	16.8003	3	13	13.1578	1.0121
Lateral Bending Right	Model 1	14.3400	3	16	258.7600	16.1725
	Model 2	13.7052	3	16	145.0700	9.0669
	Model 3	13.7601	4	11	11.1892	1.0172
Lateral Bending Left	Model 1	14.5210	3	14	228.8600	16.3471
	Model 2	13.9555	3	14	131.6500	9.4036
	Model 3	13.6541	5	10	10.1978	1.0198

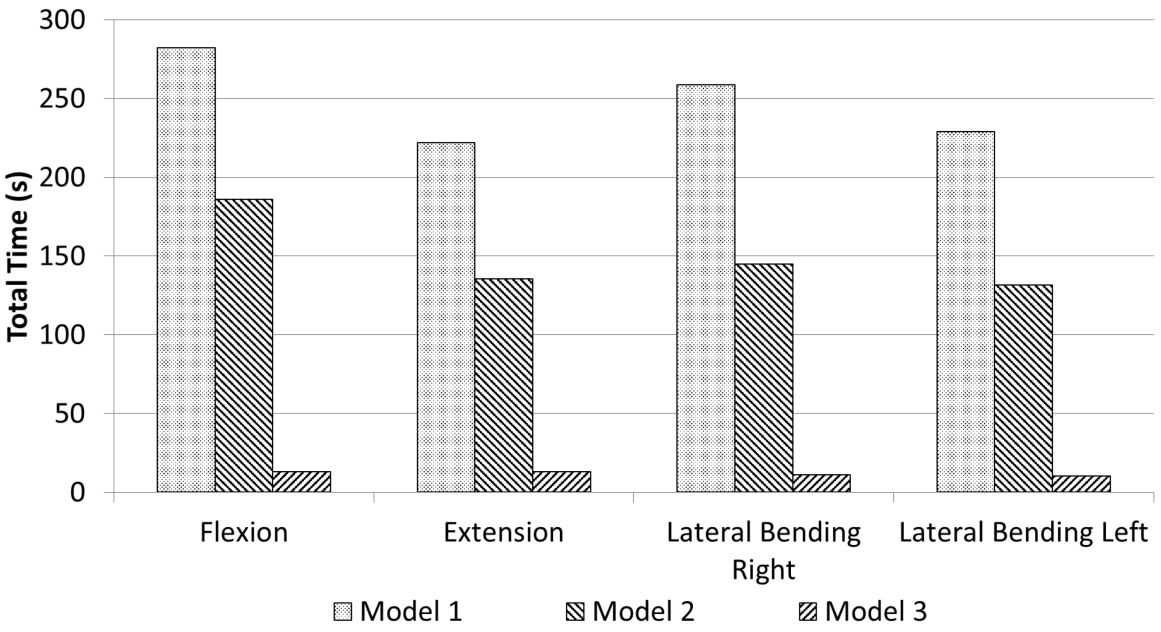

Figure 4-6: Speed comparison for the running the entire CUDA program (Model 3) including comparison to the ANSYS conventional and proposed models (Models 1 and 2).

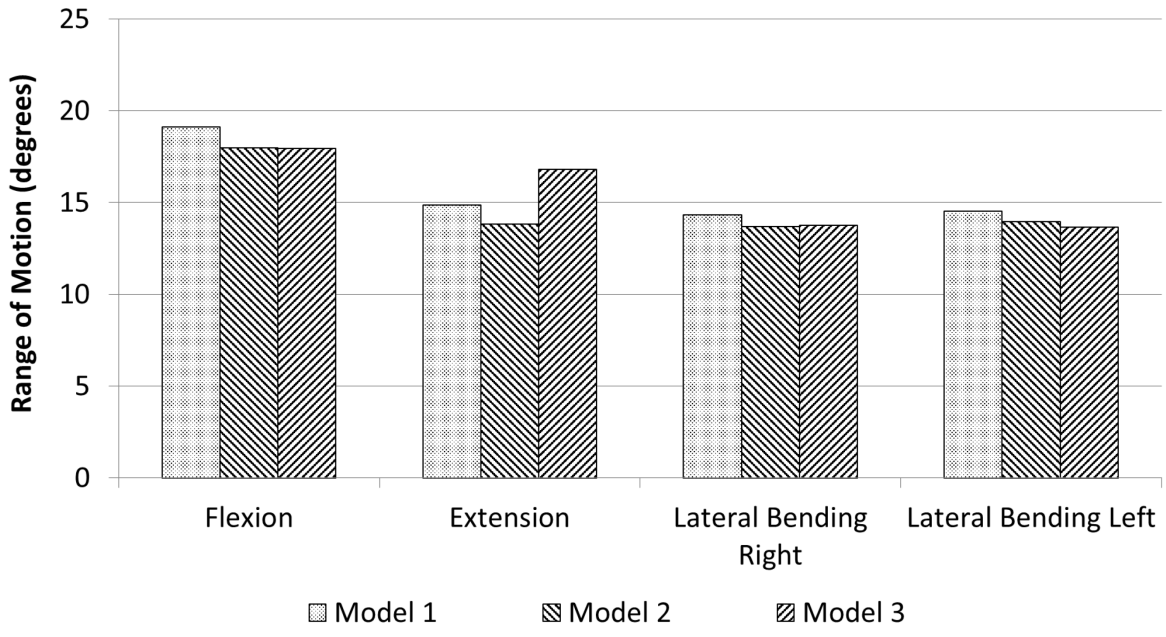

Figure 4-7: Comparison of top surface displacement between the CUDA program (Model 3) and the ANSYS conventional and proposed models (Models 1 and 2).

As with the validation stage, Model 3 and Model 2 demonstrated similar accuracy with only small differences from Model 1. As in Chapter 2, Model 2 was stiffer than Model 1, while Model 3 was slightly stiffer than Model 2. All models were still within the range of previously well-validated models in literature [30]. On the other hand, Model 3 exhibited vast increases in speed compared to either ANSYS model. Model 3 had average overall speed-ups of 20.9X and 12.5X compared to the conventional and proposed models, respectively, while the speed-ups per iteration were ~14.6X and ~8.6X. As seen in Table 4-1, Model 3 exhibited lower iterations to convergence, which explains the higher overall speed-ups compared to the speed-ups per iteration. The greatest overall speed-up was 23.1X (lateral bending) and 14.0X (flexion) and the lowest speed-up was 16.9X (extension) and 10.3X (extension), compared to the conventional and proposed models, respectively. The overall speed-up resulted from both the time per iteration, where Model 3 was exceptionally efficient, and slightly lower number of iterations for Model 3. Clearly, Model 3 demonstrated substantial efficiencies during the formulation phase.

4.4 Discussion

Feasibility of real-time FE lumbar spine simulations was tested by developing an accurate model of a lumbar spine using generic methdologies along with a custom CUDA program that optimized computation speed on a high-performance GPU. Therefore, the current study represents the state-of-the-art in technological capability and the maximum achievable computation speed with a single GPU card. Multiple GPU cards may be used to increase speed, but clinical feasibility would decrease in terms of cost. Still, as GPU technology rapidly improves, the custom CUDA program may be easily updated and used to achieve faster computation speeds. That being stated, although the accuracy compared well with the ANSYS

result, the computation speed was greatly improved at ~13 seconds (0.08 Hz) but not quite "real-time" (which would be approximately 30 Hz) [2]. Hence, the latency in computing the current lumbar spine model's response would prevent effective feedback to the clinician, among other model limitations. Therefore, the current FE lumbar spine model, including GPU implementation, does not meet feasibility criteria for real-time simulation. To address areas where speed may be improved, all stages of model and program development must be analyzed. Regardless, the current study demonstrated success towards speeding up generic FE lumbar spine models and represents a foundation from which prospective development may be based upon.

4.4.1 Finite Element Lumbar Spine Model

The current study built upon the FE lumbar spine model developed during Chapter 2. In Chapter 2, a conventional model was developed based on previous methodologies, then a proposed model was generated like Model 1 but with simpler elements and materials. In the current study, Model 3 was created from Model 2 (without facet contact) by building the same elements (without strain enhancement) and materials on top of the TL formulation instead of the ULJ formulation used by ANSYS, along with a new composite element type for the annulus fibrosus. Although differences in element types and materials existed between the models, all were constructed from the same geometry. Even though contact was removed for the current study, all models were valid for flexion and lateral bending loads, plus extension for displacement response. Accordingly, the changes of formulation and annulus fibrosus elements between the CUDA and proposed models caused minimal differences in results between the models. Likely, the most significant difference was caused by the lack of strain enhancement in Model 3, resulting in its increased stiffness response. Still, the strain enhancement was not

necessary since Model 3 demonstrated similar validation curves to both the conventional and proposed models.

Once each model was determined valid for realistic lumbar spine displacements in the tested loading scenarios, they were compared based on accuracy and speed. Considering the loading scenarios were identical to the validation stage (only less sub steps), accuracy results did not deviate significantly from the validation results. Hence, all models demonstrated similar accuracy. On the other hand, computational speed was enormously different between the models. Some of the decreased overall computation time is attributed to improved convergence resulting from the relaxed convergence criteria of Model 3, considering the speed-up per iteration was much less than the overall speed-up. Although optimized for Model 3, the relaxed convergence criteria showed that model accuracy can still be achieved for lumbar spine models under these circumstances, and therefore, relaxed convergence criteria should be used for real-time FE spine simulations. Still, Model 3 demonstrated substantial efficiencies in computation over the ANSYS models. The TL formulation requires less computation per element than the ULJ formulation, especially without strain enhancement. Also, the CUDA program used single precision instead of double precision used by ANSYS, which would increase the speed by 2X, but the overall increase was considerably more than 2X so further computational efficiencies were clearly present. Use of the Cholesky solver from the Intel MKL library added to the improved speed over ANSYS's sparse direct solver. Another source of considerable speed-up was the creation of new composite elements for the annulus fibrosus. The composite element effectively removed 11 744 spring elements from the model by combining them with the tetrahedral elements. Not only did they decrease the number of block launches (11 744 divided by 128 equals approximately 91 less blocks), but they decreased the overall amount of

computations since strain calculations for the fibers were included in the tetrahedral element rather than explicitly computed for separate spring elements. Accuracy was not lost by using the composite elements either since they are numerically similar to separate tetrahedral and spring elements. Probably the largest increase in speed-up was a result of GPU implementation. Even with all the differences between Model 3 and the proposed ANSYS model, their accuracy is nearly identical, which exhibits the power of these methods for lumbar spine models developed using generic methodologies. Numerous differences between the CUDA and proposed model contributed to the overall massive computational speed-up, but their individual contributions were not investigated in the current study. Further study focusing on the contributions of each change would determine areas of effort for future work in speeding up models similar to lumbar spine models built using generic methodologies.

Although the current study demonstrated massive increases in computation speed over conventional methods, further improvements could still be made. As presented by Shirazi-Adl [48], the vertebral bodies and posterior elements tend to act as rigid bodies. Some researchers have used OpenSim and AnyBody software to generate musculoskeletal dynamic models, in which the vertebrae were simplified as rigid bodies and the intervertebral disc was simplified by a single stiffness element. Their models have been used to investigate spinal muscle forces and vertebral displacements in orthopedics [113], ergonomics [114], [115], and rehabilitation applications [116], [118]. Such a model could be developed from the same geometry as the current study, validated, and compared to Model 3 to determine if this significant approximation would be useful. However, as considered in Chapter 2, numerical instabilities would exist in a vertebral rigid-body model and its utility would be decreased since

displacements and strains within the vertebral bodies could not be computed. Still, this could be a promising avenue of future work and requires additional study.

4.4.2 GPU Implementation

The current study used the CUDA program developed during Chapter 3 as a basis for GPU implementation of Model 3. Some features, including global memory coalescing of varying elements types and indexing arrays for the fiber and ligament nodes, were added to accommodate Model 3 into the CUDA program. In conjunction with Model 3 formulations, these features ensured that the CUDA program ran Model 3 optimally. Otherwise, the CUDA program was nearly identical to the one developed from Chapter 3.

GPU implementation of Model 3 demonstrated clear computational improvements over CPU implementation of the ANSYS models. Building the stiffness matrix and force vector at each Newton-Raphson iteration in parallel was substantially more efficient than on the CPU using ANSYS, in which Model 3 had an average formulation time of 0.05 seconds per iteration using the NVIDIA Titan Black GPU. This result shows the power of the GPU for real-time nonlinear FE computation. Furthermore, the linear solve phase of the CUDA program was highly efficient for Model 3. With a speed-up of 20.9X over Model 1, Model 3 was clearly the best candidate for prospective real-time scenarios, and it is largely a result of GPU implementation. Yet, Model 3 does not exhibit real-time solving time, with an overall average computation time of 11.9 seconds for the entire loading. Considering that most computation time is spent during the linear solve phase, improved linear solver algorithms (especially implemented on GPUs) should be focused on for future study. Regardless, as GPUs and CPUs are improved with

technological advancements, Model 3 may approach real-time speeds since the CUDA program is readily scalable to other computer workstations.

4.4.3 Application to Real-Time Clinic

Some studies have developed spine models for biomechanics research [41], [46], implant design [31], or scoliotic brace design [58], and others have developed real-time simulation techniques for various biomedical applications [2], [3], such as brain surgery [21]. The current study represents the first attempt at developing real-time simulation techniques for generic FE lumbar spine modelling methdologies towards bringing lumbar spine models into real-time clinical scenarios. For application into any clinical scenario, validation of that specific model is still required. Feasibility of real-time spine simulations was tested by developing an accurate model of a lumbar spine along with a custom CUDA program that optimized computation speed on a high-performance GPU. Hence, the current study represents the state-of-the-art in technological capability and close to the maximum achievable computation speed with a single GPU card in conjunction with a commonplace CPU. Multiple GPU cards and a faster CPU may be used to increase speed, but clinical feasibility would decrease in terms of cost. Still, as GPU technology rapidly improves, the custom CUDA program may be readily updated and used to achieve faster computation speeds. That being stated, although the accuracy compared well with the ANSYS models, the computation speed was greatly improved but not quite "real-time", since each iteration required approximately 1 second. Although a slower loading rate would be more realistic, each iteration is still too slow for the model to compute real-time spine response to the smallest loadings. Actual computation for applied loading depends upon the loading rate, where full RoM motions likely would not occur within 1 second but still faster than 12 seconds, although further study is required. In realistic clinical scenarios, physiologic

moments are typically not directly applied by the clinician, instead forces would likely be applied as a single pressure rather than coupled forces approximating a moment. An ideal real-time spine simulation would update the spine deformation response as the load is applied (depending on loading rate), but since each iteration of the current CUDA model solves in greater than one second, it is not yet feasible. For example, fast postero-anterior point loadings at rates of approximately 0.5 seconds are applied during SMT [29]. For this application, a real-time model likely would require two iterations at least, therefore each iteration must be less than 0.25 seconds. Hence, the latency in computing the spine model's response would prevent effective feedback to the clinician for that application. Longer latencies may be acceptable for other applications, such as scoliosis bracing, but faster speeds would improve the simulation considerably including the uptake of the prospective technology. Additionally, the absence of facet contact further reduces the feasibility of Model 3 for most clinical scenarios, in which a separate investigation into real-time FE contact algorithms is necessary to improve Model 3's clinical viability. Likewise, implementation into any real-time clinical scenario requires model validation for that specific application, which is a considerable challenge itself. Prospective models may be developed using the generic methodologies presented by the current study, but they would still need to cross the barrier of validation before implementation. Therefore, the current CUDA model, including GPU implementation, is not feasible for real-time simulation.

To address areas where speed may be improved, all stages of model development must undergo further investigation by building upon the results and discussions presented by the current study. For example, model building methodology could make greater approximations on the spinal tissues and GPU implementation of a sparse direct linear matrix solver could both result in real-time computation speeds for lumbar spine models.

4.4.4 Limitations and Future Work

Although the current study was successful in developing a simulation of a generic lumbar spine model that demonstrated massive increases in computation speed, Model 3 and CUDA program possess noticeable limitations and require further development before clinical application. Since it was built from Model 2, Model 3 exhibits the similar limitation of instability with compression loading; therefore, it is only applicable to scenarios where the patient is in a prone or supine position instead of standing [121]–[123]. Also, the current study only tested a lumbar spine model in the absence of contact conditions; therefore, the conclusions are useful only for contact-less lumbar spine models and model validation would still need to be conducted for each specific application. Likewise, the current model exhibits a severe limitation: it doesn't include contact conditions, hence it is not applicable to scenarios involving significant facet joint forces, such as loadings including or combined with axial rotation, which includes most clinical scenarios. Real-time simulation of contact is a considerable challenge that is highly dependent on the model geometry and FE formulations used in the base model. The prospective development of real-time facet contact algorithms can be directed from the work presented here. Therefore, facet contact and thoracic spine modelling are areas for future work building on the current study. Apart from spine modelling and program development, two other areas of research require further work to bring spine simulations into real-time clinical scenarios: fast generation and validation of patient-specific spine models for each shape and size of person that enters the clinic; and a man-machine interface to allow the clinician to interact with the spine simulation in real-time. Some research has been done towards generation of patient-specific spine models in which vertebrae were parameterized and measured [134], although much work is still required to create a clinically

useful model. On the other hand, a three-dimensional image projector using laser scanning technology that is under development has successful projected and tracked spine images onto uneven back surfaces [135], but a spine simulation has not yet been implemented into the system.

4.5 Conclusion

A custom CUDA program was developed using previous work in GPU implementation of FE methods along with some novel techniques. Plus, a generic contact-less FE lumbar spine model was developed considering GPU implementation and solved using the CUDA program. Results of the simulation determined that the real-time CUDA model (Model 3) and CUDA program were valid for displacements in flexion, extension, and lateral bending, while being massively faster than conventional models and CPU programs (i.e. ANSYS) but not fast enough for real-time spine simulations. However, the proposed methods may be readily adapted to faster CPUs and GPUs with more cores and higher compute capabilities, which could approach real-time computation speeds. Upon investigating the largest computational bottleneck, it was determined that computation speeds could also be significantly improved by developing a faster GPU implemented Cholesky factorization linear solver. Apart from computational speed, other barriers to clinical application exist: real-time facet contact, parametric modelling, and user interaction. The current study provides a basis from which further development towards real-time clinical application may be constructed, especially for real-time contact algorithms. The current study determined that the proposed spine model and GPU implementation did not meet the speed requirement for real-time simulation, although it did highlight the potential of the proposed methods and future work to achieve the speed

requirement. Regardless, the current study represents progress towards implementing spine models into real-time clinical scenarios.

Chapter 5 Discussion and Conclusion

The primary goal of the proposed work was to construct a generic contact-less FE lumbar spine simulation using real-time FE techniques. From this primary goal, three specific aims were realized: build a proposed FE lumbar spine model using a generic methodology with parallel computing in mind (Specific Aim 1); develop real-time FE techniques for spine model materials (Specific Aim 2); and apply the real-time FE techniques to the proposed FE lumbar spine model without facet contact (Specific Aim 3). Chapter 2 addressed Specific Aim 1, Chapter 3 accomplished Specific Aim 2, and Chapter 4 achieved Specific Aim 3. Upon completion of Specific Aim 3, conclusions drawn from each investigation can be compared to the primary goal of the thesis. Still, some considerations must be made regarding the limitations of the proposed work and supplementary applications that the outputs of the current study may be utilized for as well.

Through the developments of Chapter 2, the proposed FE lumbar spine model (referring to the model identification used in Chapter 2) met the prerequisite criteria: it was validated against literature data for gross physiologic loadings; it was significantly faster than conventional models; and it was more readily parallelizable than any other model investigated. Nonetheless, another model (referred to as the mixed model) that could be faster, but less parallelizable, was introduced within the discussion. The mixed model may exhibit certain advantages over the proposed model including better accuracy and stability with potentially similar computation speed. A significant limitation of the proposed model was its inability to converge for the full follower load, whereas the mixed model would probably converge for all applied loadings (yet untested). Still, the proposed model exhibited better parallelism than the mixed model

theoretically; and therefore, the current work continued with the proposed model given the prospective GPU implementation in which improved parallelism provided massive computation speed-ups. However, the mixed model has considerable potential for real-time simulation. The results from Chapter 3 and Chapter 4 showed that the parallel (GPU implemented) portion of the simulation was restricted to the element building stage between Newton-Raphson iterations, while the linear solving stage was implemented sequentially (on the CPU). Additionally, the computation time spent during the element building stage was considerably lower than expected at approximately 0.045 seconds, while the linear solving stage was approximately 1 second. Although the 0.045 second computation time represents the lowest possible element build time (given the current GPU implementation), mixed model element build times would likely be slightly longer. This observation arises from the impressive computation speed of the GPU for the tetrahedral elements of the proposed model, whereas building the reduced integration hexahedral elements requires approximately quadruple the calculations as tetrahedral elements [80]. Also, the mixed model has significantly less elements than the proposed model, which results in less kernel launches for linear matrix building. Since the number of nodes does not change between the mixed and proposed models, the computation time for linear solve stage would likely be similar, although further testing is necessary to determine if the sparsity pattern, thus matrix factorization, remains the same. Even if the computation time for element building triples for the mixed model, the overall increase in computation time would be minimal at approximately ~0.15 seconds per Newton-Raphson iteration. Furthermore, that computation time would decrease with prospective GPU hardware improvements (i.e. using the newer Titan Z instead of the

Titan Black). Regardless, further investigation and testing is necessary to prove these potential claims.

Results and discussion from Chapter 3 demonstrated that previous real-time simulation techniques were insufficient for spine material models. Thus, a novel GPU implementation was developed specifically for spine material models, which was actually a hybrid GPU/CPU implementation considering that it used both processors. The runtime of the proposed GPU implementation was broken down into two parts: element build time on the GPU and linear matrix solve time on the CPU. As shown in the previous paragraph, the element build time was exceptionally fast while the slowest part of the algorithm is the linear matrix solver; and therefore, improving the computation speed of the linear solver would greatly improve the overall computation speed. Computation speed on the GPU can readily be increased with improved GPU hardware in the form of increased number of microprocessors, considering the parallel nature of the proposed GPU implementation. On the other hand, computation speed on the CPU is only increased through faster processors considering its sequential processing nature. Hence, increasing parallelism of the linear solver would reveal quicker computation speeds than improving the solver itself. Coupled with the element build and linear solve times shown in Chapter 4, improvements to the current work should focus on two things: implementing the linear solve phase onto the GPU and/or decreasing the number of calculations required by the linear solve phase. Effective GPU implementation of linear matrix solving is a difficult challenge considering the sequential nature of linear solving. Various developers are working on implementing linear solvers onto GPUs through diverse means [136], [137]. Still, streamlining the direct sparse linear solver by analyzing the factorization steps may allow a reduction in the number of calculations. To that end, a precomputed indexing

array could be created that directly refers to the factorized version of the linear stiffness matrix instead of the unfactorized linear matrix, thereby skipping the factorization step which consumes the most time in the linear solver. Specifically, an ideal linear solver for the real-time simulation would be a streamlined version of a GPU implemented Cholesky factorization. Therefore, further investigation should focus on parallelizing and streamlining the linear solver for the simulation to approach real-time speeds.

Chapter 4 built upon the developments from Chapter 2 and Chapter 3. A novel composite element type was developed using the spine model mesh from Chapter 2 within the GPU implementation from Chapter 3. In Chapter 4, Models 1 and 2 referred to the conventional model and the proposed model solved in ANSYS, respectively, while Model 3 referred to the proposed model (with a couple alterations) solved in a custom CUDA program. As such, the annulus fibrosus was effectively represented in Model 3 (following the model identification of Chapter 4) using a fraction of the elements with no loss in accuracy. Reducing the element count, with only a small increase in calculations per element, decreases the overall computation time significantly by reducing the number of kernel launches. Yet, further study is required for the actual contribution of this novel formulation for the annulus fibrosus to the computation speed. Also, Model 3 exploited the TL formulation as opposed to the ULJ formulation used by Model 2 (Model 3 was built with the same element types and material properties as Model 2). As discussed in Chapter 3, the TL formulation requires more memory but considerably less calculations than the ULJ formulation. However, although global memory access times are lengthy and register/shared memory is limited, GPU implementation of the ULJ formulation has not been tested against the TL formulation. Plus, element build times were minimal so differences in computation time between the formulations may be small as well, although the

accuracy and stability gain would be likely negligible using the ULJ formulation. Yet, considering the stability issues associated with lumbar spine models, effective evaluation of the ULJ formulation against the TL formulation may provide helpful results and would require another investigation similar to Chapter 3.

One limitation of the current work is that only fully-deformable FE models were investigated; neither beam models [138] nor mass-spring systems [81], [139] were considered. Although significantly faster, neither of these types of models provide the accuracy or biomechanical information offered by FE models [81]. Hence, the current study explored the creation of a real-time FE model given current technology to determine real-time capability at minimal loss in accuracy. In this manner, the current work approached the problem by starting with an FE model and applied real-time FE techniques to increase computation speed. Nonetheless, an opposite approach could be attempted: start with a beam or mass-spring model that has real-time speeds and improve accuracy via theoretical means until validated. Although an attractive approach, such models by nature would reach an apex of theoretical accuracy, at which increased accuracy would involve applying FE formulations anyways. Furthermore, biomechanical information, such as strain and stress, are highly inaccurate or missing entirely for beam or mass-spring models. Using the FE method, biomechanical information and accuracy is effectively guaranteed, in which computation speed has no theoretical limit but only a practical technological limit. In other words, the FE model approach was considerably more scalable than the simplified model approach. Nonetheless, the simplified models were not tested in the current work; and therefore, further study would be necessary to determine viability for real-time lumbar spine simulation. Still, considering the promising results of the

current work, future work should build upon the FE model approach rather than the simplified model approach.

Apart from the computational successes, the current study did not include contact formulations in the facet joints, significantly limiting its applicability to scenarios where facet joint forces were not significant, such as scenarios that only involve the spinal column [55] or low facet joint forces such as flexion and lateral bending only [32]. Also, the limited compression loading further restricts the direct clinical applicability of the proposed real-time model (Model 3) to prone or supine positions, which represents many surgical spinal interventions. Considering the prevalence of facet joint and compression forces in lumbar spine biomechanics, these deficiencies severely limit the proposed real-time model. However, contact formulations were out of scope for the current thesis and they require a separate investigation in themselves, as done for other real-time biomechanical simulations approaches [85]. Considerations for prospective contact formulation include: the TL framework; using the penalty or augmented Lagrange formulations to ensure positive definiteness of the linear stiffness matrix; search methods that include triangular surfaces; optimal increments to balance speed and stability (not too small resulting in a large amount of substeps and not too large resulting in convergence issues); and others. Creation of a new contact formulation may be necessary that preserves the sparsity pattern (and thus factorization) of the linear stiffness matrix. Given the complexity of compression loading for the entire lumbar spine (via the follower load method), application of the follower load also requires further in-depth investigation to determine how to apply the increments of loading with stability, efficiency, and accuracy. For the follower loading, another novel load application method may be necessary to reduce the amount of increments essential for stable load application.

Notwithstanding, the proposed work provides a proven foundation from which to build real-time contact formulations and follower loadings upon.

Using the generic lumbar modelling methodology developed by the proposed work as a basis, application-specific models may be generated for a wide variety of applications. As done for many other application-specific models [31], [55], [122], [123], generic modelling methodologies [38], [51], [104] may be used as a guide for prospective models. In future models, validation must be conducted for each specific application: the proposed real-time model is only valid for displacement outputs in flexion and lateral bending.

Still, the proposed work is novel to the area of real-time FE spine simulation, and it provides foundational work towards applying spine simulations into clinical scenarios that would benefit from real-time biomechanical feedback. Real-time clinical scenarios would include SMT to avoid adverse events, scoliosis bracing to ensure spine straightness, spine arthroplasty surgery to ensure that the implant placement allows good physiologic movement, and other examples, all of which were out of scope for the proposed thesis but rather provided motivation for the work.

Given the results from Chapter 4 for lumbar spines in the absence of contact, the proposed real-time FE techniques presented in Chapter 3 could readily be applied to other biomechanical models. Especially, the proposed techniques can be readily applied to thoracic or cervical spine models without facet contact. Some other real-time FE techniques have been applied to other biomechanical situations, including brain surgery [88] and surgical cutting [3], but the investigation from Chapter 3 proved those FE techniques to be insufficient for spine models. Consequently, similar biomechanical situations involving bone connected to nearly-

incompressible materials, such as joint manipulations or surgery, could benefit significantly from the application of the proposed real-time FE techniques. Yet, each situation has vastly different models and constraints that require considerable investigations in themselves. Although the idea exhibits exciting potential, further study and development is necessary to integrate the proposed real-time FE techniques into other clinical situations, which would be vastly different for each scenario.

As seen in Chapter 4, the overall biomechanical results from the GPU-implemented real-time model closely resembled the purely CPU-implemented proposed model at effectively zero loss in accuracy for gross physiologic movements. In comparison with the primary objective of the thesis, the resulting model accomplishes accurate biomechanical displacements for gross physiologic movements but not at real-time speeds. Still, the proposed real-time spine model demonstrated substantial computation speed increases over previous models, in addition to improved scalability to upgraded computer hardware.

5.1 Conclusion

The primary objective of the proposed thesis was to generate a generic modelling methodology for creating a finite element lumbar spine model without facet contact that demonstrated significant increases in computation speed over previous models. Finite element models of the spine have been well investigated and validated for specific applications across numerous studies throughout literature. Likewise, a considerable body of work focusing on developing real-time finite element techniques for biomechanical models using graphics processing units has arisen recently. Yet, no current studies have combined these two areas of research to create a real-time finite element spine model. The proposed thesis explored the development and

validation (for gross physiologic movements) of a real-time finite element lumbar spine model by applying real-time finite element techniques to a fully-deformable finite element model of the lumbar spine disregarding facet contact. Through investigation, current methods were determined insufficient for developing a real-time lumbar spine model; and therefore, a novel spine modelling methodology and novel finite element techniques were proposed to fill that gap in knowledge. Accordingly, a GPU-implemented finite element model of a contact-less lumbar spine was developed that demonstrated massive improvements in computation speed at minimal loss in accuracy for gross physiologic movements. Although the resulting model did not exhibit real-time capability, the current study proved its exciting potential for real-time simulation and laid a solid foundation for further development. Specifically, the current study established that GPU implementation of the linear solver stage would greatly increase the computation speed further, including improvement of the scalability for the entire simulation to prospective GPU hardware developments. Even so, with the real-time barrier addressed through the current work, the other two barriers to clinical application, virtual haptic-visual feedback interaction and automatic patient-specific model generation (including validation), may now be addressed. Results from the current study show that should spine simulation be implemented into the clinic, a desktop computer with a moderately powerful GPU would provide the necessary computation power. Also, each patient-specific model must be comprised of a finite element mesh.

Therefore, the proposed PhD thesis delivered the following novel and significant contributions to the field of real-time lumbar spine simulations:

- Development and validation of a lumbar spine model meshed purely with linear tetrahedral elements that demonstrated improved computation speed and parallelism over conventional models. The proposed generic modelling methodology may be used to develop prospective spine models that exhibit accuracy and improved computation speed.

- GPU-implemented real-time finite element techniques that are accurate and stable for almost all biological material models, especially nearly-incompressible materials, for which previous methods were unsuccessful. Although designed and effective for spine models, the proposed real-time techniques may be useful in many clinical scenarios involving virtual biomechanical simulations.

- Formulation and implementation of a novel composite element type comprising the annulus fibrosus that significantly decreases the number of elements in the model at no loss in accuracy. This novel element can be used to improve the computation speed of prospective models involving the annulus fibrosus. The development and prospective testing of this composite element type will be disseminated at a related conference in the near future.

- Development and validation (for gross physiologic movements) of a novel finite element lumbar spine model without contact, including CUDA program, that demonstrated massive increases in computation speed over previous models. Although not real-time, the proposed work produced a solid foundation in the field of real-time spine simulations, upon which future developments will be built.

5.2 Future Work

This thesis not only made significant contributions towards implementing finite element spine models into clinical situations, but also laid the groundwork for future studies plus improvements to the current work. Based on the current work, future directions include:

- Implement facet contact into the spine simulation. Although facet joint forces are significant in the lumbar spine's response many biomechanical loadings, the current model does not include facet contact. Some previous work has attempted to integrate contact into real-time biomechanical simulations [3], [9], [85], but further work is necessary for the current spine model. Considering that axial rotation and postero-anterior forces cause noteworthy facet joint forces through spinal response, contact formulations must be added to the simulation before testing its validity for those scenarios.

- Improve the stability of the real-time lumbar model for compression loadings via follower load [68]. Allowing compression loadings, and additionally combined loadings, would enhance applicability of the real-time lumbar model to a broader range of applications involving standing positions.

- Streamline the linear matrix solve stage of the finite element method. Calculations comprising the direct sparse matrix solver must be analyzed and shortened where possible by taking advantage of characteristics unique to finite element analysis. For example, an indexing array may be generated that relates nodal calculations directly to the factorized version of the linear matrix at the point just before backwards substitution.

- Implementation of the direct linear matrix solver on the graphics processing unit. Currently, the linear solver is implemented on the central processing unit, but implementation onto the graphics processing unit would likely improve both computation speed and scalability, especially considering the computational potential of prospective graphics processing units over central processing units. For the proposed spine models of the current study, effective parallel implementation of a direct linear matrix solver is a considerable challenge and would require novel methods.

- Creation of an interactive interface between the clinician and the model. The spine simulation could be integrated into the virtual space of a visual tracking-projection system [135] or a Hololens (Microsoft). With haptic feedback included (plus some further real-time simulation developments), the clinician could effectively interact with the model and receive real-time biomechanical feedback.

- Creation and validation (for specific clinical applications) of a parametric finite element spine model that can morph to match any patient that enters the clinic. General usage of the real-time spine simulation requires automatic generation of patient-specific spine models. Considering the overall vision concerning real-time simulation, the proposed real-time spine model could be used as a basis for building the parametric model. The method for generation of patient-specific models (possibly using a parametric model as a basis) would depend upon the specific clinical application. For example, scoliosis models could potentially be generated from medical imaging scans given their availability during treatment, but anthropometric data with a statistical-type model

would be necessary for spinal manipulation therapy where medical imaging is typically not available.

- Development of thoracic and cervical spine models using the proposed spine modelling methodology and real-time finite element techniques. The current study focused only on the lumbar spine, but most clinical applications include the thoracic and cervical spine as well. The current study could be used as a guide towards developing real-time thoracic and cervical spine models.

- Test the real-time capability of a mixed model, as discussed in Chapter 2, instead of a purely tetrahedral model to improve accuracy and stability. The element building stage of the finite element analysis was minimal for the tetrahedral model and the mixed model would certainly be slower, but whether it is still sufficiently fast must be determined.

- Development of real-time biomechanical models for other clinical scenarios using the proposed real-time finite element techniques. The proposed techniques and modelling methodology could greatly improve computation speeds for other biomechanical models that are less complex than the spine geometrically and materialistically but that include near-incompressibility, such as hip or knee joint models. For example, following the methodologies used by the current study, real-time models for knee or hip arthroplasties could be developed then used clinically to improve surgeon training and surgery outcomes, in addition to potential physiotherapy applications.

Bibliography

[1] K. Miller, "Computational Biomechanics for Patient-Specific Applications," *Ann. Biomed. Eng.*, vol. 44, no. 1, pp. 1–2, 2016.

[2] R. Mafi and S. Sirouspour, "GPU-based acceleration of computations in nonlinear finite element deformation analysis," *Int. j. numer. method. biomed. eng.*, vol. 30, no. 3, pp. 365–381, Mar. 2014.

[3] H. Courtecuisse, J. Allard, P. Kerfriden, S. P. A. Bordas, S. Cotin, and C. Duriez, "Real-time simulation of contact and cutting of heterogeneous soft-tissues," *Med. Image Anal.*, vol. 18, no. 2, pp. 394–410, 2014.

[4] M. Nagendran, G. Ks, R. Aggarwal, M. Loizidou, and D. Br, "Virtual reality training for surgical trainees in laparoscopic surgery," *Cochrane Database Syst. Rev.*, no. 8, pp. 1–45, 2013.

[5] S. Cotin, H. Delingette, and N. Ayache, "Real-time elastic deformations of soft tissues for surgery simulation," *IEEE Trans. Vis. Comput. Graph.*, vol. 5, no. 1, pp. 62–73, 1999.

[6] A. Liu, F. Tendick, K. Cleary, and C. Kaufmann, "A Survey of Surgical Simulation: Applications, Technology, and Education," *Presence Teleoperators Virtual Environ.*, vol. 12, no. 6, pp. 599–614, 2003.

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

[8] H. Singh, E. J. Thomas, L. A. Petersen, and D. M. Studdert, “Medical Errors Involving Trainees,” *Arch. Intern. Med.*, vol. 167, no. 19, p. 2030, 2007.

[9] R. Mafi, “GPU-based Parallel Computing for Nonlinear Finite Element Deformation Analysis,” McMaster University, 2013.

[10] K. Miller, G. Joldes, D. Lance, and A. Wittek, “Total \uppercase{L}agrangian explicit dynamics finite element algorithm for computing soft tissue deformation,” *Commun. Numer. Methods Eng.*, vol. 23, no. 2, pp. 121–134, 2007.

[11] 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,” *Prog. Biophys. Mol. Biol.*, vol. 103, no. 2–3, pp. 159–168, 2010.

[12] F. H. Shen, D. Samartzis, and G. B. J. Andersson, “Nonsurgical management of acute and chronic low back pain.,” *J. Am. Acad. Orthop. Surg.*, vol. 14, no. 8, pp. 477–487, 2006.

[13] E. Ernst, “Adverse effects of spinal manipulation: a systematic review,” *J. R. Soc. Med.*, vol. 100, no. 7, pp. 330–338, 2007.

[14] J. J. Hebert, N. J. Stomski, S. D. French, and S. M. Rubinstein, “Serious Adverse Events and Spinal Manipulative Therapy of the Low Back Region: A Systematic Review of Cases,” *J. Manipulative Physiol. Ther.*, vol. 38, no. 9, pp. 677–691, 2015.

[15] H.-R. Weiss and M. Moramarco, “Indication for surgical treatment in patients with adolescent Idiopathic Scoliosis - a critical appraisal,” *Patient Saf. Surg.*, vol. 7, no. 1, 2013.

[16] S. Negrini, S. Minozzi, F. Zaina, N. Chockalingam, G. Tb, T. Kotwicki, T. Maruyama, M. Romano, V. Es, S. Negrini, S. Minozzi, J. Bettany-saltikov, F. Zaina, N. Chockalingam, and T. B. Grivas, "Braces for idiopathic scoliosis in adolescents," *Spine (Phila. Pa. 1976).*, vol. 35, no. 1, pp. 1–3, 2010.

[17] J. S. Smith, K. M. Fu, D. W. Polly, C. A. Sansur, S. H. Berven, P. A. Broadstone, T. J. Choma, M. J. Goytan, H. H. Noordeen, D. R. Knapp, R. A. Hart, W. F. Donaldson, J. H. Perra, O. Boachie-Adjei, and C. I. Shaffrey, "Complication rates of three common spine procedures and rates of thromboembolism following spine surgery based on 108,419 procedures: a report from the Scoliosis Research Society Morbidity and Mortality Committee," *Spine (Phila. Pa. 1976).*, vol. 35, no. 24, pp. 2140–2149, 2010.

[18] J. D. Coe, V. Arlet, W. Donaldson, S. Berven, D. S. Hanson, R. Mudiyam, J. H. Perra, and C. I. Shaffrey, "Complications in spinal fusion for adolescent idiopathic scoliosis in the new millennium. A report of the Scoliosis Research Society Morbidity and Mortality Committee," *Spine (Phila. Pa. 1976).*, vol. 31, no. 3, pp. 345–349, 2006.

[19] G. N. Kawchuk, A. Carrasco, G. Beecher, D. Goertzen, and N. Prasad, "Identification of Spinal Tissues Loaded by Manual Therapy," *Spine (Phila. Pa. 1976).*, vol. 35, no. 22, pp. 1983–1990, 2010.

[20] E. Lou, D. Hill, D. Hedden, J. Mahood, M. Moreau, and J. Raso, "An objective measurement of brace usage for the treatment of adolescent idiopathic scoliosis," *Med. Eng. Phys.*, vol. 33, no. 3, pp. 290–294, 2011.

[21] G. R. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations

on GPU - Application to neurosurgical simulation," *Comput. Methods Appl. Mech. Eng.*, vol. 199, no. 49–52, pp. 3305–3314, 2010.

[22] T. R. Oxland, "Fundamental biomechanics of the spine-What we have learned in the past 25 years and future directions," *J. Biomech.*, vol. 49, no. 6, pp. 817–832, 2016.

[23] M. A. Adams and P. Dolan, "Spine biomechanics," *J. Biomech.*, vol. 38, no. 10, pp. 1972–1983, 2005.

[24] H. J. Wilke, K. Wenger, and L. Claes, "Testing criteria for spinal implants: Recommendations for the standardization of in vitro stability testing of spinal implants," *Eur. Spine J.*, vol. 7, no. 2, pp. 148–154, 1998.

[25] M. A. Adams, D. S. McNally, and P. Dolan, "'Stress' distributions inside intervertebral discs. The effects of age and degeneration," *J. bone Jt. surgery.British Vol.*, vol. 78, no. 6, pp. 965–972, 1996.

[26] H. F. Farfan, J. W. Cossette, G. H. Robertson, R. V. Wells, and H. Kraus, "The effects of torsion on the lumbar intervertebral joints: the role of torsion in the production of disc degeneration," *J. bone Jt. surgery.American Vol.*, vol. 52, no. 3, pp. 468–497, 1970.

[27] D. M. Ikeda and S. M. McGill, "Can Altering Motions, Postures, and Loads Provide Immediate Low Back Pain Relief: A Study of 4 Cases Investigating Spine Load, Posture, and Stability," *Spine (Phila. Pa. 1976).*, vol. 37, no. 23, pp. E1469–E1475, Nov. 2012.

[28] H. J. Wilke, P. Neef, M. Caimi, T. Hoogland, and L. E. Claes, "New in vivo measurements of pressures in the intervertebral disc in daily life," *Spine (Phila. Pa.*

1976)., vol. 24, no. 8, pp. 755–762, 1999.

[29] J. J. Triano, "Biomechanics of spinal manipulative therapy," *Spine J.*, vol. 1, pp. 121–130, 2001.

[30] M. Dreischarf, T. Zander, A. Shirazi-Adl, C. M. Puttlitz, C. J. Adam, C. S. Chen, V. K. Goel, A. Kiapour, Y. H. Kim, K. M. Labus, J. P. Little, W. M. Park, Y. H. Wang, H. J. Wilke, A. Rohlmann, and H. Schmidt, "Comparison of eight published static finite element models of the intact lumbar spine: Predictive power of models improves when combined together," *J. Biomech.*, vol. 47, no. 8, pp. 1757–1766, 2014.

[31] H. Schmidt, F. Galbusera, A. Rohlmann, T. Zander, and H. J. Wilke, "Effect of multilevel lumbar disc arthroplasty on spine kinematics and facet joint loads in flexion and extension: A finite element analysis," *Eur. Spine J.*, vol. 21, no. SUPPL. 5, 2012.

[32] U. M. Ayturk and C. M. Puttlitz, "Parametric convergence sensitivity and validation of a finite element model of the human lumbar spine," *Comput. Methods Biomech. Biomed. Engin.*, vol. 14, no. 8, pp. 695–705, 2011.

[33] A. Kiapour, D. Ambati, R. W. Hoy, and V. K. Goel, "Effect of Graded Facetectomy on Biomechanics of Dynesys Dynamic Stabilization System," *Spine (Phila. Pa. 1976).*, vol. 37, no. 10, pp. E581–E589, 2012.

[34] J. P. Little, H. De Visser, M. J. Pearcy, and C. J. Adam, "Are coupled rotations in the lumbar spine largely due to the osseo-ligamentous anatomy?-A modeling study," *Comput. Methods Biomech. Biomed. Engin.*, vol. 11, no. 1, pp. 95–103, 2008.

[35] C.-L. Liu, Z.-C. Zhong, H.-W. Hsu, S.-L. Shih, S.-T. Wang, C. Hung, and C.-S. Chen,

"Effect of the cord pretension of the Dynesys dynamic stabilisation system on the biomechanics of the lumbar spine: a finite element analysis," *Eur. Spine J.*, vol. 20, no. 11, pp. 1850–1858, Nov. 2011.

[36] W. M. Park, K. Kim, and Y. H. Kim, "Effects of degenerated intervertebral discs on intersegmental rotations, intradiscal pressures, and facet joint forces of the whole lumbar spine," *Comput. Biol. Med.*, vol. 43, no. 9, pp. 1234–1240, 2013.

[37] T. Zander, A. Rohlmann, and G. Bergmann, "Influence of different artificial disc kinematics on spine biomechanics," *Clin. Biomech.*, vol. 24, no. 2, pp. 135–142, 2009.

[38] A. Shirazi-Adl, "Biomechanics of the lumbar spine in sagittal/lateral moments," *Spine*, vol. 19, no. 21. pp. 2407–2414, 1994.

[39] A. Shirazi-Adl, A. M. Ahmed, and S. C. Shrivastava, "Mechanical response of a lumbar motion segment in axial torque alone and combined with compression," *Spine (Phila. Pa. 1976).*, vol. 11, no. 9, pp. 914–927, 1986.

[40] A. Shirazi-Adl and G. Drouin, "Load-bearing role of facets in a lumbar segment under sagittal plane loadings," *J. Biomech.*, vol. 20, no. 6, pp. 601–613, 1987.

[41] A. Shirazi-Adl, "Nonlinear stress analysis of the whole lumbar spine in torsion--mechanics of facet articulation," *J. Biomech.*, vol. 27, no. 3, pp. 289–299, 1994.

[42] M. Sharma, N. a Langrana, and J. Rodriguez, "Role of ligaments and facets in lumbar spinal stability.," *Spine*, vol. 20, no. 8. pp. 887–900, 1995.

[43] L. Shi, D. Wang, M. Driscoll, I. Villemure, W. C. Chu, J. C. Cheng, and C.-E. Aubin,

"Biomechanical analysis and modeling of different vertebral growth patterns in adolescent idiopathic scoliosis and healthy subjects," *Scoliosis*, vol. 6, no. 1, p. 11, 2011.

[44] Y. M. Lu, W. C. Hutton, and V. M. Gharpuray, "Can variations in intervertebral disc height affect the mechanical function of the disc?," *Spine*, vol. 21, no. 19. p. 2208–16; discussion 2217, 1996.

[45] Y. Li and G. Lewis, "Influence of loading cycle profile and frequency on a biomechanical parameter of a model of a balloon kyphoplasty-augmented lumbar spine segment: A finite element analysis study," *Biomed. Mater. Eng.*, vol. 20, no. 6, pp. 349–359, 2010.

[46] M. El-Rich, P.-J. Arnoux, E. Wagnac, C. Brunet, and C.-E. Aubin, "Finite element investigation of the loading rate effect on the spinal load-sharing changes under impact conditions," *J. Biomech.*, vol. 42, no. 9, pp. 1252–1262, Jun. 2009.

[47] M. El-Rich, A. Shirazi-Adl, and N. Arjmand, "Muscle Activity, Internal Loads, and Stability of the Human Spine in Standing Postures: Combined Model and In Vivo Studies," *Spine (Phila. Pa. 1976).*, vol. 29, no. 23, pp. 2633–2642, 2004.

[48] A. Shirazi-Adl, "Analysis of role of bone compliance on mechanics of a lumbar motion segment," *J. Biomech. Eng.*, vol. 116, no. 4, pp. 408–412, 1994.

[49] H. Schmidt, F. Heuer, U. Simon, A. Kettler, A. Rohlmann, L. Claes, and H. J. Wilke, "Application of a new calibration method for a three-dimensional finite element model of a human lumbar annulus fibrosus," *Clin. Biomech.*, vol. 21, no. 4, pp. 337–344, 2006.

[50] V. K. Goel, B. T. Monroe, L. G. Gilbertson, and P. Brinckmann, "Interlaminar shear

stresses and laminae separation in a disc. Finite element analysis of the L3-L4 motion segment subjected to axial compressive loads.," *Spine*, vol. 20, no. 6. pp. 689–698, 1995.

[51] H. Schmidt, F. Heuer, J. Drumm, Z. Klezl, L. Claes, and H. J. Wilke, "Application of a calibration method provides more realistic results for a finite element model of a lumbar spinal segment," *Clin. Biomech.*, vol. 22, no. 4, pp. 377–384, 2007.

[52] Y. M. M. Lu, W. C. C. Hutton, and V. M. M. Gharpuray, "Do bending, twisting, and diurnal fluid changes in the disc affect the propensity to prolapse? A viscoelastic finite element model.," *Spine*, vol. 21, no. 22. pp. 2570–9, 1996.

[53] L. Li, T. Shen, and Y.-K. Li, "A Finite Element Analysis of Stress Distribution and Disk Displacement in Response to Lumbar Rotation Manipulation in the Sitting and Side-Lying Positions," *J. Manipulative Physiol. Ther.*, vol. 40, no. 8, pp. 580–586, 2017.

[54] A. Polikeit, S. J. Ferguson, L. P. Nolte, and T. E. Orr, "Factors influencing stresses in the lumbar spine after the insertion of intervertebral cages: Finite element analysis," *Eur. Spine J.*, vol. 12, no. 4, pp. 413–420, 2003.

[55] P.-L. Sylvestre, I. Villemure, and C.-É. Aubin, "Finite element modeling of the growth plate in a detailed spine model," *Med. Biol. Eng. Comput.*, vol. 45, no. 10, pp. 977–988, 2007.

[56] J. Fok, S. Adeeb, and J. Carey, "FEM Simulation of Non-Progressive Growth from Asymmetric Loading and Vicious Cycle Theory: Scoliosis Study Proof of Concept," *Open Biomed. Eng. J.*, vol. 4, no. 1, pp. 162–169, 2010.

[57] C. K. Lee, Y. E. Kim, C. S. Lee, Y. M. Hong, J. M. Jung, and V. K. Goel, "Impact

response of the intervertebral disc in a finite-element model.," *Spine (Phila. Pa. 1976).*, vol. 25, no. 19, pp. 2431–2439, 2000.

[58] N. Cobetto, C. E. Aubin, J. Clin, S. Le May, F. Desbiens-Blais, H. Labelle, and S. Parent, "Braces optimized with computer-assisted design and simulations are lighter, more comfortable, and more efficient than plaster-cast braces for the treatment of adolescent idiopathic scoliosis," *Spine Deform.*, vol. 2, no. 4, pp. 276–284, 2014.

[59] J. Clin, C. E. Aubin, S. Parent, A. Sangole, and H. Labelle, "Comparison of the biomechanical 3D efficiency of different brace designs for the treatment of scoliosis using a finite element model," *Eur. Spine J.*, vol. 19, no. 7, pp. 1169–1178, 2010.

[60] K. T. Huynh, I. Gibson, and Z. Gao, "Development of a Detailed Human Spine Model with Haptic Interface," *Haptics Render. Appl.*, pp. 165–194, 2012.

[61] G. Zhan and I. Gibson, "A finite element simulator for spine orthopaedics with haptic interface," *Virtual Phys. Prototyp.*, vol. 3, no. 3, pp. 141–149, 2008.

[62] A. H. Dicko, N. Tong-Yette, B. Gilles, F. Faure, and O. Palombi, "Construction and Validation of a Hybrid Lumbar Spine Model For the Fast Evaluation of Intradiscal Pressure and Mobility," *Int. Sci. Index, Med. Heal. Sci.*, vol. 9, no. 2, pp. 134–145, 2015.

[63] "Spine Image." [Online]. Available: https://justmeint.wordpress.com/2011/07/20/looking-for-a-new-backbone/spine/.

[64] J. Noailly, H. J. Wilke, J. A. Planell, and D. Lacroix, "How does the geometry affect the internal biomechanics of a lumbar spine bi-segment finite element model? Consequences on the validation process," *J. Biomech.*, vol. 40, no. 11, pp. 2414–2425,

2007.

[65] A. C. Jones and R. K. Wilcox, “Finite element analysis of the spine: Towards a framework of verification, validation and sensitivity analysis,” *Med. Eng. Phys.*, vol. 30, no. 10, pp. 1287–1304, 2008.

[66] J. Noailly, D. Lacroix, and J. A. Planell, “Finite Element Study of a Novel Intervertebral Disc Substitute,” *Spine (Phila. Pa. 1976).*, vol. 30, no. 20, pp. 2257–2264, 2005.

[67] C.-S. Kuo, H.-T. Hu, R.-M. Lin, K.-Y. Huang, P.-C. Lin, Z.-C. Zhong, and M.-L. Hseih, “Biomechanical analysis of the lumbar spine on facet joint force and intradiscal pressure--a finite element study.,” *BMC Musculoskelet. Disord.*, vol. 11, p. 151, 2010.

[68] A. Rohlmann, T. Zander, M. Rao, and G. Bergmann, “Applying a follower load delivers realistic results for simulating standing,” *J. Biomech.*, vol. 42, no. 10, pp. 1520–1526, 2009.

[69] R. Eberlein∗, G. A. Holzapfel†, and C. A. J. Schulze-Bauer‡, “An Anisotropic Model for Annulus Tissue and Enhanced Finite Element Analyses of Intact Lumbar Disc Bodies,” *Comput. Methods Biomech. Biomed. Engin.*, vol. 4, no. 3, pp. 209–229, Jan. 2001.

[70] G. A. Holzapfel and M. Stadler, “Role of facet curvature for accurate vertebral facet load analysis,” *Eur. Spine J.*, vol. 15, no. 6, pp. 849–856, 2006.

[71] E. Wagnac, P.-J. Arnoux, A. Garo, and C.-E. Aubin, “Finite element analysis of the influence of loading rate on a model of the full lumbar spine under dynamic loading conditions,” *Med. Biol. Eng. Comput.*, vol. 50, no. 9, pp. 903–915, 2012.

[72] B. Weisse, A. K. Aiyangar, C. Affolter, R. Gander, G. P. Terrasi, and H. Ploeg, "Determination of the translational and rotational stiffnesses of an L4-L5 functional spinal unit using a specimen-specific finite element model," *J. Mech. Behav. Biomed. Mater.*, vol. 13, pp. 45–61, 2012.

[73] L.-X. Guo, Z.-W. Wang, Y.-M. Zhang, K.-K. Lee, E.-C. Teo, H. Li, and B.-C. Wen, "Material property sensitivity analysis on resonant frequency characteristics of the human spine.," *J. Appl. Biomech.*, vol. 25, no. 1, pp. 64–72, 2009.

[74] F. Lavaste, W. Skalli, S. Robin, R. Roy-Camille, and C. Mazel, "Three-dimensional geometrical and mechanical modelling of the lumbar spine," *J. Biomech.*, vol. 25, no. 10, pp. 1153–1164, 1992.

[75] E. C. Teo, K. K. Lee, H. W. Ng, T. X. Qiu, and K. Yang, "Determination of Load Transmission and Contact Force At Facet Joints of L2–L3 Motion Segment Using Fe Method," *J. Musculoskelet. Res.*, vol. 07, no. 02, pp. 97–109, 2003.

[76] F. A. Pintar, N. Yoganandan, T. Myers, A. Elhagediab, and A. Sances, "Biomechanical properties of human lumbar spine ligaments," *J. Biomech.*, vol. 25, no. 11, pp. 1351–1356, 1992.

[77] D. C. Wilson, C. A. Niosi, Q. A. Zhu, T. R. Oxland, and D. R. Wilson, "Accuracy and repeatability of a new method for measuring facet loads in the lumbar spine," *J. Biomech.*, vol. 39, no. 2, pp. 348–353, 2006.

[78] M. J. PEARCY and S. B. TIBREWAL, "Axial Rotation and Lateral Bending in the Normal Lumbar Spine Measured by Three-Dimensional Radiography," *Spine*, vol. 9,

no. 6. pp. 582–587, 1984.

[79] "CUDA C PROGRAMMING GUIDE," 2012. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Accessed: 25-Nov-2017].

[80] K.-J. Bathe and K.-J. Bathe, *Finite element procedures*. Upper Saddle River, N.J. : Prentice Hall, c1996, 1996.

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

[82] S. F. F. Gibson, "3D ChainMail: A fast algorithm for deforming volumetric objects," in *Proc. Symposium on Interactive 3D Graphics*, 1997, pp. 149–154.

[83] A. Rodríguez, A. León, and G. Arroyo, "Parallel deformation of heterogeneous ChainMail models: Application to interactive deformation of large medical volumes," *Comput. Biol. Med.*, vol. 79, no. October, pp. 222–232, 2016.

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

[85] G. R. Joldes, A. Wittek, and K. Miller, "Suite of finite element algorithms for accurate computation of soft tissue deformation for surgical simulation," *Med. Image Anal.*, vol. 13, no. 6, pp. 912–919, 2009.

[86] G. R. Joldes, A. Wittek, and K. Miller, "Non-locking tetrahedral finite element for surgical simulation," *Commun. Numer. Methods Eng.*, vol. 25, no. 7, pp. 827–836, Jul. 2009.

[87] G. R. Joldes, A. Wittek, and K. Miller, "An efficient hourglass control implementation for the uniform strain hexahedron using the Total Lagrangian formulation," *Commun. Numer. Methods Eng.*, vol. 24, no. 11, pp. 1315–1323, Jul. 2008.

[88] G. R. Joldes, A. Wittek, and K. Miller, "Computation of intra-operative brain shift using dynamic relaxation," *Comput. Methods Appl. Mech. Eng.*, vol. 198, no. 41–44, pp. 3313–3320, 2009.

[89] K. Miller, A. Horton, G. R. Joldes, and A. Wittek, "Beyond finite elements: A comprehensive, patient-specific neurosurgical simulation utilizing a meshless method," *J. Biomech.*, vol. 45, no. 15, pp. 2698–2701, 2012.

[90] V. Strbac, D. M. Pierce, J. Vander Sloten, and N. Famaey, "GPGPU-based explicit finite element computations for applications in biomechanics: the performance of material models, element technologies, and hardware generations," *Comput. Methods Biomech. Biomed. Engin.*, vol. 20, no. 16, pp. 1643–1657, 2017.

[91] X. Jin, G. R. Joldes, K. Miller, K. H. Yang, and A. Wittek, "Meshless algorithm for soft tissue cutting in surgical simulation," *Comput. Methods Biomech. Biomed. Engin.*, vol. 17, no. 7, pp. 800–811, 2014.

[92] G. Y. Zhang, A. Wittek, G. R. Joldes, X. Jin, and K. Miller, "A three-dimensional nonlinear meshfree algorithm for simulating mechanical responses of soft tissue," *Eng.*

Anal. Bound. Elem., vol. 42, pp. 60–66, 2014.

[93] X. Liu, R. Wang, Y. Li, and D. Song, "Deformation of soft tissue and force feedback using the smoothed particle hydrodynamics," *Comput. Math. Methods Med.*, vol. 2015, 2015.

[94] S. F. Johnsen, Z. A. Taylor, M. J. Clarkson, J. Hipwell, M. Modat, B. Eiben, L. Han, Y. Hu, T. Mertzanidou, D. J. Hawkes, and S. Ourselin, "NiftySim: A GPU-based nonlinear finite element package for simulation of soft tissue biomechanics," *Int. J. Comput. Assist. Radiol. Surg.*, vol. 10, no. 7, pp. 1077–1095, 2015.

[95] Z. a Taylor, M. Cheng, and S. Ourselin, "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units.," *IEEE Trans. Med. Imaging*, vol. 27, no. 5, pp. 650–663, 2008.

[96] A. Karatarakis, P. Metsis, and M. Papadrakakis, "GPU-acceleration of stiffness matrix calculation and efficient initialization of EFG meshless methods," *Comput. Methods Appl. Mech. Eng.*, vol. 258, pp. 63–80, 2013.

[97] A. Karatarakis, P. Karakitsios, and M. Papadrakakis, "GPU accelerated computation of the isogeometric analysis stiffness matrix," *Comput. Methods Appl. Mech. Eng.*, vol. 269, pp. 334–355, 2014.

[98] A. Rodríguez, A. León, G. Arroyo, and J. M. Mantas, "SP-ChainMail: a GPU-based sparse parallel ChainMail algorithm for deforming medical volumes," *J. Supercomput.*, vol. 71, no. 9, pp. 3482–3499, 2015.

[99] S. Niroomandi, D. González, I. Alfaro, F. Bordeu, A. Leygue, E. Cueto, and F. Chinesta,

"Real-time simulation of biological soft tissues: a PGD approach: REAL-TIME SIMULATION OF BIOLOGICAL SOFT TISSUES: A PGD APPROACH," *Int. j. numer. method. biomed. eng.*, vol. 29, no. 5, pp. 586–600, May 2013.

[100] C. Felippa, "A Systematic Approach to the Element-Independent Corotational Dynamics of Finite Elements," University of Colorado, Boulder, Colorado, USA, CU-CAS-00-03, Jan. 2000.

[101] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl, "Meshless methods: an overview and recent developments," *Appl. Mech. Eng.*, vol. 139, no. 1–4, pp. 3–47, 1996.

[102] A. Shirazi-Adl and G. Drouin, "Nonlinear gross response analysis of a lumbar motion segment in combined sagittal loadings," *J. Biomech. Eng.*, vol. 110, no. 3, pp. 216–222, 1988.

[103] V. K. Goel, Y. E. Kim, T. H. Lim, and J. N. Weinstein, "An analytical investigation of the mechanics of spinal instrumentation.," *Spine*, vol. 13, no. 9. pp. 1003–11, 1988.

[104] M. El-Rich, P. J. Arnoux, E. Wagnac, C. Brunet, and C. E. Aubin, "Finite element investigation of the loading rate effect on the spinal load-sharing changes under impact conditions," *J. Biomech.*, vol. 42, no. 9, pp. 1252–1262, 2009.

[105] A. Shirazi-Adl, S. Sadouk, M. Parnianpour, D. Pop, and M. El-Rich, "Muscle force evaluation and the role of posture in human lumbar spine under compression," *Eur. Spine J.*, vol. 11, no. 6, pp. 519–526, 2002.

[106] C. J. Siepe, F. Heider, K. Wiechert, W. Hitzl, B. Ishak, and M. H. Mayer, "Mid- to long-

term results of total lumbar disc replacement: A prospective analysis with 5- to 10-year follow-up," *Spine J.*, vol. 14, no. 8, pp. 1417–1431, 2014.

[107] H. R. Malone, O. N. Syed, M. S. Downes, A. L. D'ambrosio, D. O. Quest, and M. G. Kaiser, "Simulation in neurosurgery: A review of computer-based simulation environments and their surgical applications," *Neurosurgery*, vol. 67, no. 4, pp. 1105–1116, 2010.

[108] M. Dreischarf, A. Rohlmann, G. Bergmann, and T. Zander, "Optimised in vitro applicable loads for the simulation of lateral bending in the lumbar spine," *Med. Eng. Phys.*, vol. 34, no. 6, pp. 777–780, 2012.

[109] A. Shirazi-Adl and M. Parnianpour, "Role of posture in mechanics of the lumbar spine in compression.," *Journal of spinal disorders*, vol. 9, no. 4. pp. 277–86, 1996.

[110] A. Shirazi-Adl, "Finite-element evaluation of contact loads on facets of an L2-L3 lumbar segment in complex loads.," *Spine*, vol. 16. pp. 533–541, 1991.

[111] J. Dompierre, P. Labbé, M.-G. Vallet, and R. Camarero, "How to Subdivide Pyramids, Prisms and Hexahedra into Tetrahedra," *8th Int. Meshing Roundtable*, pp. 195--204, 1999.

[112] "Mechanical APDL Theory Reference." [Online]. Available: https://www.sharcnet.ca/Software/Ansys/16.2.3/en-us/help/ans_thry/ansys.theory.html. [Accessed: 25-Nov-2017].

[113] K. S. Han, K. Kim, W. M. Park, D. S. Lim, and Y. H. Kim, "Effect of centers of rotation on spinal loads and muscle forces in total disk replacement of lumbar spine," *Proc. Inst.*

Mech. Eng. Part H J. Eng. Med., vol. 227, no. 5, pp. 543–550, 2013.

[114] X. Guan, L. Ji, R. Wang, and W. Huang, "Optimization of an unpowered energy-stored exoskeleton for patients with spinal cord injury," *Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc. EMBS*, vol. 2016–Octob, pp. 5030–5033, 2016.

[115] A. D. Nimbarte, J. A. Sivak-Callcott, M. Zreiqat, and M. Chapman, "Neck Postures and Cervical Spine Loading Among Microsurgeons Operating with Loupes and Headlamp," *IIE Trans. Occup. Ergon. Hum. Factors*, vol. 1, no. 4, pp. 215–223, 2013.

[116] B. Jia, S. Kim, and M. A. Nussbaum, "An EMG-based model to estimate lumbar muscle forces and spinal loads during complex, high-effort tasks: Development and application to residential construction using prefabricated walls," *Int. J. Ind. Ergon.*, vol. 41, no. 5, pp. 437–446, 2011.

[117] M. Christophy, N. A. F. Senan, J. C. Lotz, and O. M. O'Reilly, "A Musculoskeletal model for the lumbar spine," *Biomech. Model. Mechanobiol.*, vol. 11, no. 1–2, pp. 19–34, 2012.

[118] K. S. Han, H. Do Nam, and K. Kim, "Core muscle strengthening effect during spine stabilization exercise," *J. Electr. Eng. Technol.*, vol. 10, no. 6, pp. 2413–2419, 2015.

[119] F. Marchand and A. M. Ahmed, "Investigation of the laminate structure of lumbar disc anulus fibrosus," *Spine*, vol. 15, no. 5. pp. 402–410, 1990.

[120] G. H. Golub and C. F. Van Loan, *Matrix Computations*. JHU Press, 1996.

[121] C. R. Driscoll, C. I. Aubin, F. Canet, H. Labelle, and J. Dansereau, "Impact of prone

surgical positioning on the scoliotic spine," *J. Spinal Disord. Tech.*, vol. 25, no. 3, pp. 173–181, 2012.

[122] M. Lee, D. W. Kelly, and G. P. Steven, "A model of spine, ribcage and pelvic responses to a specific lumbar manipulative force in relaxed subjects," *J. Biomech.*, vol. 28, no. 11, pp. 1403–1408, 1995.

[123] T. S. Keller and C. J. Colloca, "A rigid body model of the dynamic posteroanterior motion response of the human lumbar spine," *J. Manipulative Physiol. Ther.*, vol. 25, no. 8, pp. 485–496, 2002.

[124] J. P. Little and C. J. Adam, "Geometric sensitivity of patient-specific finite element models of the spine to variability in user-selected anatomical landmarks," *Comput. Methods Biomech. Biomed. Engin.*, vol. 18, no. 6, pp. 676–688, 2015.

[125] F. Niemeyer, H. J. Wilke, and H. Schmidt, "Geometry strongly influences the response of numerical models of the lumbar spine-A probabilistic finite element analysis," *J. Biomech.*, vol. 45, no. 8, pp. 1414–1423, 2012.

[126] I. Villemure, C. E. Aubin, J. Dansereau, and H. Labelle, "Simulation of progressive deformities in adolescent idiopathic scoliosis using a biomechanical model integrating vertebral growth modulation," *J. Biomech. Eng.*, vol. 124, no. 6, pp. 784–790, 2002.

[127] J. Gasco, A. Patel, J. Ortega-Barnett, D. Branch, S. Desai, Y. F. Kuo, C. Luciano, S. Rizzi, P. Kania, M. Matuyauskas, P. Banerjee, and B. Z. Roitberg, "Virtual reality spine surgery simulation: an empirical study of its usefulness," *Neurol. Res.*, vol. 36, no. 11, pp. 968–973, 2014.

[128] J. D. Owens, D. Luebke, N. Govindraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, "A Survey of General Purpose Computation on Graphics Hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2006.

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

[130] S. R. Mousavi, I. Khalaji, A. S. Naini, K. Raahemifar, and A. Samani, "Statistical finite element method for real-time tissue mechanics analysis," *Comput. Methods Biomech. Biomed. Engin.*, vol. 15, no. 6, pp. 595–608, 2012.

[131] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *J. Parallel Distrib. Comput.*, vol. 69, no. 5, pp. 451–460, 2009.

[132] C. W. Hirt, A. A. Amsden, and J. L. Cook, "An Arbitrary Lagrangian-Eulerian computing method for all flow speeds," *J. Comput. Phys.*, vol. 135, no. 2, pp. 203–216, 1997.

[133] B. K. O. Cheung and J. P. Carey, "Micromechanics for braided composites," in *Handbook of Advances in Braided Composite Materials: Theory, Production, Testing and Applications*, 2016, pp. 239–257.

[134] C. Molter, N. Maeda, and J. Carey, "Development of a Parametrized Lumbar Spine Model," in *International Symposium on Computer Methods in Biomechanics and Biomedical Engineering*, 2015, p. 41.

[135] K. Punithakumar, A. R. Hareendranathan, A. McNulty, M. Biamonte, A. He, M. Noga, P. Boulanger, and H. Becher, "Multiview 3-D Echocardiography Fusion with Breath-Hold Position Tracking Using an Optical Tracking System," *Ultrasound Med. Biol.*, vol. 42, no. 8, pp. 1998–2009, 2016.

[136] H. AbouEisha, K. Jopek, B. Medygrał, M. Moshkov, S. Nosek, A. Paszyńska, M. Paszyński, and K. Pingali, "Hybrid Direct and Iterative Solver with Library of Multi-criteria Optimal Orderings for h Adaptive Finite Element Method Computations," *Procedia Comput. Sci.*, vol. 80, no. December, pp. 865–874, 2016.

[137] J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, "A Sparse Symmetric Indefinite Direct Solver for GPU Architectures," *ACM Trans. Math. Softw.*, vol. 42, no. 1, 2016.

[138] M. El-Rich and A. Shirazi-Adl, "Effect of load position on muscle forces, internal loads and stability of the human spine in upright postures," *Comput. Methods Biomech. Biomed. Engin.*, vol. 8, no. 6, pp. 359–368, 2005.

[139] K. T. Huynh, Z. Gao, I. Gibson, and W. F. Lu, "Haptically integrated simulation of a finite element model of thoracolumbar spine combining offline biomechanical response analysis of intervertebral discs," *CAD Comput. Aided Des.*, vol. 42, no. 12, pp. 1151–1166, 2010.

[140] C. Felippa, "Nonlinear Finite Element Methods." [Online]. Available: https://www.colorado.edu/engineering/cas/courses.d/NFEM.d/Home.html.

[141] T. Sussman and K.-J. Bathe, "A finite element formulation for nonlinear incompressible elastic and inelastic analysis," *Comput. Struct.*, vol. 26, no. 1–2, pp. 357–409, 1987.

Appendix A Derivation of Matrices for Linear Tetrahedral Elements

Tetrahedral elements are comprised of four nodes with linear shape functions to describe global coordinates and displacements in the element. Locations and displacements within the element are mapped from isoparametric coordinates to global coordinates using a weighted sum of shape functions, by:

$$^{0}x = \sum_{i=1}^{4} N_i x_i = (1 - r - s - t)x_1 + rx_2 + sx_3 + tx_4 \qquad (A-1)$$

$$u = \sum_{i=1}^{4} N_i u_i = (1 - r - s - t)u_1 + ru_2 + su_3 + tu_4 \qquad (A-2)$$

where *y* and *z* are defined similar to *x* plus *v* and *w* defined similar to *u*. Note that during the analysis, the nodal locations are only calculated once in the original configuration. Also, note that the subscripts refer to the node number within the element. Also, as seen through Equations (A-3) through (A-6), the shape functions for linear tetrahedral elements are:

$$N_1 = 1 - r - s - t \qquad (A-3)$$

$$N_2 = r \qquad (A-4)$$

$$N_3 = s \qquad (A-5)$$

$$N_4 = t \qquad (A-6)$$

To corroborate the relationship between global and isoparametric coordinates, the derivatives must be determined which leads to the generation of the Jacobian. The derivatives of global coordinates with respect to isoparametric coordinates are calculated by:

$$\frac{\partial\ {}^{0}x}{\partial r} = x_2 - x_1 \qquad (A-7)$$

$$\frac{\partial\ {}^{0}x}{\partial s} = x_3 - x_1 \qquad (A-8)$$

$$\frac{\partial\ {}^{0}x}{\partial t} = x_4 - x_1 \qquad (A-9)$$

The Jacobian matrix for the linear tetrahedral element can be built from Equations (A-7) through (A-9):

$$\frac{\boldsymbol{\partial}}{\boldsymbol{\partial r}} = \boldsymbol{J}\frac{\boldsymbol{\partial}}{\boldsymbol{\partial}\ {}^{0}\boldsymbol{x}} \Rightarrow \boldsymbol{J} = \begin{bmatrix} x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\ x_3 - x_1 & y_3 - y_1 & z_3 - z_1 \\ x_4 - x_1 & y_4 - y_1 & z_4 - z_1 \end{bmatrix} \qquad (A-10)$$

Using Equation (A-10), an expression for the derivative of *u* with respect to *x* in the TL framework may be determined:

$$\frac{\partial u}{\partial\ {}^{0}x} = \frac{\partial N_i}{\partial\ {}^{0}x}u_i = N_{i,1} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}x}u_1 + \frac{\partial r}{\partial\ {}^{0}x}u_2 + \frac{\partial s}{\partial\ {}^{0}x}u_3 + \frac{\partial t}{\partial\ {}^{0}x}u_4 \qquad (A-11)$$

$$\frac{\partial u}{\partial\ {}^{0}y} = \frac{\partial N_i}{\partial\ {}^{0}y}u_i = N_{i,2} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}y}u_1 + \frac{\partial r}{\partial\ {}^{0}y}u_2 + \frac{\partial s}{\partial\ {}^{0}y}u_3 + \frac{\partial t}{\partial\ {}^{0}y}u_4 \qquad (A-12)$$

$$\frac{\partial u}{\partial\ {}^{0}z} = \frac{\partial N_i}{\partial\ {}^{0}z}u_i = N_{i,3} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}z}u_1 + \frac{\partial r}{\partial\ {}^{0}z}u_2 + \frac{\partial s}{\partial\ {}^{0}z}u_3 + \frac{\partial t}{\partial\ {}^{0}z}u_4 \qquad (A-13)$$

where the derivatives for *v* and *w* are defined similarly and *i* is summed from 1 to 4.

Considering Equation (A-10), the inverse of the Jacobian is required to determine the partial derivatives of isoparametric coordinates with respect to global coordinates:

$$\boldsymbol{J^{-1}} = \begin{bmatrix} N_{2,1} = \dfrac{\partial r}{\partial\ {}^{0}x} & N_{3,1} = \dfrac{\partial s}{\partial\ {}^{0}x} & N_{4,1} = \dfrac{\partial t}{\partial\ {}^{0}x} \\ N_{2,2} = \dfrac{\partial r}{\partial\ {}^{0}y} & N_{3,2} = \dfrac{\partial s}{\partial\ {}^{0}y} & N_{4,2} = \dfrac{\partial t}{\partial\ {}^{0}y} \\ N_{2,3} = \dfrac{\partial r}{\partial\ {}^{0}z} & N_{3,3} = \dfrac{\partial s}{\partial\ {}^{0}z} & N_{4,3} = \dfrac{\partial t}{\partial\ {}^{0}z} \end{bmatrix}$$

$$= \frac{1}{\det(\boldsymbol{J})} \begin{bmatrix} (y_{31}z_{41} - y_{41}z_{31}) & (y_{41}z_{21} - y_{21}z_{41}) & (y_{21}z_{31} - y_{31}z_{21}) \\ (x_{41}z_{31} - x_{31}z_{41}) & (x_{21}z_{41} - x_{41}z_{21}) & (x_{31}z_{21} - x_{21}z_{31}) \\ (x_{31}y_{41} - x_{41}y_{31}) & (x_{41}y_{21} - x_{21}y_{41}) & (x_{21}y_{31} - x_{31}y_{21}) \end{bmatrix} \qquad (A-14)$$

Where $\det(\boldsymbol{J}) = x_{21}y_{31}z_{41} + x_{31}y_{41}z_{21} + x_{41}y_{21}z_{31} - x_{41}y_{31}z_{21} - x_{31}y_{21}z_{41} - x_{21}y_{41}z_{31}$ and $x_{ij} = x_i - x_j$.

As well, the final shape function derivatives in the TL framework may be found using the inverse Jacobian:

$$N_{1,1} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}x} = \frac{\partial 1}{\partial\ {}^{0}x} - \frac{\partial r}{\partial\ {}^{0}x} - \frac{\partial s}{\partial\ {}^{0}x} - \frac{\partial t}{\partial\ {}^{0}x} \qquad (A-15)$$

$$= \frac{1}{\det(\boldsymbol{J})}(y_{24}z_{32} - y_{23}z_{42})$$

$$N_{1,2} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}y} = \frac{\partial 1}{\partial\ {}^{0}y} - \frac{\partial r}{\partial\ {}^{0}y} - \frac{\partial s}{\partial\ {}^{0}y} - \frac{\partial t}{\partial\ {}^{0}y} \qquad (A-16)$$

$$= \frac{1}{\det(\boldsymbol{J})}(x_{32}z_{24} - x_{42}z_{23})$$

$$N_{1,3} = \frac{\partial(1-r-s-t)}{\partial\ {}^{0}z} = \frac{\partial 1}{\partial\ {}^{0}z} - \frac{\partial r}{\partial\ {}^{0}z} - \frac{\partial s}{\partial\ {}^{0}z} - \frac{\partial t}{\partial\ {}^{0}z} \qquad (A-17)$$

$$= \frac{1}{\det(\boldsymbol{J})}(x_{24}y_{32} - x_{23}y_{42})$$

Note that the inverse Jacobian does not depend on isoparametric parameters; hence it is constant across the tetrahedral element. Also, since the framework is TL, the original nodal coordinates are constant throughout the analysis; and therefore, the inverse Jacobian (ie shape function derivatives) can be pre-calculated once and stored during FE analysis (ie does not require recalculation at each Newton-Raphson iteration).

Given Equations (A-15) to (A-17) for the shape function derivatives and since ${}^{t}u_i$ is calculated for each node at each Newton-Raphson iteration, the element displacement gradient may be generated:

$${}^{t}_{0}u_{,j} = \sum_{k=1}^{4} N_{k,j}\ {}^{t}u_k \qquad (A-18)$$

$${}^{t}_{0}v_{,j} = \sum_{k=1}^{4} N_{k,j}\ {}^{t}v_k \qquad (A-19)$$

$${}^{t}_{0}w_{,j} = \sum_{k=1}^{4} N_{k,j}\ {}^{t}w_k \qquad (A-20)$$

Given Equations (1-15) and (1-16), the strain at the current configuration can be found including expressions for ${}_{0}e_{ij}$ and ${}_{0}\eta_{ij}$:

$$ {}_0e_{11} = \frac{1}{\det(\mathbf{J})}\left(N_{i,1}u_i + {}^t_0u_{,1}N_{i,1}u_i + {}^t_0v_{,1}N_{i,1}v_i + {}^t_0w_{,1}N_{i,1}w_i\right) \qquad (A-21) $$

$$ {}_0e_{22} = \frac{1}{\det(\mathbf{J})}\left(N_{i,2}v_i + {}^t_0u_{,2}N_{i,2}u_i + {}^t_0v_{,2}N_{i,2}v_i + {}^t_0w_{,2}N_{i,2}w_i\right) \qquad (A-22) $$

$$ {}_0e_{33} = \frac{1}{\det(\mathbf{J})}\left(N_{i,3}w_i + {}^t_0u_{,3}N_{i,3}u_i + {}^t_0v_{,3}N_{i,3}v_i + {}^t_0w_{,3}N_{i,3}w_i\right) \qquad (A-23) $$

$$ {}_0e_{23} = \frac{1}{2\det(\mathbf{J})}\begin{pmatrix} N_{i,2}w_i + N_{i,3}v_i + {}^t_0u_{,3}N_{i,2}u_i + {}^t_0v_{,3}N_{i,2}v_i + {}^t_0w_{,3}N_{i,2}w_i \\ +{}^t_0u_{,2}N_{i,3}u_i + {}^t_0v_{,2}N_{i,3}v_i + {}^t_0w_{,2}N_{i,3}w_i \end{pmatrix} \qquad (A-24) $$

$$ {}_0e_{13} = \frac{1}{2\det(\mathbf{J})}\begin{pmatrix} N_{i,1}w_i + N_{i,3}u_i + {}^t_0u_{,3}N_{i,1}u_i + {}^t_0v_{,3}N_{i,1}v_i + {}^t_0w_{,3}N_{i,1}w_i \\ +{}^t_0u_{,1}N_{i,3}u_i + {}^t_0v_{,1}N_{i,3}v_i + {}^t_0w_{,1}N_{i,3}w_i \end{pmatrix} \qquad (A-25) $$

$$ {}_0e_{12} = \frac{1}{2\det(\mathbf{J})}\begin{pmatrix} N_{i,2}u_i + N_{i,1}v_i + {}^t_0u_{,1}N_{i,2}u_i + {}^t_0v_{,1}N_{i,2}v_i + {}^t_0w_{,1}N_{i,2}w_i \\ +{}^t_0u_{,2}N_{i,1}u_i + {}^t_0v_{,2}N_{i,1}v_i + {}^t_0w_{,2}N_{i,1}w_i \end{pmatrix} \qquad (A-26) $$

where $\boldsymbol{u_i} = [u_i \quad v_i \quad w_i]^T$ is either $\boldsymbol{\Delta u}$ (incremental displacement) or $\boldsymbol{\delta u}$ (virtual displacement).

$$ \delta\Delta\ {}_0\eta_{11} = \frac{1}{\det(\boldsymbol{J})}\left(N_{i,1}\delta u_i N_{j,1}\Delta u_j + N_{i,1}\delta v_i N_{j,1}\Delta v_j + N_{i,1}\delta w_i N_{j,1}\Delta w_j\right) \qquad (A-27) $$

$$ \delta\Delta\ {}_0\eta_{22} = \frac{1}{\det(\boldsymbol{J})}\left(N_{i,2}\delta u_i N_{j,2}\Delta u_j + N_{i,2}\delta v_i N_{j,2}\Delta v_j + N_{i,2}\delta w_i N_{j,2}\Delta w_j\right) \qquad (A-28) $$

$$ \delta\Delta\ {}_0\eta_{33} = \frac{1}{\det(\boldsymbol{J})}\left(N_{i,3}\delta u_i N_{j,3}\Delta u_j + N_{i,3}\delta v_i N_{j,3}\Delta v_j + N_{i,3}\delta w_i N_{j,3}\Delta w_j\right) \qquad (A-29) $$

$$ \delta\Delta\ {}_0\eta_{23} = \frac{1}{2\det(\boldsymbol{J})}\begin{pmatrix} N_{i,2}\delta u_i N_{j,3}\Delta u_j + N_{i,2}\delta v_i N_{j,3}\Delta v_j + N_{i,2}\delta w_i N_{j,3}\Delta w_j \\ +N_{i,2}\Delta u_i N_{j,3}\delta u_j + N_{i,2}\Delta v_i N_{j,3}\delta v_j + N_{i,2}\Delta w_i N_{j,3}\delta w_j \end{pmatrix} \qquad (A-30) $$

$$\delta\Delta\,{}_0\eta_{13} = \frac{1}{2\det(\boldsymbol{J})}\begin{pmatrix} N_{i,1}\delta u_i N_{j,3}\Delta u_j + N_{i,1}\delta v_i N_{j,3}\Delta v_j + N_{i,1}\delta w_i N_{j,3}\Delta w_j \\ +N_{i,1}\Delta u_i N_{j,3}\delta u_j + N_{i,1}\Delta v_i N_{j,3}\delta v_j + N_{i,1}\Delta w_i N_{j,3}\delta w_j \end{pmatrix} \qquad (A-31)$$

$$\delta\Delta\,{}_0\eta_{12} = \frac{1}{2\det(\boldsymbol{J})}\begin{pmatrix} N_{i,1}\delta u_i N_{j,2}\Delta u_j + N_{i,1}\delta v_i N_{j,2}\Delta v_j + N_{i,1}\delta w_i N_{j,2}\Delta w_j \\ +N_{i,1}\Delta u_i N_{j,2}\delta u_j + N_{i,1}\Delta v_i N_{j,2}\delta v_j + N_{i,1}\Delta w_i N_{j,2}\delta w_j \end{pmatrix} \qquad (A-32)$$

From Equations (A-21) to (A-32), the strain displacement matrices can be generated by factoring out the incremental and virtual displacement terms, by:

$$\begin{bmatrix} e_{11} \\ e_{22} \\ e_{33} \\ e_{23} \\ e_{13} \\ e_{12} \end{bmatrix} = \hat{\boldsymbol{e}} = \boldsymbol{B}\boldsymbol{u} = (\boldsymbol{B_{L0}} + \boldsymbol{B_{L1}})\boldsymbol{u}$$

$$= \left(\begin{bmatrix} N_{1,1} & 0 & 0 & N_{2,1} & 0 & 0 & N_{3,1} & 0 & 0 & N_{4,1} & 0 & 0 \\ 0 & N_{2,1} & 0 & 0 & N_{2,2} & 0 & 0 & N_{3,2} & 0 & 0 & N_{4,2} & 0 \\ 0 & 0 & N_{1,3} & 0 & 0 & N_{2,3} & 0 & 0 & N_{3,3} & 0 & 0 & N_{4,3} \\ 0 & N_{1,3} & N_{1,2} & 0 & N_{2,3} & N_{2,2} & 0 & N_{3,3} & N_{3,2} & 0 & N_{4,3} & N_{4,2} \\ N_{1,3} & 0 & N_{1,1} & N_{2,3} & 0 & N_{2,1} & N_{3,3} & 0 & N_{3,1} & N_{4,3} & 0 & N_{4,1} \\ N_{1,2} & N_{1,1} & 0 & N_{2,2} & N_{2,1} & 0 & N_{3,2} & N_{3,1} & 0 & N_{4,2} & N_{4,1} & 0 \end{bmatrix} + [\boldsymbol{B_{L1_1}} \quad \boldsymbol{B_{L1_2}} \quad \boldsymbol{B_{L1_3}} \quad \boldsymbol{B_{L1_4}}] \right) \begin{bmatrix} u_1 \\ v_1 \\ w_1 \\ u_2 \\ v_2 \\ w_2 \\ u_3 \\ v_3 \\ w_3 \\ u_4 \\ v_4 \\ w_4 \end{bmatrix} (A-33)$$

$$\boldsymbol{B_{L1_i}} = \begin{bmatrix} {}_0^t u_{,1} N_{i,1} & {}_0^t v_{,1} N_{i,1} & {}_0^t w_{,1} N_{i,1} \\ {}_0^t u_{,2} N_{i,2} & {}_0^t v_{,2} N_{i,2} & {}_0^t w_{,2} N_{i,2} \\ {}_0^t u_{,3} N_{i,3} & {}_0^t v_{,3} N_{i,3} & {}_0^t w_{,3} N_{i,3} \\ ({}_0^t u_{,2} N_{i,3} + {}_0^t u_{,3} N_{i,2}) & ({}_0^t v_{,2} N_{i,3} + {}_0^t v_{,3} N_{i,2}) & ({}_0^t w_{,2} N_{i,3} + {}_0^t w_{,3} N_{i,2}) \\ ({}_0^t u_{,1} N_{i,3} + {}_0^t u_{,3} N_{i,1}) & ({}_0^t v_{,1} N_{i,3} + {}_0^t v_{,3} N_{i,1}) & ({}_0^t w_{,1} N_{i,3} + {}_0^t w_{,3} N_{i,1}) \\ ({}_0^t u_{,1} N_{i,2} + {}_0^t u_{,2} N_{i,1}) & ({}_0^t v_{,1} N_{i,2} + {}_0^t v_{,2} N_{i,1}) & ({}_0^t w_{,1} N_{i,2} + {}_0^t w_{,2} N_{i,1}) \end{bmatrix}$$

Note that with the strain definitions, the strain is constant throughout the element and doesn't rely on the isoparametric variables. Therefore, the presented formulation for tetrahedral elements corresponds to full integration (one Gauss integration point). Considering Equation (A-33), $\boldsymbol{B_{L0}}$ can be pre-computed at the original configuration and stored for all other iterations, while $\boldsymbol{B_{L1}}$ represents the initial displacement effect and must be computed at each iteration. When compared to Equation (1-21), Equation (A-33) represents strain in the first term on the left hand side and the second term on the right hand side:

$$\int_{{}_0V} {}_0C_{ijrs}^{k-1}\Delta\ {}_0e_{rs}^{k}\delta\ {}_0e_{ij}\, d\ {}^0V = \boldsymbol{\delta u}^T\ \boldsymbol{K_L \Delta u} = \boldsymbol{\delta u}^T\ \left(\boldsymbol{B_L^T C B_L}\ {}^0V\right)\boldsymbol{\Delta u} \qquad (A-34)$$

$$\int_{{}_0V} {}_0^{t+\Delta t}S_{ij}^{k-1}\delta\ {}_0^{t+\Delta t}\epsilon_{ij}^{k-1}\, d\ {}^0V = \boldsymbol{\delta u}^T\ \boldsymbol{F} = \boldsymbol{\delta u}^T\ \left(\boldsymbol{B_L^T \widehat{S}}\ {}^0V\right) \qquad (A-35)$$

where $\hat{\mathbf{S}} = [S_{11} \quad S_{22} \quad S_{33} \quad S_{23} \quad S_{13} \quad S_{12}]^T$.

The matrix for remaining unknown term representing nonlinear (i.e. geometric) stiffness in Equation (1-21) is generated by:

$$\int_{{}_0V} {}_0^{t+\Delta t}S_{ij}\,\delta\Delta\ {}_0\eta_{ij}\, d\ {}^0V$$

$$= \begin{pmatrix} {}_0^{t+\Delta t}S_{11}\delta\Delta\ {}_0\eta_{11} + {}_0^{t+\Delta t}S_{22}\delta\Delta\ {}_0\eta_{22} + {}_0^{t+\Delta t}S_{33}\delta\Delta\ {}_0\eta_{33} \\ +2\,{}_0^{t+\Delta t}S_{23}\delta\Delta\ {}_0\eta_{23} + 2\,{}_0^{t+\Delta t}S_{13}\delta\Delta\ {}_0\eta_{13} + 2\,{}_0^{t+\Delta t}S_{12}\delta\Delta\ {}_0\eta_{12} \end{pmatrix} {}^0V \qquad (A-36)$$

$$\Rightarrow \boldsymbol{\delta u}^T\ \boldsymbol{K_{NL}}\ \boldsymbol{\Delta u}$$

$$= \boldsymbol{\delta u}^T$$

$$\begin{bmatrix} M_{11} & 0 & 0 & M_{12} & 0 & 0 & M_{13} & 0 & 0 & M_{14} & 0 & 0 \\ 0 & M_{11} & 0 & 0 & M_{12} & 0 & 0 & M_{13} & 0 & 0 & M_{14} & 0 \\ 0 & 0 & M_{11} & 0 & 0 & M_{12} & 0 & 0 & M_{13} & 0 & 0 & M_{14} \\ M_{12} & 0 & 0 & M_{22} & 0 & 0 & M_{23} & 0 & 0 & M_{24} & 0 & 0 \\ 0 & M_{12} & 0 & 0 & M_{22} & 0 & 0 & M_{23} & 0 & 0 & M_{24} & 0 \\ 0 & 0 & M_{12} & 0 & 0 & M_{22} & 0 & 0 & M_{23} & 0 & 0 & M_{24} \\ M_{13} & 0 & 0 & M_{23} & 0 & 0 & M_{33} & 0 & 0 & M_{34} & 0 & 0 \\ 0 & M_{13} & 0 & 0 & M_{23} & 0 & 0 & M_{33} & 0 & 0 & M_{34} & 0 \\ 0 & 0 & M_{13} & 0 & 0 & M_{23} & 0 & 0 & M_{33} & 0 & 0 & M_{34} \\ M_{14} & 0 & 0 & M_{24} & 0 & 0 & M_{34} & 0 & 0 & M_{44} & 0 & 0 \\ 0 & M_{14} & 0 & 0 & M_{24} & 0 & 0 & M_{34} & 0 & 0 & M_{44} & 0 \\ 0 & 0 & M_{14} & 0 & 0 & M_{24} & 0 & 0 & M_{34} & 0 & 0 & M_{44} \end{bmatrix} \boldsymbol{\Delta u} \quad (A-37)$$

$$M_{ij} = M_{ji} = \left(S_{kl} N_{i,l} N_{j,k}\right) {}^0V$$

where *k,l* are summed in each direction from 1 to 3.

When compared to Equation (1-21), Equation (A-34) is added to Equation (A-37) to get the nonlinear stiffness matrix for the tetrahedral element.

Therefore, all matrices in Equation (1-22) are generated for the linear tetrahedral elements. The material property matrix and stress vector are derived in Appendix C for the various materials comprising the spine. Linear tetrahedral typically make up the solid components of the spine, such as the vertebrae, cortical bone, endplates, annulus fibrosus matrix, and nucleus pulposus.

Appendix B Derivation of Matrices for Linear Tension-Only Spring Elements

Linear spring elements are comprised of a one-dimensional line connecting two nodes at either end. Considering that the elements are one-dimensional, the derivation is different than for the linear tetrahedral elements. The ensuing formulation follows an FE methods course notes offered through the University of Colorado [140]. First, the axial GL strain must be derived:

$$ {}^{t+\Delta t}_{0}\epsilon = {}^{t+\Delta t}_{0}\epsilon_{11} = \frac{\partial^{t+\Delta t}u}{\partial\ {}^{0}x} + \frac{1}{2}\left(\left(\frac{\partial^{t+\Delta t}u}{\partial\ {}^{0}x}\right)^{2} + \left(\frac{\partial^{t+\Delta t}v}{\partial\ {}^{0}x}\right)^{2} + \left(\frac{\partial^{t+\Delta t}w}{\partial\ {}^{0}x}\right)^{2}\right) $$

$$ \Rightarrow {}^{t+\Delta t}_{0}\epsilon = \frac{d^{t+\Delta t}u}{d\ {}^{0}x} + \frac{1}{2}\left(\frac{d^{t+\Delta t}u}{d\ {}^{0}x}\right)^{2} = \frac{L - {}^{0}L}{{}^{0}L} + \frac{\left(L - {}^{0}L\right)^{2}}{2\ {}^{0}L^{2}} $$

$$ \Rightarrow {}^{t+\Delta t}_{0}\epsilon = \frac{2\ {}^{0}LL - 2\ {}^{0}L^{2} + L^{2} - 2\ {}^{0}LL + {}^{0}L^{2}}{2\ {}^{0}L^{2}} = \frac{L^{2} - {}^{0}L^{2}}{2\ {}^{0}L^{2}} \qquad (B-1) $$

where ε_{11} is the axial strain along the length of the element rather than the global strain in the x-direction.

Note that all variables in Equation (B-1) are in the element coordinate system and only axial variables contribute to the axial strain, thus *v* and *w* are zero. Also, constant strain is assumed throughout the element, thus $\frac{du}{dx} = \frac{L - {}^{0}L}{{}^{0}L}$. Note that with the constant strain definition, the strain calculation doesn't rely on the isoparametric variables. Therefore, like the linear tetrahedral elements, the presented formulation for spring elements corresponds to full integration (one Gauss integration point). Also, note in Equation (B-1) that the original length, ${}^{0}L$ is calculated

once at the beginning of the analysis and does not change throughout the analysis. The formula to calculate the original length and current length of the element is:

$$ {}^{0}L = \left({}^{0}x_2 - {}^{0}x_1 \right)^2 + \left({}^{0}y_2 - {}^{0}y_1 \right)^2 + \left({}^{0}z_2 - {}^{0}z_1 \right)^2 \qquad (B-2) $$

$$ L^2 = \left({}^{0}x_{21} + {}^{t+\Delta t}u_{21} \right)^2 + \left({}^{0}y_{21} + {}^{t+\Delta t}v_{21} \right)^2 + \left({}^{0}z_{21} + {}^{t+\Delta t}w_{21} \right)^2 \qquad (B-3) $$

where $x_{21} = x_2 - x_1$.

Looking at Equation (B-1), the current length, *L*, must be related to displacement to integrate the spring element into the general FE equation, Equation (B-3).

The directional cosines may be used to simplify the calculations:

$$ c_x = \frac{x_{21}}{{}^{0}L} \qquad (B-4) $$

$$ c_y = \frac{y_{21}}{{}^{0}L} \qquad (B-5) $$

$$ c_z = \frac{z_{21}}{{}^{0}L} \qquad (B-6) $$

Equation (B-3) is inserted into Equation (B-1) in terms of displacement:

$$ {}^{t+\Delta t}_{0}\epsilon \; 2 \; {}^{0}L^2 = \left({}^{0}x_{21}^2 + 2 \; {}^{0}x_{21} \, {}^{t+\Delta t}u_{21} + {}^{t+\Delta t}u_{21}^2 + {}^{0}y_{21}^2 + 2 \; {}^{0}y_{21} \, {}^{t+\Delta t}v_{21} + {}^{t+\Delta t}v_{21}^2 + {}^{0}z_{21}^2 \right. $$

$$ \left. + 2 \; {}^{0}z_{21} \, {}^{t+\Delta t}w_{21} + {}^{t+\Delta t}w_{21}^2 \right) - \left({}^{0}x_{21}^2 + {}^{0}y_{21}^2 + {}^{0}z_{21}^2 \right) $$

$$ \Rightarrow {}^{t+\Delta t}_{0}\epsilon = $$

$$ \frac{2 \; {}^{0}x_{21} \, {}^{t+\Delta t}u_{21} + 2 \; {}^{0}y_{21} \, {}^{t+\Delta t}v_{21} + 2 \; {}^{0}z_{21} \, {}^{t+\Delta t}u_{21} + {}^{t+\Delta t}u_{21}^2 + {}^{t+\Delta t}v_{21}^2 + {}^{t+\Delta t}w_{21}^2}{2 \; {}^{0}L^2} \qquad (B-4) $$

From Equations (1-5) and (1-6), the GL strain can be broken down into its incremental components:

$$ {}^{t+\Delta t}_{0}\epsilon = {}^{t}_{0}\epsilon + {}_{0}\epsilon $$

$$ {}^{t+\Delta t}_{0}\epsilon\ 2L_0^2 = 2\ {}^{0}x_{21}\left({}^{t}u_{21} + u_{21}\right) + 2\ {}^{0}y_{21}\left({}^{t}v_{21} + v_{21}\right) + 2\ {}^{0}z_{21}\left({}^{t}w_{21} + w_{21}\right) $$

$$ + \left({}^{t}u_{21} + u_{21}\right)^2 + \left({}^{t}v_{21} + v_{21}\right)^2 + \left({}^{t}w_{21} + w_{21}\right)^2 $$

$$ = 2\ {}^{0}x_{21}\ {}^{t}u_{21} + 2\ {}^{0}y_{21}\ {}^{t}v_{21} + 2\ {}^{0}z_{21}\ {}^{t}w_{21} + {}^{t}u_{21}^2 + {}^{t}v_{21}^2 + {}^{t}w_{21}^2 + 2\ {}^{t}x_{21}u_{21} $$

$$ + 2\ {}^{t}y_{21}v_{21} + 2\ {}^{t}z_{21}w_{21} + u_{21}^2 + v_{21}^2 + w_{21}^2 $$

$$ \Rightarrow {}^{t}_{0}\epsilon = \frac{2\ {}^{0}x_{21}\ {}^{t}u_{21} + 2\ {}^{0}y_{21}\ {}^{t}v_{21} + 2\ {}^{0}z_{21}\ {}^{t}w_{21} + {}^{t}u_{21}^2 + {}^{t}v_{21}^2 + {}^{t}w_{21}^2}{2\ {}^{0}L^2} \qquad (B-5) $$

$$ \Rightarrow {}_{0}\epsilon = \frac{2\ {}^{t}x_{21}u_{21} + 2\ {}^{t}y_{21}v_{21} + 2\ {}^{t}z_{21}w_{21} + u_{21}^2 + v_{21}^2 + w_{21}^2}{2\ {}^{0}L^2} \qquad (B-6) $$

Where ${}^{t}x = {}^{0}x + {}^{t}u$.

Given Equation (1-2), the linear and nonlinear components of the GL strain increment can be derived and simplified using the cosine Equations (B-4) to (B-6):

$$ {}_{0}\epsilon = {}_{0}e + {}_{0}\eta $$

$$ \Rightarrow {}_{0}e = \frac{{}^{t}x_{21}u_{21} + {}^{t}y_{21}v_{21} + {}^{t}z_{21}w_{21}}{{}^{0}L^2} = \frac{{}^{t}c_x u_{21} + {}^{t}c_y v_{21} + {}^{t}c_z w_{21}}{{}^{0}L^2} \qquad (B-7) $$

$$ \Rightarrow {}_{0}\eta = \frac{u_{21}^2 + v_{21}^2 + w_{21}^2}{2\ {}^{0}L^2} \qquad (B-8) $$

To determine the strain displacement matrices for input into Equation (1-22), the variation of the nonlinear incremental GL strain component must be generated:

$$\delta\Delta\ {}_0\eta = \frac{\Delta u_{21}\delta u_{21} + \Delta v_{21}\delta v_{21} + \Delta w_{21}\delta w_{21}}{{}^0L^2} \qquad (B-9)$$

where $\Delta u_{21} = \Delta u_2 - \Delta u_1$ and $\delta u_{21} = \delta u_2 - \delta u_1$.

Equation (B-7) can readily be converted into matrix form:

$${}_0e = \boldsymbol{Bu} = \frac{1}{{}^0L}[-c_x \quad -c_y \quad -c_z \quad c_x \quad c_y \quad c_z]\,[u_1 \quad v_1 \quad w_1 \quad u_2 \quad v_2 \quad w_2]^T \quad (B-10)$$

From Equation (1-22) and considering that $C_{1111} = E$ in a one-dimensional element, the expression for the linear stiffness matrix is generated:

$$\int_{{}_0V} {}_0C_{1111}^{k-1}\Delta\ {}_0e^k\,\delta\ {}_0e\ \ d\ {}^0V = \boldsymbol{\delta u}\,\boldsymbol{K_L}\boldsymbol{\Delta u} = \boldsymbol{\delta u}\left(\ {}^tE\boldsymbol{B_L^T}\boldsymbol{B_L}\ {}^0V\right)\boldsymbol{\Delta u}$$

$$= \boldsymbol{\delta u}^T\frac{{}^tE\ {}^0A}{{}^0L}\begin{bmatrix} c_x^2 & c_xc_y & c_xc_z & -c_x^2 & -c_xc_y & -c_xc_z \\ c_xc_y & c_y^2 & c_yc_z & -c_xc_y & -c_y^2 & -c_yc_z \\ c_xc_z & c_yc_z & c_z^2 & -c_xc_z & -c_yc_z & -c_z^2 \\ -c_x^2 & -c_xc_y & -c_xc_z & c_x^2 & c_xc_y & c_xc_z \\ -c_xc_y & -c_y^2 & -c_yc_z & c_xc_y & c_y^2 & c_yc_z \\ -c_xc_z & -c_yc_z & -c_z^2 & c_xc_z & c_yc_z & c_z^2 \end{bmatrix}\boldsymbol{\Delta u} \qquad (B-11)$$

Likewise, the expression for the nonlinear (i.e. geometric) stiffness matrix is generated considering that the PK2 stress is ${}^{t+\Delta t}_0S = \frac{{}^{t+\Delta t}F}{A_0}$:

$$\int_{{}_0V} {}^{t+\Delta t}_0S\ \ \delta\Delta\ {}_0\eta\ \ d\ {}^0V = \frac{{}^{t+\Delta t}F}{{}^0A}\ {}^0V\delta\Delta\ {}_0\eta$$

$$= \boldsymbol{\delta u}^T \frac{^{t+\Delta t}F}{^0L} \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{\Delta u} \qquad (B-12)$$

Lastly, the expression for the out-of-balance force vector is generated:

$$\int_{^0V} {}^{t+\Delta t}_{0}S^{k-1} \delta {}^{t+\Delta t}_{0}\epsilon^{k-1} d\ {}^0V = \boldsymbol{\delta u}^T \left(\frac{^{t+\Delta t}F}{^0A} \boldsymbol{B_L^T}\ {}^0V \right)$$

$$= \boldsymbol{\delta u}^T {}^{t+\Delta t}F \begin{bmatrix} -c_x & -c_y & -c_z & c_x & c_y & c_z \end{bmatrix}^T \qquad (B-13)$$

Therefore, all matrices in Equation (1-22) are generated for the linear spring elements. The stiffness and ligament force at each iteration is determined directly from the force-displacement curve of the ligament, where the elements only add force and stiffness to the overall model when they are in tension. The tension-only spring elements are used to generate the ligaments only.

Appendix C Derivation of Material Property Matrices

Various material models were used to represent the various biomechanical materials present in the spine. The material models used were: linear elastic isotropic for the cartilage endplates, posterior elements, and nucleus pulposus; linear elastic orthotropic for the cancellous bone and cortical bone; and Mooney-Rivlin hyperelastic for the annulus fibrosus. Note that all derivations are calculated within the TL formulation for the current configuration (t) with reference to the initial configuration (0) (so the left super- and sub-scripts have been removed for this section). To implement the material models into the FE program considering Equation (1-22), mathematical relations between PK2 stress and GL strain must be found. In the nonlinear elastic material region, PK2 is related to GL strain via the strain energy density of the material:

$$\boldsymbol{S} = \frac{\partial W}{\partial \boldsymbol{\epsilon}} \qquad (C-1)$$

At each increment of Equation (1-21), the tangential material property matrix, C_{ijrs}, must be calculated as the linear portion of the nonlinear material curve at the current GL strain:

$$S_{ij} = C_{ijrs}\epsilon_{rs} \qquad (C-2)$$

The linear elastic isotropic models are based off the Saint Venant-Kirchhoff model. The strain energy density function and resulting relation derived using Equation (C-1) are:

$$W = \frac{\lambda}{2}(\epsilon_{ii})^2 + \mu\epsilon_{ii}^2 \qquad (C-3)$$

$$S_{ij} = \lambda\epsilon_{kk}\delta_{ij} + 2\mu\epsilon_{ij} \qquad (C-4)$$

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}$$

$$\mu = \frac{E}{2(1+\nu)}$$

where δ_{ij} is the Kronecker delta and λ and μ are Lame constants.

Note that the Saint Venant-Kirchhoff model is an extension of Hooke's law into the nonlinear deformation regime.

The material property tensor can be derived by comparing Equations (C-4) and (C-2):

$$C_{ijrs} = \lambda\delta_{ij}\delta_{rs} + \mu\left(\delta_{ir}\delta_{js} + \delta_{is}\delta_{jr}\right) \qquad (C-5)$$

Then, the material property matrix can be derived:

$$\boldsymbol{C} = \lambda\begin{bmatrix} (1-\nu) & \nu & \nu & 0 & 0 & 0 \\ \nu & (1-\nu) & \nu & 0 & 0 & 0 \\ \nu & \nu & (1-\nu) & 0 & 0 & 0 \\ 0 & 0 & 0 & 2\mu & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\mu & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\mu \end{bmatrix} \qquad (C-6)$$

Considering $\hat{\boldsymbol{\epsilon}} = [\epsilon_{11} \quad \epsilon_{22} \quad \epsilon_{33} \quad \epsilon_{23} \quad \epsilon_{13} \quad \epsilon_{12}]^T$ and $\widehat{\boldsymbol{S}} = [S_{11} \quad S_{22} \quad S_{33} \quad S_{23} \quad S_{13} \quad S_{12}]^T$ are the vectors forms of the PK2 stress and GL strains, respectively. Clearly, the material property matrix for Saint Venant-Kirchhoff materials are constant during the entire analysis.

For linear elastic orthotropic materials, a similar relation may be derived from the Saint Venant-Kirchhoff model for orthotropic materials. The compliance matrix in each direction is:

$$\begin{bmatrix}\epsilon_{11}\\ \epsilon_{22}\\ \epsilon_{33}\\ \epsilon_{23}\\ \epsilon_{13}\\ \epsilon_{12}\end{bmatrix} = \begin{bmatrix} \frac{1}{E_{11}} & -\frac{\nu_{21}}{E_{22}} & -\frac{\nu_{31}}{E_{33}} & 0 & 0 & 0 \\ -\frac{\nu_{12}}{E_{11}} & \frac{1}{E_{22}} & -\frac{\nu_{32}}{E_{33}} & 0 & 0 & 0 \\ -\frac{\nu_{13}}{E_{11}} & -\frac{\nu_{23}}{E_{22}} & \frac{1}{E_{33}} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2G_{23}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2G_{31}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2G_{12}} \end{bmatrix} \begin{bmatrix}S_{11}\\ S_{22}\\ S_{33}\\ S_{23}\\ S_{13}\\ S_{12}\end{bmatrix} \qquad (C-7)$$

where $\frac{\nu_{21}}{E_{22}} = \frac{\nu_{12}}{E_{11}}$, $\frac{\nu_{31}}{E_{33}} = \frac{\nu_{13}}{E_{11}}$, and $\frac{\nu_{32}}{E_{33}} = \frac{\nu_{23}}{E_{22}}$ to satisfy symmetry and positive definiteness, which ensure positive strain energy.

Note that orthotropic materials have three orthogonal symmetry planes and that the strains in each direction are found more readily than the stresses in each direction.

To obtain the stiffness matrix (i.e. material property matrix), the inverse of the compliance matrix must be calculated:

$$\boldsymbol{C} \qquad (C-8)$$

$$= \begin{bmatrix} \frac{E_{11}(E_{22}-\nu_{23}^2E_{33})}{E_{22}\Delta} & \frac{E_{22}\nu_{12}+\nu_{13}\nu_{23}E_{33}}{\Delta} & \frac{E_{33}(\nu_{13}+\nu_{23}\nu_{12})}{\Delta} & 0 & 0 & 0 \\ \frac{E_{22}\nu_{12}+\nu_{13}\nu_{23}E_{33}}{\Delta} & \frac{E_{22}(E_{11}-\nu_{23}^2E_{33})}{E_{11}\Delta} & \frac{E_{33}(E_{11}\nu_{23}+\nu_{13}\nu_{12}E_{22})}{E_{11}\Delta} & 0 & 0 & 0 \\ \frac{E_{33}(\nu_{13}+\nu_{23}\nu_{12})}{\Delta} & \frac{E_{33}(E_{11}\nu_{23}+\nu_{13}\nu_{12}E_{22})}{E_{11}\Delta} & \frac{E_{33}(E_{11}-\nu_{12}^2E_{22})}{E_{11}\Delta} & 0 & 0 & 0 \\ 0 & 0 & 0 & 2G_{23} & 0 & 0 \\ 0 & 0 & 0 & 0 & 2G_{31} & 0 \\ 0 & 0 & 0 & 0 & 0 & 2G_{12} \end{bmatrix}$$

where here $\Delta = 1 - \frac{\nu_{12}^2E_{22}}{E_{11}} - \frac{\nu_{23}^2E_{33}}{E_{22}} - \frac{\nu_{13}^2E_{33}}{E_{11}} - \frac{2\nu_{12}\nu_{23}\nu_{13}E_{33}}{E_{11}}$.

Note that the material property matrix is again constant throughout the FE analysis.

For the following derivation regarding Mooney-Rivlin material, the right Cauchy-Green tensor, denoted by D_{ij}, is more useful than the GL strain tensor. The relation between the right Cauchy-Green tensor and the GL strain tensor is:

$$D_{ij} = 2\epsilon_{ij} + \delta_{ij} \qquad (C-9)$$

The ensuing derivation follows work done by Sussman & Bathe [141]. For Mooney-Rivlin hyperelastic material including compressibility, the strain energy density function is:

$$W = C_{01}(\overline{I_2} - 3) + C_{10}(\overline{I_1} - 3) + \frac{\kappa}{2}(J-1)^2 \qquad (C-10)$$

where $J = I_3^{\frac{1}{2}} = \det(\mathbf{D})^{\frac{1}{2}}$ is the determinant of the deformation gradient tensor (I_3 is the third invariant of the right Cauchy-Green tensor), $\overline{I_2} = J^{-\frac{2}{3}}I_2$ is the reduced second invariant (I_2 is the second invariant) of the right Cauchy-Green tensor, $\overline{I_1} = J^{-\frac{4}{3}}I_1$ is the reduced first invariant

(I_1 is the first invariant) of the right Cauchy-Green tensor, C_{01} and C_{10} are Mooney-Rivlin constants, and κ is the bulk modulus. The formulae for the Cauchy-Green tensor invariants are:

$$I_1 = D_{11} + D_{22} + D_{33} \qquad (C-11)$$

$$I_2 = D_{11}D_{22} + D_{11}D_{33} + D_{22}D_{33} - D_{12}^2 - D_{13}^2 - D_{23}^2 \qquad (C-12)$$

The relationship between the right Cauchy-Green tensor and the PK2 stress is updated by:

$$S_{ij} = \frac{\partial W}{\partial D_{ij}} + \frac{\partial W}{\partial D_{ji}} \qquad (C-13)$$

Inserting Equation (C-10) into Equation (C-13), the PK2 stress is calculated:

$$S_{ij} = C_{10}(\bar{I}_1)^*_{ij} + C_{01}(\bar{I}_2)^*_{ij} + \kappa(J-1)(J)^*_{ij} \qquad (C-14)$$

where $(\,)^*_{ij} = \frac{\partial(\,)}{\partial D_{ij}} + \frac{\partial(\,)}{\partial D_{ji}}$. Hence, the unknown relations in Equation (C-14) are:

$$(\bar{I}_1)^*_{ij} = I_3^{-\frac{1}{3}}(I_1)^*_{ij} - \frac{1}{3}I_1 I_3^{-\frac{4}{3}}(I_3)^*_{ij} \qquad (C-15)$$

$$(\bar{I}_2)^*_{ij} = I_3^{-\frac{2}{3}}(I_2)^*_{ij} - \frac{2}{3}I_2 I_3^{-\frac{5}{3}}(I_3)^*_{ij} \qquad (C-16)$$

$$(J)^*_{ij} = \frac{1}{2}I_3^{-\frac{1}{2}}(I_3)^*_{ij} \qquad (C-17)$$

where the invariant derivatives are given by Equations (C-18) to (C-20).

$$(I_1)^*_{ij} = 2\delta_{ij} \qquad (C-18)$$

$$(I_2)^*_{ij} = 2I_1\delta_{ij} - 2D_{ij} \qquad (C-19)$$

$$(I_3)^*_{ij} = \frac{1}{2}\left(\hat{\varepsilon}_{ibc}\hat{\varepsilon}_{jdf} + \hat{\varepsilon}_{jbc}\hat{\varepsilon}_{idf}\right)D_{bd}D_{cf} \qquad (C-20)$$

where $\hat{\varepsilon}_{ijk}$ is the permutation tensor.

By substituting Equations (C-15) through (C-20) into Equation (C-14), the PK2 stress at each iteration for Mooney-Rivlin materials is calculated. Note that PK2 stress is directly calculated in this manner rather than multiplying by the tangential material property matrix.

For the tangential material property matrix of Mooney-Rivlin materials at each iteration, Equation (C-2) implies that the second derivative of the strain energy density must be calculated:

$$C_{ijrs} = \frac{\partial S_{ij}}{\partial D_{rs}} + \frac{\partial S_{ij}}{\partial D_{sr}} \qquad (C-21)$$

Inserting Equation (C-14) into Equation (C-21), the tangential material property matrix is derived:

$$C_{ijrs} = C_{10}(\bar{I}_1)^{**}_{ijrs} + C_{01}(\bar{I}_2)^{**}_{ijrs} + \kappa\left((J)^*_{ij}(J)^*_{rs} + (J-1)(J)^{**}_{ijrs}\right) \qquad (C-22)$$

where $(\,)^{**}_{ijrs} = \frac{\partial(\,)^*_{ij}}{\partial D_{rs}} + \frac{\partial(\,)^*_{ij}}{\partial D_{sr}}$.

Hence, the unknown relations in Equation (C-22) are:

$$(\bar{I}_1)^{**}_{ijrs} = I_3^{-\frac{1}{3}}(I_1)^{**}_{ijrs} - \frac{1}{3}I_3^{-\frac{4}{3}}\left(2\delta_{ij}(I_3)^*_{rs} + 2\delta_{rs}(I_3)^*_{ij} + I_1(I_3)^{**}_{ijrs}\right)$$

$$+\frac{4}{9}I_1 I_3^{-\frac{7}{3}}(I_3)_{ij}^{*}(I_3)_{rs}^{*} \qquad (C-23)$$

$$(\bar{I}_2)_{ijrs}^{**} = I_3^{-\frac{2}{3}}(I_2)_{ijrs}^{**} - \frac{2}{3}I_3^{-\frac{5}{3}}\Big(\big(2\delta_{ij}I_1 - 2D_{ij}\big)(I_3)_{rs}^{*} + (2\delta_{rs}I_1 - 2C_{rs})(I_3)_{ij}^{*} + I_2(I_3)_{ijrs}^{**}\Big)$$

$$+\frac{10}{9}I_2 I_3^{-\frac{8}{3}}(I_3)_{ij}^{*}(I_3)_{rs}^{*} \qquad (C-24)$$

$$(J)_{ijrs}^{**} = -\frac{1}{4}I_3^{-\frac{3}{2}}(I_3)_{ij}^{*}(I_3)_{rs}^{*} + \frac{1}{2}I_3^{-\frac{1}{2}}(I_3)_{ijrs}^{**} \qquad (C-25)$$

where the unknown invariant derivatives are given by:

$$(I_1)_{ijrs}^{**} = 0 \qquad (C-26)$$

$$(I_2)_{ijrs}^{**} = 4\delta_{ij}\delta_{rs} - 2\big(\delta_{ir}\delta_{js} + \delta_{is}\delta_{jr}\big) \qquad (C-27)$$

$$(I_3)_{ijrs}^{**} = \big(\hat{\varepsilon}_{irc}\hat{\varepsilon}_{jsf} + \hat{\varepsilon}_{isc}\hat{\varepsilon}_{jrf} + \hat{\varepsilon}_{jrc}\hat{\varepsilon}_{isf} + \hat{\varepsilon}_{jsc}\hat{\varepsilon}_{irf}\big)D_{cf} \qquad (C-28)$$

By substituting Equations (C-23) to (C-28) into Equation (C-22), the tangential material property matrix for Mooney-Rivlin material is calculated at each iteration, as required by Equation (1-21). When generating the matrix from Equation (C-22), the symmetry of the matrix must be noted, meaning that for C_{ijrs}: $ij = ji$; $rs = sr$; and $ij = rs$. As such, the resulting matrix has only 21 entries:

$$\boldsymbol{C} = \begin{bmatrix} C_{1111} & C_{1122} & C_{1133} & C_{1123} & C_{1113} & C_{1112} \\ C_{1122} & C_{2222} & C_{2233} & C_{2223} & C_{2213} & C_{2212} \\ C_{1133} & C_{2233} & C_{3333} & C_{3323} & C_{3313} & C_{3312} \\ C_{1123} & C_{2223} & C_{3323} & C_{2323} & C_{2313} & C_{2312} \\ C_{1113} & C_{2213} & C_{3313} & C_{2313} & C_{1313} & C_{1312} \\ C_{1112} & C_{2212} & C_{3312} & C_{2312} & C_{1312} & C_{1212} \end{bmatrix} \qquad (C-29)$$

Appendix D Transformation Matrix

For the rule of mixtures in a composite material, such as the annulus fibrosus, the fibers must be rotated into the coordinate system of the bulk material. Therefore, the fibers must be multiplied by a transformation matrix before adding their stiffness and stress contributions to the bulk material to form the material property matrix representing the composite material.

When given two coordinate systems (one for the bulk material and one for the fibers) defined by basis vectors, a rotation matrix may be specified to map vectors from one coordinate system (i.e. the fiber) to the other (i.e. the bulk material):

$$\vec{e}_b = \boldsymbol{R}\,\vec{e}_f \qquad (D-1)$$

where $\vec{e}_b$ is the set of basis vectors for the bulk material coordinate system, $\vec{e}_f$ is for the fibre coordinate system, and $\boldsymbol{R}$ is the rotation matrix.

The fiber direction is at given angles, θ and ϕ, within the bulk material. Hence, $\boldsymbol{R}$ may be determined by:

$$\boldsymbol{R} = \begin{bmatrix} l_1 = \cos(\theta)\cos(\phi) & l_2 = \sin(\theta)\sin(\phi) & l_3 = \sin(\theta) \\ m_1 & m_2 & m_3 \\ n_1 & n_2 & n_3 \end{bmatrix} \qquad (D-2)$$

Note that the fiber exhibits uniaxial stiffness; and therefore, it's material property matrix is given by:

$$\boldsymbol{C_F} = \begin{bmatrix} E & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad (D-3)$$

The rotation matrix presented in Equation (D-2) must then converted into a second-rank transformation matrix, $\boldsymbol{T}$, to transform the second-order tensor of stress:

$$T_{ijkl} = R_{ik}R_{jl}$$

$$\boldsymbol{T} = \begin{bmatrix} l_1^1 & l_2^2 & l_3^2 & l_2l_3 & l_1l_3 & l_1l_1 \\ m_1^2 & m_2^2 & m_3^2 & m_2m_3 & m_1m_3 & m_1m_2 \\ n_1^2 & n_2^2 & n_3^2 & n_2n_3 & n_1n_3 & n_1n_2 \\ 2m_1n_1 & 2m_2n_2 & 2m_3n_3 & n_2m_3 + n_3m_2 & m_1n_3 + m_3n_1 & n_1m_2 + n_2m_1 \\ 2l_1n_1 & 2\,l_2n_2 & 2l_3n_3 & l_2n_3 + l_3n_2 & l_1n_3 + l_3n_1 & l_1n_2 + l_2n_1 \\ 2l_1m_1 & 2l_2m_2 & 2l_3m_3 & l_2m_3 + l_3m_2 & l_1m_3 + l_3m_1 & l_1m_2 + l_2m_1 \end{bmatrix} \quad (D-4)$$

Consider that stress is transformed in its vector form, in which the vector form (considering symmetry of the stress tensor) is given by Equation (A-35).

Therefore, transformation matrix, $\boldsymbol{T}$, as a second-rank tensor may be used to directly transform stress from the fiber coordinate system into the bulk material coordinate system. To transform the material property matrix of the fiber into the bulk material coordinate system, the second-rank transformation matrix must be converted into a fourth-rank transformation matrix, as done in Equation (4-3). Therefore, the transformation matrix in Equations (4-3) and (4-4) becomes:

$$\boldsymbol{T}^T\boldsymbol{C_F} = E\begin{bmatrix} l_1^2 & 0 & 0 & 0 & 0 & 0 \\ l_2^2 & 0 & 0 & 0 & 0 & 0 \\ l_3^2 & 0 & 0 & 0 & 0 & 0 \\ l_2l_3 & 0 & 0 & 0 & 0 & 0 \\ l_1l_3 & 0 & 0 & 0 & 0 & 0 \\ l_1l_2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\Rightarrow \boldsymbol{T} = \begin{bmatrix} l_1^2 & l_2^2 & l_3^2 & l_2l_3 & l_1l_3 & l_1l_2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (D-5)$$

Appendix E CUDA Program Code

The CUDA real-time program, RealTimeSim.cu, accepted five text files as inputs that represented the spine model mesh in addition to vector and matrix assembly indexing arrays in preparation for the model solving. These five text files were: NODE_LOC.txt; ELEMENT_IND_EDIT.txt,. To obtain these text files, certain files were output from ANSYS for processing in a CUDA preparation program (written in C++): a node location text file; an element indexing file that included element and material type identification numbers for each element. The CUDA preparation program, CUDAprep.cpp, reorganized the ANSYS files for usage within the CUDA real-time program, where ANSYS were input to the CUDA preparation program and the above mentioned text files were output (for input to the CUDA real-time program). Please see below the code for the CUDAprep.cpp program and the RealTimeSim.cu program.

CUDAprep.cpp

```
// CreateSharedMemIndex.cpp : Defines the entry point for the console application.
//

#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>
#include <set>
#include <unordered_set>
#include <stdio.h>

#define BLOCK_SIZE 32

using namespace std;
using std::cin;
using std::cout;
using std::endl;

int main(int argc)
{

	float a1;
```

```
	int count = 0;
	float a2, a3, a4;
	const int global_dof = 3;
	vector<float> node_loc;

	string line;
	ifstream infile("NODE_LOC_BEFORE.txt");
	ifstream infile2("ELEMENT_IND_EDIT.txt");
	ifstream infile4("F_NODES.txt");
	ifstream infile7("BC_NODES.txt");
	ifstream infile5("AF_NODES.txt");
	if (infile.is_open()) {
		while (getline(infile, line))
		{

			infile >> a1 >> a2 >> a3 >> a4;
			node_loc.push_back(a2);
			node_loc.push_back(a3);
			node_loc.push_back(a4);
			//printf("Count %i Node %i: Location %f %f %f\n", count + 1, a1, node_loc[count * 3],
node_loc[count * 3 + 1], node_loc[count * 3 + 2]);
			count++;

		}
		infile.close();
	}
	else cout << "Unable to open node file" << endl;

	int node_loc_size = (int)node_loc.size();
	int node_count = (int)node_loc.size() / global_dof;
	float* node_loc_array = new float[node_loc_size];
	for (int i = 0; i < node_loc_size; i++) {
		node_loc_array[i] = node_loc[i];
	}

	float b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14;
	long count2 = 0;
	vector<float> element_ind_vector;
	vector<float> element_mat_vec;
	vector<float> element_type_vec;
	vector<float> element_real_vec;
	vector<float> element_sec_vec;
	vector<float> element_num_vec;
	const int nodes_per_tet = 4;
	const int nodes_per_shell = 3;
	const int nodes_per_lig = 2;
	const int nodes_per_fib = 2;
	const int tet_element_dof = 12;
	const int lig_element_dof = 6;
	int max = 0;
	int temp = 0;
	int ind = 0;
	int total_dof = 0;

	//Initiate outfiles before the goto Error
	ofstream outfile2("SHARED_GLOBAL_ASSEMBLY_IND.txt");
	ofstream outfile_err("ERROR_CHECK.txt");
	ofstream outfile_csr("OUTFILE_CSR.txt");
	ofstream outfile8("AF_IND_PRESSURE.txt");  // Adjust this when adding spine model back in
```

```
                                        //Initiate vectors before the goto Error
        vector<int> shell_nodes;
        vector<int> P_nodes_vec;

        string line2;

        if (infile2.is_open())
        {
                while (getline(infile2, line2))
                {

                        infile2 >> b1 >> b2 >> b3 >> b4 >> b5 >> b6 >> b7 >> b8 >> b9 >> b10 >> b11 >> b12 >>
b13 >> b14;
                        element_ind_vector.push_back(b1 - 1);
                        element_ind_vector.push_back(b2 - 1);
                        element_ind_vector.push_back(b3 - 1);
                        element_ind_vector.push_back(b5 - 1);
                        element_mat_vec.push_back(b9);
                        element_type_vec.push_back(b10);
                        element_real_vec.push_back(b11);
                        element_sec_vec.push_back(b12);
                        element_num_vec.push_back(b14);
                        count2++;

                }
                infile2.close();
        }
        else cout << "Unable to open element file" << endl;

        // Collect all force nodes
        //****************Ensure to add the "rotation node" at the end of the F_NODES array before running this code
        vector<int> f_nodes_vec;
        count = 0;

        string line4;
        if (infile4.is_open()) {
                while (getline(infile4, line4))
                {

                        infile4 >> a1 >> a2 >> a3 >> a4;
                        f_nodes_vec.push_back((int)a1 - 1);
                        //printf("Count %i Node %i: Location %f %f %f\n", count + 1, a1, node_loc[count * 3],
node_loc[count * 3 + 1], node_loc[count * 3 + 2]);
                        count++;

                }
                infile4.close();
        }
        else cout << "Unable to open f_node file" << endl;

        int f_nodes_size = (int)f_nodes_vec.size(); // f_nodes_size-2 gives the element count.  was +1
        int f_node_count = f_nodes_size;
        int* f_nodes = new int[f_nodes_size];
        for (int i = 0; i < f_nodes_size; i++) {
                f_nodes[i] = f_nodes_vec[i];
        }

        vector<int> bc_nodes_vec;
```

```
	int BC_count = 0;
	if (infile7.is_open()) {
		while (getline(infile7, line4))
		{

			infile7 >> a1 >> a2 >> a3 >> a4;
			bc_nodes_vec.push_back((int)a1 - 1);
			//printf("Count %i Node %i: Location %f %f %f\n", count + 1, a1, node_loc[count * 3],
node_loc[count * 3 + 1], node_loc[count * 3 + 2]);
			BC_count++;

		}
		infile7.close();
	}
	else cout << "Unable to open f_node file" << endl;

	////Collect AF_NODES and INDEXING
	vector<int> af_nodes_vec;
	float af2;
	int af_count = 0;
	if (infile5.is_open()) {
		while (getline(infile5, line4))
		{
			infile5 >> b1 >> af2;
			af_nodes_vec.push_back((int)b1 - 1);
			af_nodes_vec.push_back((int)af2 - 1);
			af_count++;
		}
		infile5.close();
	}
	else cout << "Unable to open af_nodes file" << endl;

	int buff_size = (int)element_mat_vec.size();
	int file_element_count = buff_size;
	int* element_ind = new int[(int)element_ind_vector.size()];
	int* element_mat = new int[buff_size];
	int* element_type = new int[(int)element_type_vec.size()];
	int* element_real = new int[(int)element_real_vec.size()];
	int* element_sec = new int[(int)element_sec_vec.size()];
	int* element_num = new int[(int)element_num_vec.size()];
	int solid_element_count = 0;
	int met_element_count = 0;
	int shell_element_count = 0;
	int lig_count = 0;

	while (element_type_vec[solid_element_count] == 24.0f) {
		element_ind[solid_element_count * nodes_per_tet] = (int)element_ind_vector[solid_element_count *
nodes_per_tet];
		element_ind[solid_element_count * nodes_per_tet + 1] = (int)element_ind_vector[solid_element_count
* nodes_per_tet + 1];
		element_ind[solid_element_count * nodes_per_tet + 2] = (int)element_ind_vector[solid_element_count
* nodes_per_tet + 2];
		element_ind[solid_element_count * nodes_per_tet + 3] = (int)element_ind_vector[solid_element_count
* nodes_per_tet + 3];
		element_mat[solid_element_count] = int(element_mat_vec[solid_element_count]);
		element_type[solid_element_count] = (int)element_type_vec[solid_element_count];
		element_real[solid_element_count] = (int)element_real_vec[solid_element_count];
		element_sec[solid_element_count] = (int)element_sec_vec[solid_element_count];
		element_num[solid_element_count] = (int)element_num_vec[solid_element_count];
		solid_element_count++;
	}
```

```
	cout << "solid_element_count = " << solid_element_count << endl;

	while (element_type_vec[solid_element_count + met_element_count] == 27.0f) {
		//while (file_element_count > (solid_element_count + met_element_count + shell_element_count +
lig_count)){
		element_ind[(solid_element_count + met_element_count) * nodes_per_tet] =
(int)element_ind_vector[(solid_element_count + met_element_count) * nodes_per_tet];
		element_ind[(solid_element_count + met_element_count) * nodes_per_tet + 1] =
(int)element_ind_vector[(solid_element_count + met_element_count) * nodes_per_tet + 1];
		element_ind[(solid_element_count + met_element_count) * nodes_per_tet + 2] =
(int)element_ind_vector[(solid_element_count + met_element_count) * nodes_per_tet + 2];
		element_ind[(solid_element_count + met_element_count) * nodes_per_tet + 3] =
(int)element_ind_vector[(solid_element_count + met_element_count) * nodes_per_tet + 3];
		element_mat[(solid_element_count + met_element_count)] =
int(element_mat_vec[(solid_element_count + met_element_count)]);
		element_type[(solid_element_count + met_element_count)] =
(int)element_type_vec[(solid_element_count + met_element_count)];
		element_real[(solid_element_count + met_element_count)] =
(int)element_real_vec[(solid_element_count + met_element_count)];
		element_sec[(solid_element_count + met_element_count)] =
(int)element_sec_vec[(solid_element_count + met_element_count)];
		element_num[(solid_element_count + met_element_count)] =
(int)element_num_vec[(solid_element_count + met_element_count)];
		met_element_count++;
	}
	if (element_type_vec[(solid_element_count + met_element_count) + shell_element_count] == 24.0f) {
		cout << "Error in element order: please re-order in Excel" << endl;
		cin.get();
	}

	cout << "met_element_count = " << met_element_count << endl;

	//*****************************Adjust element_ind to *nodes_per_tet only (there will be empty unsused
spaces within the array - but quicker access to elements)*********************************************
	//*****************************Fill unused spaces with -
1*****************************************************************************************************
******************
	while (element_type_vec[solid_element_count + met_element_count + shell_element_count] == 26.0f) {
		element_ind[((solid_element_count + met_element_count + shell_element_count)*nodes_per_tet)] =
(int)element_ind_vector[((solid_element_count + met_element_count) + shell_element_count)*nodes_per_tet];
		element_ind[((solid_element_count + met_element_count + shell_element_count)*nodes_per_tet) + 1]
= (int)element_ind_vector[(((solid_element_count + met_element_count) + shell_element_count)*nodes_per_tet) + 1];
		element_ind[((solid_element_count + met_element_count + shell_element_count)*nodes_per_tet) + 2]
= (int)element_ind_vector[(((solid_element_count + met_element_count) + shell_element_count)*nodes_per_tet) + 2];
		element_ind[((solid_element_count + met_element_count + shell_element_count)*nodes_per_tet) + 3]
= -1;
		element_mat[(solid_element_count + met_element_count) + shell_element_count] =
(int)element_mat_vec[(solid_element_count + met_element_count) + shell_element_count];
		element_type[(solid_element_count + met_element_count) + shell_element_count] =
(int)element_type_vec[(solid_element_count + met_element_count) + shell_element_count];
		element_real[(solid_element_count + met_element_count) + shell_element_count] =
(int)element_real_vec[(solid_element_count + met_element_count) + shell_element_count];
		element_sec[(solid_element_count + met_element_count) + shell_element_count] =
(int)element_sec_vec[(solid_element_count + met_element_count) + shell_element_count];
		element_num[(solid_element_count + met_element_count) + shell_element_count] =
(int)element_num_vec[(solid_element_count + met_element_count) + shell_element_count];
		shell_element_count++;
	}
	if ((element_type_vec[(solid_element_count + met_element_count) + shell_element_count] == 24.0f) ||
(element_type_vec[solid_element_count + met_element_count + shell_element_count] == 27.0f)) {
		cout << "Error in element order: please re-order in Excel" << endl;
```

```
                cin.get();
        }

        cout << "shell_element_count = " << shell_element_count << endl;

        while (file_element_count > ((solid_element_count + met_element_count) + shell_element_count + lig_count)) {
                if (element_type_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count]
== 25.0f) {
                        element_ind[((solid_element_count + met_element_count + shell_element_count +
lig_count)*nodes_per_tet)]
                                = (int)element_ind_vector[((solid_element_count + met_element_count) +
shell_element_count + lig_count)*nodes_per_tet];
                        element_ind[((solid_element_count + met_element_count + shell_element_count +
lig_count)*nodes_per_tet) + 1]
                                = (int)element_ind_vector[(((solid_element_count + met_element_count) +
shell_element_count + lig_count)*nodes_per_tet) + 1];
                        element_ind[((solid_element_count + met_element_count + shell_element_count +
lig_count)*nodes_per_tet) + 2] = -1;
                        element_ind[((solid_element_count + met_element_count + shell_element_count +
lig_count)*nodes_per_tet) + 3] = -1;
                        element_mat[(solid_element_count + met_element_count) + shell_element_count + lig_count]
= (int)element_mat_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count];
                        element_type[(solid_element_count + met_element_count) + shell_element_count +
lig_count] = (int)element_type_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count];
                        element_real[(solid_element_count + met_element_count) + shell_element_count + lig_count]
= (int)element_real_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count];
                        element_sec[(solid_element_count + met_element_count) + shell_element_count + lig_count]
= (int)element_sec_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count];
                        element_num[(solid_element_count + met_element_count) + shell_element_count +
lig_count] = (int)element_num_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count];
                }
                else if ((element_type_vec[(solid_element_count + met_element_count) + shell_element_count +
lig_count] == 24.0f) || (element_type_vec[(solid_element_count + met_element_count) + shell_element_count + lig_count]
== 26.0f)) {
                        cout << "Error in element order: please re-order in Excel" << endl;
                        cin.get();
                        break;
                }
                lig_count++;
        }

        cout << "lig_count = " << lig_count << endl;

        // total count
        int total_element_count = (solid_element_count + met_element_count) + shell_element_count + lig_count;
        printf("total_element_count = %i | count2 = %i | file_element_size = %i\n", total_element_count, count2,
file_element_count);

        float x12, x13, x14, x23, x24, x34;
        float y12, y13, y14, y23, y24, y34;
        float z12, z13, z14, z23, z24, z34;
        int temp_node;
        float detD = 0;
        float element_nloc[nodes_per_tet*global_dof];
        float* volume = new float[solid_element_count + met_element_count];

        for (int i = 0; i < (solid_element_count + met_element_count); i++) {
                for (int a = 0; a < nodes_per_tet; a++) {
                        for (int j = 0; j < global_dof; j++) {
                                element_nloc[a*global_dof + j] = node_loc_array[element_ind[(i*nodes_per_tet) +
a] * global_dof + j];
```

```
                        }
                }
                x12 = element_nloc[0 * global_dof] - element_nloc[1 * global_dof];
                x13 = element_nloc[0 * global_dof] - element_nloc[2 * global_dof];
                x14 = element_nloc[0 * global_dof] - element_nloc[3 * global_dof];
                x23 = element_nloc[1 * global_dof] - element_nloc[2 * global_dof];
                x24 = element_nloc[1 * global_dof] - element_nloc[3 * global_dof];
                x34 = element_nloc[2 * global_dof] - element_nloc[3 * global_dof];
                y12 = element_nloc[0 * global_dof + 1] - element_nloc[1 * global_dof + 1];
                y13 = element_nloc[0 * global_dof + 1] - element_nloc[2 * global_dof + 1];
                y14 = element_nloc[0 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                y23 = element_nloc[1 * global_dof + 1] - element_nloc[2 * global_dof + 1];
                y24 = element_nloc[1 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                y34 = element_nloc[2 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                z12 = element_nloc[0 * global_dof + 2] - element_nloc[1 * global_dof + 2];
                z13 = element_nloc[0 * global_dof + 2] - element_nloc[2 * global_dof + 2];
                z14 = element_nloc[0 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                z23 = element_nloc[1 * global_dof + 2] - element_nloc[2 * global_dof + 2];
                z24 = element_nloc[1 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                z34 = element_nloc[2 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                detD = -(x12*y13*z14) - (x13*y14*z12) - (x14*y12*z13) + (x14*y13*z12) + (x13*y12*z14) +
(x12*y14*z13);
                volume[i] = detD / 6.0f;
                if (volume[i] <= 0.0f) {
                        temp_node = element_ind[(i*nodes_per_tet)];
                        element_ind[(i*nodes_per_tet)] = element_ind[(i*nodes_per_tet) + 1];
                        element_ind[(i*nodes_per_tet) + 1] = temp_node;
                        for (int a = 0; a < nodes_per_tet; a++) {
                                for (int j = 0; j < global_dof; j++) {
                                        element_nloc[a*global_dof + j] =
node_loc_array[element_ind[(i*nodes_per_tet) + a] * global_dof + j];
                                }
                        }
                        x12 = element_nloc[0 * global_dof] - element_nloc[1 * global_dof];
                        x13 = element_nloc[0 * global_dof] - element_nloc[2 * global_dof];
                        x14 = element_nloc[0 * global_dof] - element_nloc[3 * global_dof];
                        x23 = element_nloc[1 * global_dof] - element_nloc[2 * global_dof];
                        x24 = element_nloc[1 * global_dof] - element_nloc[3 * global_dof];
                        x34 = element_nloc[2 * global_dof] - element_nloc[3 * global_dof];
                        y12 = element_nloc[0 * global_dof + 1] - element_nloc[1 * global_dof + 1];
                        y13 = element_nloc[0 * global_dof + 1] - element_nloc[2 * global_dof + 1];
                        y14 = element_nloc[0 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                        y23 = element_nloc[1 * global_dof + 1] - element_nloc[2 * global_dof + 1];
                        y24 = element_nloc[1 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                        y34 = element_nloc[2 * global_dof + 1] - element_nloc[3 * global_dof + 1];
                        z12 = element_nloc[0 * global_dof + 2] - element_nloc[1 * global_dof + 2];
                        z13 = element_nloc[0 * global_dof + 2] - element_nloc[2 * global_dof + 2];
                        z14 = element_nloc[0 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                        z23 = element_nloc[1 * global_dof + 2] - element_nloc[2 * global_dof + 2];
                        z24 = element_nloc[1 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                        z34 = element_nloc[2 * global_dof + 2] - element_nloc[3 * global_dof + 2];
                        detD = -(x12*y13*z14) - (x13*y14*z12) - (x14*y12*z13) + (x14*y13*z12) + (x13*y12*z14)
+ (x12*y14*z13);
                        volume[i] = detD / 6.0f;
                }
                if (volume[i] <= 0.0f) {
                        cout << "Still Bad volume!!!! " << i << ": " << volume[i] << endl;
                        printf("Element nodes: %i %i %i %i\n", element_ind[i*nodes_per_tet],
element_ind[(i*nodes_per_tet) + 1], element_ind[(i*nodes_per_tet) + 2], element_ind[(i*nodes_per_tet) + 3]);
                        printf("Node location 1: %6.2f %6.2f %6.2f\n", element_nloc[0], element_nloc[1],
element_nloc[2]);
```

```
                                printf("Node location 2: %6.2f %6.2f %6.2f\n", element_nloc[1 * global_dof], element_nloc[1
* global_dof + 1], element_nloc[1 * global_dof + 2]);
                                printf("Node location 3: %6.2f %6.2f %6.2f\n", element_nloc[2 * global_dof], element_nloc[2
* global_dof + 1], element_nloc[2 * global_dof + 2]);
                                printf("Node location 4: %6.2f %6.2f %6.2f\n", element_nloc[3 * global_dof], element_nloc[3
* global_dof + 1], element_nloc[3 * global_dof + 2]);
                                printf("Node location 1 before: %6.2f %6.2f %6.2f\n",
node_loc_array[element_ind[(i*nodes_per_tet) + 0] * global_dof + 0], node_loc_array[element_ind[(i*nodes_per_tet) + 0]
* global_dof + 1],
                                        node_loc_array[element_ind[(i*nodes_per_tet) + 0] * global_dof + 2]);
                                cin.get();
                        }
                }
                // Eliminate shell elements but modify the tets attached to shell elements
                bool shell_flag1 = false;
                bool shell_flag2 = false;
                bool shell_flag3 = false;
                for (int i = 0; i < solid_element_count; i++) {
                        shell_flag1 = false;
                        shell_flag2 = false;
                        shell_flag3 = false;
                        if (element_mat[i] == 3) {
                                for (int j = (solid_element_count + met_element_count); j < (shell_element_count +
solid_element_count + met_element_count); j++) {
                                        for (int k = 0; k < nodes_per_tet; k++) {
                                                for (int l = 0; l < nodes_per_shell; l++) {
                                                        if ((element_ind[i*nodes_per_tet + k] ==
element_ind[j*nodes_per_tet + l]) && (shell_flag1 == false)) {
                                                                shell_flag1 = true;
                                                                break;
                                                        }
                                                        else if ((shell_flag1 == true) && (shell_flag2 == false) &&
(element_ind[i*nodes_per_tet + k] == element_ind[j*nodes_per_tet + l])) {
                                                                shell_flag2 = true;
                                                                break;
                                                        }
                                                        else if ((shell_flag1 == true) && (shell_flag2 == true) &&
(element_ind[i*nodes_per_tet + k] == element_ind[j*nodes_per_tet + l])) {
                                                                shell_flag3 = true;
                                                                break;
                                                        }
                                                }
                                                if ((shell_flag1 == true) && (shell_flag2 == true) && (shell_flag3 ==
true)) break;
                                        }
                                        if ((shell_flag1 == true) && (shell_flag2 == true) && (shell_flag3 == true)) {
                                                element_mat[i] = 26;
                                                break;
                                        }
                                }
                        }
                }
                // Adjust BC_nodes
                int* bc_nodes = new int[BC_count];
                for (int i = 0; i < BC_count; i++) {
                        bc_nodes[i] = bc_nodes_vec[i];
                }
                // Re-organize element order for efficient adding of stiffness dof - avoid atomic add by considering BLOCK_SIZE
                int node_ind = 0;
                bool block_flag = false;
                bool node_flag = false;
```

```
	bool node_flag2 = false;
	int swap_buff = 0;
	int node_ind2 = 0;
	bool first_run = false;
	int itr_count = 0;
	int block_ind = 0;
	int allowed_count = 0; // number of allowed node conflicts
	int conflict_count1 = 0;
	int conflict_count2 = 0;
	int GRID_SIZE1 = (total_element_count / BLOCK_SIZE) + 1;
	while (!block_flag) {
		for (int i = 0; i < GRID_SIZE1; i++){
			block_ind = i;
			for (int j = 1; j < BLOCK_SIZE; j++){
				node_flag = true;
				node_ind = i*BLOCK_SIZE + j;
				if (node_ind >= total_element_count) break;
				first_run = true;
				allowed_count = 0;
				conflict_count1 = 0;
				conflict_count2 = 0;
				block_flag = true;
				while (node_flag && block_flag) {
					node_flag2 = true;
					while (node_flag2 && block_flag){
						node_flag2 = false;
						for (int k = 0; k < BLOCK_SIZE; k++){
							if (k != j){
								node_ind2 = i*BLOCK_SIZE + k;
								if (node_ind2 >= total_element_count)
break;
								for (int l = 0; l < nodes_per_tet; l++){
									for (int m = 0; m < nodes_per_tet;
m++){
										if
(element_ind[node_ind*nodes_per_tet + l] == element_ind[node_ind2*nodes_per_tet + m]){

	conflict_count1++;
											if
(conflict_count1 > allowed_count){

	node_flag2 = true;

	block_ind++;

	node_ind = block_ind*BLOCK_SIZE;

	conflict_count1 = 0;
	if
(node_ind >= total_element_count) {

	node_ind += BLOCK_SIZE;

	node_ind %= GRID_SIZE1*BLOCK_SIZE;

	block_ind = 0;
	}
	if
(block_ind == i) block_flag = false;

	break;
```

```
                                                                                                }
                                                                                        }
                                                                                }
                                                                                if (node_flag2 == true) break;
                                                                        }
                                                                        if (node_flag2 == true) break;
                                                                }
                                                        }
                                                }
                                                node_flag = false;
                                                for (int k = 0; k < BLOCK_SIZE; k++){
                                                        if (k != j){
                                                                node_ind2 = block_ind*BLOCK_SIZE + k;
                                                                if (node_ind2 >= total_element_count) break;
                                                                for (int l = 0; l < nodes_per_tet; l++){
                                                                        for (int m = 0; m < nodes_per_tet; m++){
                                                                                if
(element_ind[node_ind2*nodes_per_tet + l] == element_ind[(i*BLOCK_SIZE + j)*nodes_per_tet + m]){
                                                                                        conflict_count2++;
                                                                                        if (conflict_count2 >
allowed_count){
                                                                                                node_flag =
true;
                                                                                                block_ind++;
                                                                                                node_ind =
block_ind*BLOCK_SIZE;

                                                        conflict_count2 = 0;
                                                                                                if (node_ind
>= total_element_count) {

                                                        node_ind += BLOCK_SIZE;

                                                        node_ind %= GRID_SIZE1*BLOCK_SIZE;

                                                        block_ind = 0;
                                                                                                }
                                                                                                if (block_ind
== i) block_flag = false;
                                                                                                break;
                                                                                        }
                                                                                }
                                                                        }
                                                                        if (node_flag == true) break;
                                                                }
                                                                if (node_flag == true) break;
                                                        }
                                                }
                                                if ((node_flag || node_flag2) && (!block_flag)){
                                                        allowed_count++;
                                                        block_flag = true;
                                                }
                                        }
                                        for (int k = 0; k < nodes_per_tet; k++) {
                                                swap_buff = element_ind[node_ind*nodes_per_tet + k];
                                                element_ind[node_ind*nodes_per_tet + k] =
element_ind[(i*BLOCK_SIZE + j)*nodes_per_tet + k];
                                                element_ind[(i*BLOCK_SIZE + j)*nodes_per_tet + k] = swap_buff;
                                        }
                                        swap_buff = element_mat[node_ind];
                                        element_mat[node_ind] = element_mat[(i*BLOCK_SIZE + j)];
```

```
                                        element_mat[(i*BLOCK_SIZE + j)] = swap_buff;
                                        swap_buff = element_type[node_ind];
                                        element_type[node_ind] = element_type[(i*BLOCK_SIZE + j)];
                                        element_type[(i*BLOCK_SIZE + j)] = swap_buff;
                                        swap_buff = element_sec[node_ind];
                                        element_sec[node_ind] = element_sec[(i*BLOCK_SIZE + j)];
                                        element_sec[(i*BLOCK_SIZE + j)] = swap_buff;
                                        swap_buff = element_real[node_ind];
                                        element_real[node_ind] = element_real[(i*BLOCK_SIZE + j)];
                                        element_real[(i*BLOCK_SIZE + j)] = swap_buff;
                                }
                        }
                        block_flag = true;
                        if (itr_count > 10) block_flag = true;
                        else {
                                for (int i = 0; i < GRID_SIZE1; i++){
                                        for (int j = 0; j < BLOCK_SIZE; j++){
                                                for (int k = j + 1; k < BLOCK_SIZE; k++){
                                                        if ((i*BLOCK_SIZE + k) >= total_element_count) break;
                                                        for (int l = 0; l < nodes_per_tet; l++){
                                                                for (int m = 0; m < nodes_per_tet; m++){
                                                                        if (element_ind[(i*BLOCK_SIZE +
j)*nodes_per_tet + l] == element_ind[(i*BLOCK_SIZE + k)*nodes_per_tet + m]){
                                                                                block_flag = false;
                                                                                goto err_skip;
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                err_skip:
                        itr_count++;
                        cout << "itr " << itr_count << endl;
                }

                // Double check
                if (outfile_err.is_open()){
                        outfile_err << "Error File" << endl;
                        for (int i = 0; i < GRID_SIZE1; i++){
                                for (int j = 0; j < BLOCK_SIZE; j++){
                                        for (int k = j + 1; k < BLOCK_SIZE; k++){
                                                if ((i*BLOCK_SIZE + k) >= total_element_count) break;
                                                for (int l = 0; l < nodes_per_tet; l++){
                                                        for (int m = 0; m < nodes_per_tet; m++){
                                                                if (element_ind[(i*BLOCK_SIZE + j)*nodes_per_tet
+ l] == element_ind[(i*BLOCK_SIZE + k)*nodes_per_tet + m]){
                                                                        //printf("issue with block %i and element
(within block) %i\n", i, j / nodes_per_tet);
                                                                        outfile_err << "issue with block " << i << "
and element (within block) " << j << endl;
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                        outfile_err.close();
                }
                else cout << "Error file not open" << endl;
```

```
	printf("Done the BLOCK check\n");

	//*****************************************For no reorganization of the
element_ind_buff*****************************************************************
	vector<int> element_ind_buff((buff_size - shell_element_count)*nodes_per_tet
		+ ((BLOCK_SIZE - (((buff_size - shell_element_count)) % BLOCK_SIZE)))*nodes_per_tet, -1);
	vector<int> element_mat_buff(buff_size - shell_element_count + (BLOCK_SIZE - ((buff_size -
shell_element_count) % BLOCK_SIZE)), -1);
	vector<int> element_type_buff((buff_size - shell_element_count) + (BLOCK_SIZE - ((buff_size -
shell_element_count) % BLOCK_SIZE)), -1);
	vector<int> element_real_buff((buff_size - shell_element_count) + (BLOCK_SIZE - ((buff_size -
shell_element_count) % BLOCK_SIZE)), -1);
	vector<int> element_sec_buff((buff_size - shell_element_count) + (BLOCK_SIZE - ((buff_size -
shell_element_count) % BLOCK_SIZE)), -1);
	vector<int>		tet_fiber_ind_buff(((buff_size - shell_element_count) + (BLOCK_SIZE - ((buff_size -
shell_element_count) % BLOCK_SIZE))), -1);
	node_ind = 0;
	for (int i = 0; i < (buff_size); i++) {
		if (element_type[i] != 26) {
			for (int j = 0; j < nodes_per_tet; j++) {
				element_ind_buff[node_ind*nodes_per_tet + j] = element_ind[i*nodes_per_tet + j];
			}
			element_mat_buff[node_ind] = element_mat[i];
			//if (element_type[i] == 24) element_type_buff[node_ind] = 27;
			//else element_type_buff[node_ind] = element_type[i];
			element_type_buff[node_ind] = element_type[i];
			element_real_buff[node_ind] = element_real[i];
			element_sec_buff[node_ind] = element_sec[i];
			//tet_fiber_ind_buff[i] = tet_fiber_ind_vec[i];
			node_ind++;
		}
	}

	total_element_count = buff_size - shell_element_count + (BLOCK_SIZE - ((buff_size - shell_element_count) %
BLOCK_SIZE));
	int actual_element_count = node_ind;
	printf("total_element_count = %i, actual_element_count = %i\n", total_element_count, actual_element_count);
	//*************************************************************************************************
*************************************************************************

	total_dof = node_count*global_dof;

	// Prep for CSR storage
	vector< vector<int> > element_csr(node_count);
	int* node_type = new int[node_count];
	for (int i = 0; i < node_count; i++) {
		node_type[i] = -1;
	}
	int node_ind3 = 0;
	bool csr_flag = false;
	for (int i = 0; i < total_element_count; i++) {
		for (int j = 0; j < nodes_per_tet; j++) {
			node_ind = element_ind_buff[i*nodes_per_tet + j];
			for (int k = 0; k < nodes_per_tet; k++) {
				node_ind3 = element_ind_buff[i*nodes_per_tet + k];
				if ((node_ind != -1) && (node_ind3 != -1))
element_csr[node_ind].push_back(node_ind3);
			}
		}
	}
```

```
	vector< unordered_set<int> > s(node_count);
	size_t size;
	for (int j = 0; j < node_count; j++) {
		for (int i : element_csr[j])
			s[j].insert(i);
		element_csr[j].assign(s[j].begin(), s[j].end());
		sort(element_csr[j].begin(), element_csr[j].end());
	}

	// Check to see if every node is there
	for (int k = 0; k < node_count; k++) {
		csr_flag = false;
		//for (int i = 0; i < node_count; i++) {
		size = element_csr[k].size();
		for (unsigned j = 0; j < size; j++) {
			if (k == element_csr[k][j]) {
				csr_flag = true;
				break;
			}
		}
		//if (csr_flag == true) break;
		//}
		if (csr_flag == false) {
			printf("OMG ERROR. node %i. Press enter\n", k);
			cin.get();
		}
	}

	vector<int> stiff_IA;
	vector<int> stiff_JA;
	stiff_IA.push_back(0);
	int IA_count = 0;
	for (int i = 0; i < node_count; i++) {
		for (int o = 0; o < global_dof; o++) {
			IA_count = 0;
			size = element_csr[i].size();
			for (int j = 0; j < (int)size; j++) {
				node_ind = element_csr[i][j];
				for (int l = 0; l < global_dof; l++) {
					if ((node_ind*global_dof + l) >= (i*global_dof + o)) { // uncomment for
sym
						IA_count++;
						stiff_JA.push_back(node_ind*global_dof + l);
					}
				}
			}
			stiff_IA.push_back(stiff_IA[i*global_dof + o] + IA_count);
		}
	}

	int* stiff_JA_buff = new int[stiff_IA[total_dof]];
	for (int i = 0; i < stiff_IA[total_dof]; i++) {
		stiff_JA_buff[i] = stiff_JA[i];
	}
	FILE* csr_col_file;
	errno_t err;
	if ((err = fopen_s(&csr_col_file, "CSR_COLUMN.binary", "wb")) != 0)
		printf("CSR_COLUMN file not opened\n");
	else fwrite(stiff_JA_buff, 1, stiff_IA[total_dof] * sizeof(int), csr_col_file);
	fclose(csr_col_file);
```

```
	int* stiff_IA_buff = new int[total_dof + 1];
	for (int i = 0; i < (total_dof + 1); i++) stiff_IA_buff[i] = stiff_IA[i];
	FILE* csr_row_file;
	if ((err = fopen_s(&csr_row_file, "CSR_ROW.binary", "wb")) != 0)
		printf("CSR_ROW file not opened\n");
	else fwrite(stiff_IA_buff, 1, (total_dof + 1) * sizeof(int), csr_row_file);
	fclose(csr_row_file);

	////Check element_ind_buff
	//vector<int> e_check;
	//bool e_c_flag = false;
	//for (int k = 0; k < p_node_count; k++){
	//	e_check.push_back(-2);
	//}
	//for (int i = 0; i < (int)element_ind_buff.size(); i++){
	//	if ((element_ind_buff[i] >= 0) && (element_ind_buff[i] < p_node_count)){
	//		e_check[element_ind_buff[i]] = element_ind_buff[i];
	//	}
	//	else if ((element_ind_buff[i] != -1) && (element_type_buff[i/nodes_per_tet] == 27))
	//		printf("error at location %i with value %i\n", i, element_ind_buff[i]);
	//}
	//for (int k = 0; k < p_node_count; k++){
	//	if (e_check[k] != k){
	//		e_c_flag = true;
	//		printf("element k = %i is missing.  e_check[k] = %i\n", k, e_check[k]);
	//	}
	//}
	//if (e_c_flag == true){
	//	cout << "some elements are missing. See above" << endl;
	//}
	//else cout << "all elements are fine" << endl;

	//vector<int>		af_ind_buff((((int)element_sec_vec.size() + f_elem_count) + (BLOCK_SIZE -
(((int)element_sec_vec.size() + f_elem_count) % BLOCK_SIZE)))*nodes_per_fib, -1);
	//Create ****INDEX***** array for the CSR stiffness matrix array for storing stiffness non-zero values
	int ematrix_ref_size = tet_element_dof*tet_element_dof;
	int stiff_ind_j_size = total_element_count*ematrix_ref_size; // 144 indexes for each element
	int* stiff_ind_j = new int[stiff_ind_j_size];
	node_ind2 = 0;
	int node_ind_ind = 0;
	int node_ind_ind2 = 0;
	int node_ind_m = 0;
	int node_ind_n = 0;
	int node_ind_j = 0;
	int node_ind_l = 0;
	bool stiff_ind_flag = false;
	for (int i = 0; i < total_element_count; i++) {
		if (element_type_buff[i] == 24) {
			for (int j = 0; j < nodes_per_tet; j++) {
				node_ind = element_ind_buff[i*nodes_per_tet + j];
				for (int m = 0; m < global_dof; m++) {
					for (int l = j; l < nodes_per_tet; l++) { // j for sym
						node_ind2 = element_ind_buff[i*nodes_per_tet + l];
						for (int n = 0; n < global_dof; n++) {
							if (node_ind > node_ind2) {
								node_ind_ind = node_ind2;
								node_ind_ind2 = node_ind;
								node_ind_m = n;
								node_ind_n = m;
								node_ind_j = l;
								node_ind_l = j;
```

```
                                                                }
                                                                else { node_ind_ind = node_ind; node_ind_ind2 =
node_ind2; node_ind_m = m; node_ind_n = n; node_ind_j = j; node_ind_l = l; }
                                                                //node_ind_ind = node_ind; node_ind_ind2 =
node_ind2; node_ind_m = m; node_ind_n = n; node_ind_j = j; node_ind_l = l; // comment out for sym
                                                                for (int k = stiff_IA[node_ind_ind*global_dof +
node_ind_m]; k < stiff_IA[node_ind_ind*global_dof + node_ind_m + 1]; k++) {
                                                                        if ((node_ind_ind2*global_dof +
node_ind_n) == stiff_JA[k]) {
                                                                                stiff_ind_j[(i * ematrix_ref_size) +
((j*global_dof + m) * tet_element_dof) + l*global_dof + n] = k;
                                                                                break;
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                                for (int j = (tet_element_dof*tet_element_dof); j < ematrix_ref_size; j++) {
                                        stiff_ind_j[(i*ematrix_ref_size) + j] = -1;
                                }
                        }
                        else if (element_type_buff[i] == 25) {
                                for (int j = 0; j < nodes_per_lig; j++) {
                                        node_ind = element_ind_buff[i*nodes_per_tet + j];
                                        for (int m = 0; m < global_dof; m++) {
                                                for (int l = j; l < nodes_per_lig; l++) { // j
                                                        node_ind2 = element_ind_buff[i*nodes_per_tet + l];
                                                        for (int n = 0; n < global_dof; n++) {
                                                                if (node_ind > node_ind2) {
                                                                        node_ind_ind = node_ind2;
                                                                        node_ind_ind2 = node_ind;
                                                                        node_ind_m = n;
                                                                        node_ind_n = m;
                                                                        node_ind_j = l;
                                                                        node_ind_l = j;
                                                                }
                                                                else { node_ind_ind = node_ind; node_ind_ind2 =
node_ind2; node_ind_m = m; node_ind_n = n; node_ind_j = j; node_ind_l = l; }
                                                                //node_ind_ind = node_ind; node_ind_ind2 =
node_ind2; node_ind_m = m; node_ind_n = n; node_ind_j = j; node_ind_l = l; // comment out for sym
                                                                for (int k = stiff_IA[node_ind_ind*global_dof +
node_ind_m]; k < stiff_IA[node_ind_ind*global_dof + node_ind_m + 1]; k++) {
                                                                        if ((node_ind_ind2*global_dof +
node_ind_n) == stiff_JA[k]) {
                                                                                stiff_ind_j[(i * ematrix_ref_size) +
((j*global_dof + m) * lig_element_dof) + l*global_dof + n] = k;
                                                                                break;
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                                for (int j = (lig_element_dof*lig_element_dof); j < ematrix_ref_size; j++) {
                                        stiff_ind_j[(i*ematrix_ref_size) + j] = -1;
                                }
                        }
                        else if ((element_type_buff[i] == -1)) { //Need to change this once force elements are incorporated
                                for (int j = 0; j < ematrix_ref_size; j++) {
                                        stiff_ind_j[(i * ematrix_ref_size) + j] = -1;
```

```
                }
        }
        else printf("unknown element type for i=%i\n", i);
}
////check if k from 0 to A_non_zeros
//for (int k = 0; k < stiff_IA[(int)stiff_IA.size() - 1]; k++){
//      stiff_ind_flag = false;
//      for (int l = 0; l < ematrix_ref_size*(int)element_mat_buff.size(); l++){
//              if (k == stiff_ind_j[l]){
//                      stiff_ind_flag = true;
//                      break;
//              }
//      }
//      if (stiff_ind_flag == false){
//              printf("k = %i is missing from stiff_ind_j\n", k);
//      }
//}

//stiff_ind_flag = false;
//for (int i = 0; i < (int)element_mat_buff.size()*ematrix_ref_size; i++) {
//      if ((stiff_ind_j[i] >= stiff_IA[(int)stiff_IA.size() - 1]) || (stiff_ind_j[i] < -1)) {
//              stiff_ind_j[i] = -1;
//              //printf("k = %i is missing from stiff_ind_j\n", i);
//              //stiff_ind_flag = true;
//      }
//}

//if (stiff_ind_flag == true) {
//      printf("FAILURE occurred, see above for specific dof in the element array\n");
//      goto Error;
//}

int* element_i_buff = new int[element_mat_buff.size()*(8)];
for (int i = 0; i < (int)element_mat_buff.size(); i++) {
        for (int j = 0; j < nodes_per_tet; j++) element_i_buff[i * 8 + j] = element_ind_buff[i*nodes_per_tet + j];
        element_i_buff[i * 8 + 4] = element_mat_buff[i];
        element_i_buff[i * 8 + 5] = element_type_buff[i];
        element_i_buff[i * 8 + 6] = element_real_buff[i];
        element_i_buff[i * 8 + 7] = element_sec_buff[i];
}
FILE* element_i_file;
if ((err = fopen_s(&element_i_file, "ELEMENT_IND_I.binary", "wb")) != 0)
        printf("ELEMENT_IND_I file not opened\n");
else fwrite(element_i_buff, 1, element_mat_buff.size() * 8 * sizeof(int), element_i_file);
fclose(element_i_file);

FILE* stiff_ind_j_file;
if ((err = fopen_s(&stiff_ind_j_file, "ELEMENT_IND_J.binary", "wb")) != 0)
        printf("ELEMENT_IND_J file not opened\n");
else fwrite(stiff_ind_j, 1, stiff_ind_j_size * sizeof(int), stiff_ind_j_file);
fclose(stiff_ind_j_file);

FILE* bc_nodes_adj_file;
if ((err = fopen_s(&bc_nodes_adj_file, "BC_NODES_PRESSURE.binary", "wb")) != 0)
        printf("BC_NODES_PRESSURE file not opened\n");
else fwrite(bc_nodes, 1, BC_count * sizeof(int), bc_nodes_adj_file);
fclose(bc_nodes_adj_file);

for (int i = af_count; i < total_element_count; i++) {
        af_nodes_vec.push_back(-1);
        af_nodes_vec.push_back(-1);
```

```
	}
	if (outfile8.is_open()) {
		outfile8 << "AF_IND_PRESSURE" << endl;
		for (int i = 0; i < (int)element_mat_buff.size() - 1; i++) {
			for (int j = 0; j < nodes_per_fib; j++) {
				outfile8.width(8);
				outfile8 << af_nodes_vec[i*nodes_per_fib + j];
			}
			outfile8 << endl;
		}
		for (int j = 0; j < nodes_per_fib; j++) {
			outfile8.width(8);
			outfile8 << af_nodes_vec[((int)element_mat_buff.size() - 1)*nodes_per_fib + j];
		}
		outfile8.close();
	}
	else cout << "Unable to create file8" << endl;

	//**************************************************Vector assembly
matrices**************************************
	//Create arrays for fast vector assembly using shared memory
	int GRID_SIZE = ((int)element_mat_buff.size() / BLOCK_SIZE) + 1;
	int* share_local_element_ind = new int[GRID_SIZE*BLOCK_SIZE*nodes_per_tet];
	int* share_global_element_ind = new int[GRID_SIZE*BLOCK_SIZE*nodes_per_tet];
	int* vec = NULL;

	{
		for (int i = 0; i < GRID_SIZE*BLOCK_SIZE*nodes_per_tet; i++) {
			share_global_element_ind[i] = 0;
			share_local_element_ind[i] = 0;
		}
	}

	{
		for (int a = 0; a < GRID_SIZE; a++) {
			for (int i = 0; i < BLOCK_SIZE; i++) {
				if ((a*BLOCK_SIZE + i)<(int)element_mat_buff.size()) {
					for (int j = 0; j < nodes_per_tet; j++) {
						if (element_ind_buff[((a*BLOCK_SIZE + i)*nodes_per_tet) +
j] > max) {
							max = element_ind_buff[((a*BLOCK_SIZE +
i)*nodes_per_tet) + j];
						}
					}
				}
			}

			vec = new int[max + 1];
			fill(vec, vec + max + 1, 0);
			temp = 0;
			{
				for (int i = 0; i < BLOCK_SIZE; i++) {
					if ((a*BLOCK_SIZE + i)<(int)element_mat_buff.size()) {
						for (int j = 0; j < nodes_per_tet; j++) {
							ind = element_ind_buff[((a*BLOCK_SIZE +
i)*nodes_per_tet) + j];
							if (vec[ind]) {
								share_local_element_ind[((a*BLOCK_SIZE
+ i)*nodes_per_tet) + j] = vec[ind];

	share_global_element_ind[((a*BLOCK_SIZE + i)*nodes_per_tet) + j] = 0;
```

```
                                                                        }
                                                                        else {
                                                                                share_local_element_ind[((a*BLOCK_SIZE
+ i)*nodes_per_tet) + j] = temp;

                share_global_element_ind[((a*BLOCK_SIZE + i)*nodes_per_tet) + j] = 1;
                                                                                vec[ind] = temp;
                                                                                temp++;
                                                                        }
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }

                FILE* share_assembly_file;
                if ((err = fopen_s(&share_assembly_file, "SHARED_ASSEMBLY_IND.binary", "wb")) != 0)
                        printf("SHARED_ASSEMBLY_IND file not opened\n");
                else fwrite(share_local_element_ind, 1, GRID_SIZE*BLOCK_SIZE*nodes_per_tet * sizeof(int),
share_assembly_file);
                fclose(share_assembly_file);

                if (outfile2.is_open()) {
                        outfile2 << "SHARED_GLOBAL_ASSEMBLY_IND" << endl;
                        for (int i = 0; i < BLOCK_SIZE*GRID_SIZE - 1; i++) {
                                for (int j = 0; j < nodes_per_tet; j++) {
                                        outfile2.width(12);
                                        outfile2 << left << share_global_element_ind[i * nodes_per_tet + j];
                                }
                                outfile2 << endl;
                        }
                        for (int j = 0; j < nodes_per_tet; j++) {
                                outfile2.width(12);
                                outfile2 << left << share_global_element_ind[(BLOCK_SIZE*GRID_SIZE - 1) *
nodes_per_tet + j];
                        }
                        outfile2.close();
                }
                else cout << "Unable to create file" << endl;

                vector<float>().swap(element_ind_vector);
                vector<float>().swap(element_type_vec);
                vector<float>().swap(element_num_vec);
                vector<float>().swap(node_loc);
                vector<int>().swap(f_nodes_vec);

                cout << "Done!!" << endl;
                cin.get();

                return 0;
}
```

```
// CUDA headers
#define _USE_MATH_DEFINES
#include <cuda.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cublas_v2.h>
#define BLOCK_SIZE 128

// Windows headers
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>

// MKL headers
#include <stdlib.h>
#include "mkl.h"
#include "mkl_types.h"
#include "mkl_blas.h"
#include "mkl_rci.h"
#include "mkl_service.h"
#include "math.h"

#include "mkl_pardiso.h"

using namespace std;
using std::cin;
using std::cout;
using std::endl;

// Global variables
const int global_dof = 3;
const int nodes_per_tet = 4;
const int nodes_per_lig = 2;
const int nodes_per_fib = 2;
const int tet_element_dof = 12;
const int force_node_dof = 6;
const float substep_size = 0.33333333333333333333333333333333333333333333333f;
const int ematrix_ref_size = tet_element_dof*tet_element_dof;
int total_node_count;
int solid_element_count;
int lig_count;
int blank_count;
int A_non_zeros;
int csr_row_size;
int BC_node_count;
int BC_stiff_count;
int F_node_count;
int total_dof;
int total_element_count;
int GRID_SIZE1;
int GRID_SIZE2;
int GRID_SIZE12;
int GRID_SIZE22;
```

```
int GRID_SIZE32;
int GRID_SIZE0;
int GRID_SIZE52;
int GRID_SIZE01;
int GRID_SIZE5;
int GRID_SIZE6;
int GRID_SIZE_BC;
int GRID_SIZE_NON_ZEROS;
int GRID_SIZE_ORIG;
int GRID_SIZE_ORIG1;

int GRID_SIZE_P;

// Non-cuda function declaration
cudaError_t RealTimeSim(float* nodes_x, int* element_i, int* element_j, int* csr_row, int* csr_col, int*
share_element_ind, float* ma_params,
	bool* share_bool_ind, int* type_ind, int* body_ind, int* sec_ind, int* BC_node_ind, int* F_ind, int*
BCapply_ind, float* BCapply_stiff, int* fib_ind, float* h_force);

//**************CUDA functions for calculating stiffness matrices (and add together to get matrix A for solver) and nodal
forces (to get vector b for solver)**************
__global__ void PreStiffCalculations(int* element_nodes, int* type_ind, int* body_ind, float* nodes, float* IntDeriv, float*
volumes, float* BL0, float* lig_lengths0,
	float* local_coords, float* local_coords0, int total_node_count) {
	const int size = 72;
	register float x_diff = 0;
	register float y_diff = 0;
	register float z_diff = 0;
	register int vb_node = 0;
	register int ground_node = 0;
	//for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < solid_element_count; ind += blockDim.x *
gridDim.x){
	int ind = blockIdx.x*blockDim.x + threadIdx.x;
	//if (ind < (solid_element_count + mixed_element_count + shell_element_count + lig_count)){
	if (type_ind[ind] == 24) { // change back to 24
		register float x12, x13, x14, x23, x24, x34;
		register float y12, y13, y14, y23, y24, y34;
		register float z12, z13, z14, z23, z24, z34;
		register float element_nloc[nodes_per_tet*global_dof];
		register float r_intderiv[tet_element_dof];
		register float detD;
		for (int a = 0; a < nodes_per_tet; a++) {
			for (int i = 0; i < global_dof; i++) {
				element_nloc[a*global_dof + i] = nodes[element_nodes[(ind*nodes_per_tet) + a] *
global_dof + i];
			}
		}
		// Calculate node differences
		x12 = element_nloc[0 * global_dof] - element_nloc[1 * global_dof];
		x13 = element_nloc[0 * global_dof] - element_nloc[2 * global_dof];
		x14 = element_nloc[0 * global_dof] - element_nloc[3 * global_dof];
		x23 = element_nloc[1 * global_dof] - element_nloc[2 * global_dof];
		x24 = element_nloc[1 * global_dof] - element_nloc[3 * global_dof];
		x34 = element_nloc[2 * global_dof] - element_nloc[3 * global_dof];
		y12 = element_nloc[0 * global_dof + 1] - element_nloc[1 * global_dof + 1];
		y13 = element_nloc[0 * global_dof + 1] - element_nloc[2 * global_dof + 1];
		y14 = element_nloc[0 * global_dof + 1] - element_nloc[3 * global_dof + 1];
		y23 = element_nloc[1 * global_dof + 1] - element_nloc[2 * global_dof + 1];
		y24 = element_nloc[1 * global_dof + 1] - element_nloc[3 * global_dof + 1];
		y34 = element_nloc[2 * global_dof + 1] - element_nloc[3 * global_dof + 1];
		z12 = element_nloc[0 * global_dof + 2] - element_nloc[1 * global_dof + 2];
```

```
z13 = element_nloc[0 * global_dof + 2] - element_nloc[2 * global_dof + 2];
z14 = element_nloc[0 * global_dof + 2] - element_nloc[3 * global_dof + 2];
z23 = element_nloc[1 * global_dof + 2] - element_nloc[2 * global_dof + 2];
z24 = element_nloc[1 * global_dof + 2] - element_nloc[3 * global_dof + 2];
z34 = element_nloc[2 * global_dof + 2] - element_nloc[3 * global_dof + 2];
// Calculate determinant of D matrix
detD = -(x12*y13*z14) - (x13*y14*z12) - (x14*y12*z13) + (x14*y13*z12) + (x13*y12*z14) +
(x12*y14*z13);
volumes[ind] = detD * 0.1666666666666666f;
{
        if (detD != 0.0f) {
                r_intderiv[0] = ((y24*z23) - (y23*z24)) / detD;
                r_intderiv[1] = ((x23*z24) - (x24*z23)) / detD;
                r_intderiv[2] = ((x24*y23) - (x23*y24)) / detD;
                r_intderiv[3] = ((y13*z34) - (y34*z13)) / detD;
                r_intderiv[4] = ((x34*z13) - (x13*z34)) / detD;
                r_intderiv[5] = ((x13*y34) - (x34*y13)) / detD;
                r_intderiv[6] = ((y24*z14) - (y14*z24)) / detD;
                r_intderiv[7] = ((x14*z24) - (x24*z14)) / detD;
                r_intderiv[8] = ((x24*y14) - (x14*y24)) / detD;
                r_intderiv[9] = (-(y13*z12) + (y12*z13)) / detD;
                r_intderiv[10] = (-(x12*z13) + (x13*z12)) / detD;
                r_intderiv[11] = (-(x13*y12) + (x12*y13)) / detD;
        }
        else if (detD < 0.0f) {
                printf("Error!! negative volume");
                r_intderiv[0] = ((y24*z23) - (y23*z24)) / detD;
                r_intderiv[1] = ((x23*z24) - (x24*z23)) / detD;
                r_intderiv[2] = ((x24*y23) - (x23*y24)) / detD;
                r_intderiv[3] = ((y13*z34) - (y34*z13)) / detD;
                r_intderiv[4] = ((x34*z13) - (x13*z34)) / detD;
                r_intderiv[5] = ((x13*y34) - (x34*y13)) / detD;
                r_intderiv[6] = ((y24*z14) - (y14*z24)) / detD;
                r_intderiv[7] = ((x14*z24) - (x24*z14)) / detD;
                r_intderiv[8] = ((x24*y14) - (x14*y24)) / detD;
                r_intderiv[9] = (-(y13*z12) + (y12*z13)) / detD;
                r_intderiv[10] = (-(x12*z13) + (x13*z12)) / detD;
                r_intderiv[11] = (-(x13*y12) + (x12*y13)) / detD;
        }
        else {
                printf("Error!! zero volume at ind = %i", ind);
                r_intderiv[0] = 0.0f;
                r_intderiv[1] = 0.0f;
                r_intderiv[2] = 0.0f;
                r_intderiv[3] = 0.0f;
                r_intderiv[4] = 0.0f;
                r_intderiv[5] = 0.0f;
                r_intderiv[6] = 0.0f;
                r_intderiv[7] = 0.0f;
                r_intderiv[8] = 0.0f;
                r_intderiv[9] = 0.0f;
                r_intderiv[10] = 0.0f;
                r_intderiv[11] = 0.0f;
        }
}
// Fill BL0 matrix from interpolation derivatives
BL0[(size*blockDim.x*blockIdx.x) + (0 * blockDim.x) + threadIdx.x] = r_intderiv[0];
BL0[(size*blockDim.x*blockIdx.x) + (1 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (2 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (3 * blockDim.x) + threadIdx.x] = r_intderiv[3];
BL0[(size*blockDim.x*blockIdx.x) + (4 * blockDim.x) + threadIdx.x] = 0.0f;
```

```
BL0[(size*blockDim.x*blockIdx.x) + (5 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (6 * blockDim.x) + threadIdx.x] = r_intderiv[6];
BL0[(size*blockDim.x*blockIdx.x) + (7 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (8 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (9 * blockDim.x) + threadIdx.x] = r_intderiv[9];
BL0[(size*blockDim.x*blockIdx.x) + (10 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (11 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (12 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (13 * blockDim.x) + threadIdx.x] = r_intderiv[1];
BL0[(size*blockDim.x*blockIdx.x) + (14 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (15 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (16 * blockDim.x) + threadIdx.x] = r_intderiv[4];
BL0[(size*blockDim.x*blockIdx.x) + (17 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (18 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (19 * blockDim.x) + threadIdx.x] = r_intderiv[7];
BL0[(size*blockDim.x*blockIdx.x) + (20 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (21 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (22 * blockDim.x) + threadIdx.x] = r_intderiv[10];
BL0[(size*blockDim.x*blockIdx.x) + (23 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (24 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (25 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (26 * blockDim.x) + threadIdx.x] = r_intderiv[2];
BL0[(size*blockDim.x*blockIdx.x) + (27 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (28 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (29 * blockDim.x) + threadIdx.x] = r_intderiv[5];
BL0[(size*blockDim.x*blockIdx.x) + (30 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (31 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (32 * blockDim.x) + threadIdx.x] = r_intderiv[8];
BL0[(size*blockDim.x*blockIdx.x) + (33 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (34 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (35 * blockDim.x) + threadIdx.x] = r_intderiv[11];
BL0[(size*blockDim.x*blockIdx.x) + (36 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (37 * blockDim.x) + threadIdx.x] = r_intderiv[2];
BL0[(size*blockDim.x*blockIdx.x) + (38 * blockDim.x) + threadIdx.x] = r_intderiv[1];
BL0[(size*blockDim.x*blockIdx.x) + (39 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (40 * blockDim.x) + threadIdx.x] = r_intderiv[5];
BL0[(size*blockDim.x*blockIdx.x) + (41 * blockDim.x) + threadIdx.x] = r_intderiv[4];
BL0[(size*blockDim.x*blockIdx.x) + (42 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (43 * blockDim.x) + threadIdx.x] = r_intderiv[8];
BL0[(size*blockDim.x*blockIdx.x) + (44 * blockDim.x) + threadIdx.x] = r_intderiv[7];
BL0[(size*blockDim.x*blockIdx.x) + (45 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (46 * blockDim.x) + threadIdx.x] = r_intderiv[11];
BL0[(size*blockDim.x*blockIdx.x) + (47 * blockDim.x) + threadIdx.x] = r_intderiv[10];
BL0[(size*blockDim.x*blockIdx.x) + (48 * blockDim.x) + threadIdx.x] = r_intderiv[2];
BL0[(size*blockDim.x*blockIdx.x) + (49 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (50 * blockDim.x) + threadIdx.x] = r_intderiv[0];
BL0[(size*blockDim.x*blockIdx.x) + (51 * blockDim.x) + threadIdx.x] = r_intderiv[5];
BL0[(size*blockDim.x*blockIdx.x) + (52 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (53 * blockDim.x) + threadIdx.x] = r_intderiv[3];
BL0[(size*blockDim.x*blockIdx.x) + (54 * blockDim.x) + threadIdx.x] = r_intderiv[8];
BL0[(size*blockDim.x*blockIdx.x) + (55 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (56 * blockDim.x) + threadIdx.x] = r_intderiv[6];
BL0[(size*blockDim.x*blockIdx.x) + (57 * blockDim.x) + threadIdx.x] = r_intderiv[11];
BL0[(size*blockDim.x*blockIdx.x) + (58 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (59 * blockDim.x) + threadIdx.x] = r_intderiv[9];
BL0[(size*blockDim.x*blockIdx.x) + (60 * blockDim.x) + threadIdx.x] = r_intderiv[1];
BL0[(size*blockDim.x*blockIdx.x) + (61 * blockDim.x) + threadIdx.x] = r_intderiv[0];
BL0[(size*blockDim.x*blockIdx.x) + (62 * blockDim.x) + threadIdx.x] = 0.0f;
BL0[(size*blockDim.x*blockIdx.x) + (63 * blockDim.x) + threadIdx.x] = r_intderiv[4];
BL0[(size*blockDim.x*blockIdx.x) + (64 * blockDim.x) + threadIdx.x] = r_intderiv[3];
BL0[(size*blockDim.x*blockIdx.x) + (65 * blockDim.x) + threadIdx.x] = 0.0f;
```

```
                BL0[(size*blockDim.x*blockIdx.x) + (66 * blockDim.x) + threadIdx.x] = r_intderiv[7];
                BL0[(size*blockDim.x*blockIdx.x) + (67 * blockDim.x) + threadIdx.x] = r_intderiv[6];
                BL0[(size*blockDim.x*blockIdx.x) + (68 * blockDim.x) + threadIdx.x] = 0.0f;
                BL0[(size*blockDim.x*blockIdx.x) + (69 * blockDim.x) + threadIdx.x] = r_intderiv[10];
                BL0[(size*blockDim.x*blockIdx.x) + (70 * blockDim.x) + threadIdx.x] = r_intderiv[9];
                BL0[(size*blockDim.x*blockIdx.x) + (71 * blockDim.x) + threadIdx.x] = 0.0f;

                // Fill BNL matrix from interpolation derivatives (holds values used in both IntDeriv and BNL
calculations)
                for (int i = 0; i < tet_element_dof; i++) {
                        IntDeriv[(tet_element_dof*blockDim.x*blockIdx.x) + (i*blockDim.x) + threadIdx.x] =
r_intderiv[i];
                }
        }
        //for (int ind = blockIdx.x*blockDim.x + threadIdx.x; (ind >= (solid_element_count+shell_element_count)) &&
(ind < (solid_element_count + lig_count + shell_element_count)); ind += blockDim.x*gridDim.x){
        else if (type_ind[ind] == 25) {
                vb_node = element_nodes[(ind*nodes_per_tet)];
                ground_node = element_nodes[(ind*nodes_per_tet) + 1];

                x_diff = nodes[ground_node*global_dof] - nodes[vb_node*global_dof];
                y_diff = nodes[ground_node*global_dof + 1] - nodes[vb_node*global_dof + 1];
                z_diff = nodes[ground_node*global_dof + 2] - nodes[vb_node*global_dof + 2];

                lig_lengths0[ind] = sqrtf((x_diff*x_diff) + (y_diff*y_diff) + (z_diff*z_diff));
        }
}
__global__ void FibLengths0(float* fib_lengths_0, float* global_coords, int* fib_ind, int* body_ind, int
total_element_count) {
        int ind = blockIdx.x*blockDim.x + threadIdx.x;
        register float fib_x;
        register float fib_y;
        register float fib_z;
        register int node1;
        register int node2;

        if (ind < total_element_count) {

                if (body_ind[ind] == 3) {
                        // Calculate fiber lengths squared for strain calculations in other functions
                        node1 = fib_ind[ind*nodes_per_fib];
                        node2 = fib_ind[ind*nodes_per_fib + 1];
                        fib_x = global_coords[node2*global_dof] - global_coords[node1*global_dof];
                        fib_y = global_coords[node2*global_dof + 1] - global_coords[node1*global_dof + 1];
                        fib_z = global_coords[node2*global_dof + 2] - global_coords[node1*global_dof + 2];
                        fib_lengths_0[ind] = (fib_x*fib_x) + (fib_y*fib_y) + (fib_z*fib_z);
                }
        }
}
__global__ void SolidStiffness(float* K_matrix, float* BL0, float* IntDeriv, float* volumes, float* node_loc, float*
u_global_vector, float* u_vector, float* u_increment, float* b_vector,
        float* b_global, int* BC_nodes, int* d_element_ind, int* type_ind, int* dof_ind, int* body_ind, int* sec_ind, float
BC_value, float* lig_lengths0, int* fib_nodes, float* fib_length0,
        float* local_coords, float* local_coords0, int total_node_count) {
        const int Bsize = 72;
        const int CBsize = 78;
        const int Csize = 21;
        const int C_cols = 6;
        const int strain_size = 6;
        const int Ssize = 6;
        // Material constants
```

```
	const float AF_1 = 0.56f;// 0.18f; 0.56f
	const float AF_2 = 0.14f;// 0.045f; 0.14f
	const float AF_k = 13.9997f; //4.35f; d=0.14286 -> 13.9997
	const float Ex_vert = 140.0f;
	const float Ey_vert = 200.0f;
	const float Ez_vert = 140.0f;
	const float Gxy_vert = 48.3f;
	const float Gzy_vert = 48.3f;
	const float Gzx_vert = 48.3f;
	const float pryz_vert = 0.315f;
	const float prxz_vert = 0.45f;
	const float prxy_vert = 0.315f;
	const float Ex_cort = 11300.0f; //11300
	const float Ey_cort = 22000.0f;//11300
	const float Ez_cort = 11300.0f;//22000
	const float Gxy_cort = 5400.0f;//3800
	const float Gzy_cort = 5400.0f;//5400
	const float Gzx_cort = 3800.0f;//5400
	const float pryz_cort = 0.203f;//0.203
	const float prxz_cort = 0.484f;//0.203
	const float prxy_cort = 0.203f;//0.484
	const float E_post = 3500.0f; // change back to 3500
	const float nu_post = 0.25f; //change back to 0.25
	const float E_NP = 1.0f; // 1
	const float nu_NP = 0.49958f; // 0.49999f
	const float E_end = 23.8f; // 23.8f
	const float nu_end = 0.4f; // 0.4f
	__shared__ float CB[CBsize*BLOCK_SIZE]; // 78 for MET elements (s_u_vector has the other 12 spaces
necessary)
	__shared__ float s_u_vector[tet_element_dof*BLOCK_SIZE]; // also holds the invar3 calcs necessary for
Mooney-Rivlin and the shell_strain (6 spaces)
	__shared__ float AF_l[global_dof*BLOCK_SIZE];
	__shared__ float AF_V_M[BLOCK_SIZE];
	//all above are __shared__

	//for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < solid_element_count; ind += blockDim.x *
gridDim.x){
	int ind = blockDim.x*blockIdx.x + threadIdx.x;
	if ((type_ind[ind] == 24)) {
		// register memory for thread
		register int node_ind = 0;
		register float del = 0.0f; // also used for J3 Mooney-Rivlin calc
		register float sum = 0.0f;
		register float fib_force;
		register float l_vector[3][3]; // also holds the invar_deriv calcs for Mooney_Rivlin
		register float BL[Bsize];
		register float CMatrix[Csize];
		register float r_IntDeriv[tet_element_dof];
		register float r_strain[strain_size]; // also holds Cauchy-Green deformation components
		register float AF_E;//Also used for modified element pressure and modified Jacob
		register float AF_V;//Also used for original jacob
		register int kind = 0;

		// Move global memory to register/shared memory
		{
			for (int k = 0; k < tet_element_dof; k++) {
				s_u_vector[(k*blockDim.x) + threadIdx.x] =
u_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (k*blockDim.x) + threadIdx.x];
				r_IntDeriv[k] = IntDeriv[(tet_element_dof*blockDim.x*blockIdx.x) +
(k*blockDim.x) + threadIdx.x];
			}
```

```
		}
		// Calculate l_vector for BL
		{
			for (int i = 0; i < global_dof; i++) {
				for (int j = 0; j < global_dof; j++) {
					l_vector[i][j] = 0.0f;
					for (int k = 0; k < nodes_per_tet; k++) {
						l_vector[i][j] += (r_IntDeriv[global_dof*k + j] *
s_u_vector[((global_dof*k + i)*blockDim.x) + threadIdx.x]);
					}
				}
			}
		}
		// Calculate BL
		for (int k = 0; k < global_dof; k++) {
			for (int i = 0; i < nodes_per_tet; i++) {
				for (int j = 0; j < global_dof; j++) {
					BL[k*tet_element_dof + i*global_dof + j] = ((r_IntDeriv[i*global_dof +
k] * l_vector[j][k]) +
						BL0[(Bsize*blockDim.x*blockIdx.x) + ((k*tet_element_dof +
i*global_dof + j)*blockDim.x) + threadIdx.x]);
				}
			}
		}
		for (int i = 0; i < nodes_per_tet; i++) {
			for (int j = 0; j < global_dof; j++) {
				BL[36 + i*global_dof + j] = ((r_IntDeriv[i*global_dof + 2] * l_vector[j][1]) +
(r_IntDeriv[i*global_dof + 1] * l_vector[j][2]) +
					BL0[(Bsize*blockDim.x*blockIdx.x) + ((36 + i*global_dof +
j)*blockDim.x) + threadIdx.x]);
			}
		}
		for (int i = 0; i < nodes_per_tet; i++) {
			for (int j = 0; j < global_dof; j++) {
				BL[48 + i*global_dof + j] = ((r_IntDeriv[i*global_dof + 2] * l_vector[j][0]) +
(r_IntDeriv[i*global_dof] * l_vector[j][2]) +
					BL0[(Bsize*blockDim.x*blockIdx.x) + ((48 + i*global_dof +
j)*blockDim.x) + threadIdx.x]);
			}
		}
		for (int i = 0; i < nodes_per_tet; i++) {
			for (int j = 0; j < global_dof; j++) {
				BL[60 + i*global_dof + j] = ((r_IntDeriv[i*global_dof + 1] * l_vector[j][0]) +
(r_IntDeriv[i*global_dof] * l_vector[j][1]) +
					BL0[(Bsize*blockDim.x*blockIdx.x) + ((60 + i*global_dof +
j)*blockDim.x) + threadIdx.x]);
			}
		}
		// Compute strain from BL and u
		r_strain[0] = l_vector[0][0];
		r_strain[1] = l_vector[1][1];
		r_strain[2] = l_vector[2][2];
		r_strain[3] = 0.5f*(l_vector[1][2] + l_vector[2][1]);
		r_strain[4] = 0.5f*(l_vector[0][2] + l_vector[2][0]);
		r_strain[5] = 0.5f*(l_vector[0][1] + l_vector[1][0]);
		for (int k = 0; k < global_dof; k++) {
			r_strain[0] += 0.5f*(l_vector[k][0] * l_vector[k][0]);
			r_strain[1] += 0.5f*(l_vector[k][1] * l_vector[k][1]);
			r_strain[2] += 0.5f*(l_vector[k][2] * l_vector[k][2]);
			r_strain[3] += 0.5f*(l_vector[k][1] * l_vector[k][2]);
			r_strain[4] += 0.5f*(l_vector[k][0] * l_vector[k][2]);
```

```
                r_strain[5] += 0.5f*(l_vector[k][0] * l_vector[k][1]);
        }

        //Calculate KL by defining materials based on each body and multiplying by BL.  Select threads based on which body the elements belong to
        switch (body_ind[ind])
        {
        case(1):
                // CMatrix is symmetric, so only half of the matrix is stored
                del = 1.0f - ((prxy_vert*prxy_vert*Ey_vert) / Ex_vert) - ((pryz_vert*pryz_vert*Ez_vert) / Ey_vert) - ((prxz_vert*prxz_vert*Ez_vert) / Ex_vert) - ((2.0f*prxy_vert*pryz_vert*prxz_vert*Ez_vert) / Ex_vert);
                CMatrix[0] = (Ey_vert - (pryz_vert*pryz_vert*Ez_vert))*(Ex_vert / (Ey_vert*del));
                CMatrix[1] = ((Ey_vert*prxy_vert) + (prxz_vert*pryz_vert*Ez_vert)) / del;
                CMatrix[2] = (prxz_vert + (pryz_vert*prxy_vert))*(Ez_vert / del);
                CMatrix[3] = 0;
                CMatrix[4] = 0;
                CMatrix[5] = 0;
                CMatrix[6] = (Ex_vert - (prxz_vert*prxz_vert*Ez_vert))*(Ey_vert / (Ex_vert*del));
                CMatrix[7] = ((Ex_vert*pryz_vert) + (prxz_vert*prxy_vert*Ey_vert))*(Ez_vert / (Ex_vert*del));
                CMatrix[8] = 0;
                CMatrix[9] = 0;
                CMatrix[10] = 0;
                CMatrix[11] = (Ex_vert - (prxy_vert*prxy_vert*Ey_vert))*(Ez_vert / (Ex_vert*del));
                CMatrix[12] = 0;
                CMatrix[13] = 0;
                CMatrix[14] = 0;
                CMatrix[15] = Gzy_vert;
                CMatrix[16] = 0;
                CMatrix[17] = 0;
                CMatrix[18] = Gzx_vert;
                CMatrix[19] = 0;
                CMatrix[20] = Gxy_vert;
                //Adjust strain for shear strain
                r_strain[3] *= 2.0f;
                r_strain[4] *= 2.0f;
                r_strain[5] *= 2.0f;
                {
                        for (int i = 0; i < Ssize; i++) {
                                sum = 0.0f;
                                // below Cmatrix diagonal (columns)
                                kind = i;
                                for (int k = 0; k < i; k++) {
                                        sum += CMatrix[kind] * r_strain[k];
                                        kind += C_cols - k - 1;
                                }
                                // above CMatrix diagonal (rows)
                                kind = i*C_cols - i*(i - 1) / 2;
                                for (int k = i; k < C_cols; k++) {
                                        sum += CMatrix[kind] * r_strain[k];
                                        kind++;
                                }
                                s_u_vector[i*blockDim.x + threadIdx.x] = sum;
                        }
                }
                break;
        case(26):
                // CMatrix is symmetric, so only half of the matrix is stored
                del = 1.0f - ((prxy_cort*prxy_cort*Ey_cort) / Ex_cort) - ((pryz_cort*pryz_cort*Ez_cort) / Ey_cort) - ((prxz_cort*prxz_cort*Ez_cort) / Ex_cort) - ((2.0f*prxy_cort*pryz_cort*prxz_cort*Ez_cort) / Ex_cort);
                CMatrix[0] = (Ey_cort - (pryz_cort*pryz_cort*Ez_cort))*(Ex_cort / (Ey_cort*del));
```

```
                        CMatrix[1] = ((Ey_cort*prxy_cort) + (prxz_cort*pryz_cort*Ez_cort)) / del;
                        CMatrix[2] = (prxz_cort + (pryz_cort*prxy_cort))*(Ez_cort / del);
                        CMatrix[3] = 0;
                        CMatrix[4] = 0;
                        CMatrix[5] = 0;
                        CMatrix[6] = (Ex_cort - (prxz_cort*prxz_cort*Ez_cort))*(Ey_cort / (Ex_cort*del));
                        CMatrix[7] = ((Ex_cort*pryz_cort) + (prxz_cort*prxy_cort*Ey_cort))*(Ez_cort /
(Ex_cort*del));
                        CMatrix[8] = 0;
                        CMatrix[9] = 0;
                        CMatrix[10] = 0;
                        CMatrix[11] = (Ex_cort - (prxy_cort*prxy_cort*Ey_cort))*(Ez_cort / (Ex_cort*del));
                        CMatrix[12] = 0;
                        CMatrix[13] = 0;
                        CMatrix[14] = 0;
                        CMatrix[15] = Gzy_cort;
                        CMatrix[16] = 0;
                        CMatrix[17] = 0;
                        CMatrix[18] = Gzx_cort;
                        CMatrix[19] = 0;
                        CMatrix[20] = Gxy_cort;

                        //Adjust strain for shear strain
                        r_strain[3] *= 2.0f;
                        r_strain[4] *= 2.0f;
                        r_strain[5] *= 2.0f;
                        {
                                for (int i = 0; i < Ssize; i++) {
                                        sum = 0.0f;
                                        // below Cmatrix diagonal (columns)
                                        kind = i;
                                        for (int k = 0; k < i; k++) {
                                                sum += CMatrix[kind] * r_strain[k];
                                                kind += C_cols - k - 1;
                                        }
                                        // above CMatrix diagonal (rows)
                                        kind = i*C_cols - i*(i - 1) / 2;
                                        for (int k = i; k < C_cols; k++) {
                                                sum += CMatrix[kind] * r_strain[k];
                                                kind++;
                                        }
                                        s_u_vector[i*blockDim.x + threadIdx.x] = sum;
                                }
                        }

                        break;
                case(2):

                        // CMatrix is symmetric, so only half of the matrix is stored
                        del = E_NP / ((1.0f + nu_NP)*(1.0f - (2.0f*nu_NP)));
                        CMatrix[0] = del*(1.0f - nu_NP);
                        CMatrix[1] = del*nu_NP;
                        CMatrix[2] = del*nu_NP;
                        CMatrix[3] = 0;
                        CMatrix[4] = 0;
                        CMatrix[5] = 0;
                        CMatrix[6] = del*(1.0f - nu_NP);
                        CMatrix[7] = del*nu_NP;
                        CMatrix[8] = 0;
                        CMatrix[9] = 0;
                        CMatrix[10] = 0;
```

```
CMatrix[11] = del*(1.0f - nu_NP);
CMatrix[12] = 0;
CMatrix[13] = 0;
CMatrix[14] = 0;
CMatrix[15] = del*(0.5f - nu_NP);
CMatrix[16] = 0;
CMatrix[17] = 0;
CMatrix[18] = del*(0.5f - nu_NP);
CMatrix[19] = 0;
CMatrix[20] = del*(0.5f - nu_NP);
//Adjust strain for shear strain
r_strain[3] *= 2.0f;
r_strain[4] *= 2.0f;
r_strain[5] *= 2.0f;
{
	for (int i = 0; i < Ssize; i++) {
		sum = 0.0f;
		// below Cmatrix diagonal (columns)
		kind = i;
		for (int k = 0; k < i; k++) {
			sum += CMatrix[kind] * r_strain[k];
			kind += C_cols - k - 1;
		}
		// above CMatrix diagonal (rows)
		kind = i*C_cols - i*(i - 1) / 2;
		for (int k = i; k < C_cols; k++) {
			sum += CMatrix[kind] * r_strain[k];
			kind++;
		}
		s_u_vector[i*blockDim.x + threadIdx.x] = sum;
	}
}

break;
case(3):
// Annulus Fibrosus
//********************NOTE THAT THE NONLINEAR PORTION OF THE FIBRE STIFFNESS (AS PER TL BAR FORMULATION) HAS NOT YET BEEN ADDED TO THE STIFFNESS MATRIX FOR THIS ELEMENT*******************
// Calculate C_matrix as a Mooney-Rivlin material with elastic fibres added to it (iso-strain)
// Update fiber direction
// Update the fib_dir
kind = fib_nodes[ind*nodes_per_fib]; //kind is 0
node_ind = fib_nodes[ind*nodes_per_fib + 1]; //node_ind is 1
AF_l[threadIdx.x + (0 * blockDim.x)] = node_loc[node_ind * global_dof] - node_loc[kind * global_dof];
AF_l[threadIdx.x + (1 * blockDim.x)] = node_loc[node_ind*global_dof + 1] - node_loc[kind*global_dof + 1];
AF_l[threadIdx.x + (2 * blockDim.x)] = node_loc[node_ind*global_dof + 2] - node_loc[kind*global_dof + 2];
l_vector[0][0] = (AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 * blockDim.x)]) + (AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)])
	+ (AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]); // fib_length
l_vector[0][1] = fib_length0[ind]; // fib_length0
sum = (l_vector[0][0] - l_vector[0][1]) / (2.0f*l_vector[0][1]);
l_vector[0][0] = sqrtf(l_vector[0][1]); // was l_vector[0][0]
AF_l[threadIdx.x + (0 * blockDim.x)] = AF_l[threadIdx.x + (0 * blockDim.x)] / l_vector[0][0];
AF_l[threadIdx.x + (1 * blockDim.x)] = AF_l[threadIdx.x + (1 * blockDim.x)] / l_vector[0][0];
```

```
                        AF_l[threadIdx.x + (2 * blockDim.x)] = AF_l[threadIdx.x + (2 * blockDim.x)] /
l_vector[0][0];
                        // Determine stiffness (and volume proportion) based on annulus layer and strain range
                        switch (sec_ind[ind]) {
                        case(1):
                                AF_V = 0.23f;
                                del = 1.0f; //Not actually the stiffness but scale factor multiplied to stifness below
                                break;
                        case(2):
                                AF_V = 0.17f;
                                del = 0.9f;
                                break;
                        case(3):
                                AF_V = 0.11f;
                                del = 0.75f;
                                break;
                        case(4):
                                AF_V = 0.05f;
                                del = 0.65;
                                break;
                        default:
                                break;
                        }
                        if ((sum >= 0.0f) && (sum < 0.02f)) {
                                AF_E = 600.0f*del;
                                fib_force = AF_E*sum;
                        }
                        else if ((sum >= 0.02f) && (sum < 0.06f)) {
                                AF_E = 1061.1f*del;
                                fib_force = AF_E*(sum - 0.02f) + (del*12.0f);
                        }
                        else if ((sum >= 0.06f) && (sum < 0.11f)) {
                                AF_E = 455.56f*del;
                                fib_force = AF_E*(sum - 0.06f) + (del*54.444f);
                        }
                        else if (sum >= 0.11f) {
                                AF_E = 188.143f*del;
                                fib_force = AF_E*(sum - 0.11f) + (del*77.222f);
                        }
                        else {
                                AF_E = 0.0f;
                                fib_force = 0.0f;
                        }
                        AF_E *= AF_V;
                        AF_V_M[threadIdx.x] = 1.0f - AF_V;
                        // Precalculations for the C_matrix
                        // Calculate components of Cauchy-Green matrix using strain array (write back regular strain
later)
                        CB[0 * blockDim.x + threadIdx.x] = 2.0f*r_strain[0] + 1.0f;
                        CB[1 * blockDim.x + threadIdx.x] = 2.0f*r_strain[1] + 1.0f;
                        CB[2 * blockDim.x + threadIdx.x] = 2.0f*r_strain[2] + 1.0f;
                        CB[3 * blockDim.x + threadIdx.x] = 2.0f*r_strain[3];
                        CB[4 * blockDim.x + threadIdx.x] = 2.0f*r_strain[4];
                        CB[5 * blockDim.x + threadIdx.x] = 2.0f*r_strain[5];
                        // Calculate invariants
                        l_vector[0][0] = CB[0 * blockDim.x + threadIdx.x] + CB[1 * blockDim.x + threadIdx.x] +
CB[2 * blockDim.x + threadIdx.x];
                        l_vector[0][1] = (CB[0 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x + threadIdx.x]) +
(CB[0 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x + threadIdx.x])
                                + (CB[1 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x + threadIdx.x]) - (CB[3
* blockDim.x + threadIdx.x] * CB[3 * blockDim.x + threadIdx.x])
```

```
                                    - (CB[4 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x + threadIdx.x]) - (CB[5
* blockDim.x + threadIdx.x] * CB[5 * blockDim.x + threadIdx.x]);
                                l_vector[0][2] = (CB[0 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x + threadIdx.x] *
CB[2 * blockDim.x + threadIdx.x])
                                    + (2.0f*CB[3 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x + threadIdx.x] *
CB[5 * blockDim.x + threadIdx.x])
                                    - (CB[0 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x + threadIdx.x] * CB[3 *
blockDim.x + threadIdx.x])
                                    - (CB[1 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x + threadIdx.x] * CB[4 *
blockDim.x + threadIdx.x])
                                    - (CB[2 * blockDim.x + threadIdx.x] * CB[5 * blockDim.x + threadIdx.x] * CB[5 *
blockDim.x + threadIdx.x]);
                                // Calculate derivative of third invariant
                                l_vector[1][0] = 2.0f*((CB[1 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x +
threadIdx.x]) - (CB[3 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x + threadIdx.x]));
                                l_vector[1][1] = 2.0f*((CB[0 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x +
threadIdx.x]) - (CB[4 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x + threadIdx.x]));
                                l_vector[1][2] = 2.0f*((CB[0 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x +
threadIdx.x]) - (CB[5 * blockDim.x + threadIdx.x] * CB[5 * blockDim.x + threadIdx.x]));
                                l_vector[2][0] = 2.0f*((CB[5 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x +
threadIdx.x]) - (CB[3 * blockDim.x + threadIdx.x] * CB[0 * blockDim.x + threadIdx.x]));
                                l_vector[2][1] = 2.0f*((CB[5 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x +
threadIdx.x]) - (CB[4 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x + threadIdx.x]));
                                l_vector[2][2] = 2.0f*((CB[4 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x +
threadIdx.x]) - (CB[5 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x + threadIdx.x]));
                                // Calculate third invariant of deformation gradient
                                del = sqrtf(l_vector[0][2]);
                                // Calculate third invariant roots
                                s_u_vector[0 * blockDim.x + threadIdx.x] = 1.0f / (2.0f*del);
                                s_u_vector[1 * blockDim.x + threadIdx.x] = 1.0f / cbrtf(l_vector[0][2]);
                                s_u_vector[2 * blockDim.x + threadIdx.x] = s_u_vector[1 * blockDim.x + threadIdx.x] *
s_u_vector[1 * blockDim.x + threadIdx.x];
                                s_u_vector[3 * blockDim.x + threadIdx.x] = s_u_vector[2 * blockDim.x + threadIdx.x] *
s_u_vector[2 * blockDim.x + threadIdx.x];
                                s_u_vector[4 * blockDim.x + threadIdx.x] = s_u_vector[3 * blockDim.x + threadIdx.x] *
s_u_vector[1 * blockDim.x + threadIdx.x];
                                s_u_vector[5 * blockDim.x + threadIdx.x] = s_u_vector[4 * blockDim.x + threadIdx.x] *
s_u_vector[2 * blockDim.x + threadIdx.x];
                                s_u_vector[6 * blockDim.x + threadIdx.x] = s_u_vector[3 * blockDim.x + threadIdx.x] *
s_u_vector[3 * blockDim.x + threadIdx.x];
                                s_u_vector[7 * blockDim.x + threadIdx.x] = 8.0f*s_u_vector[0 * blockDim.x + threadIdx.x] *
s_u_vector[0 * blockDim.x + threadIdx.x] * s_u_vector[0 * blockDim.x + threadIdx.x];
                                // Adjust third invariant roots to include common calculations for C_Matrix
                                s_u_vector[8 * blockDim.x + threadIdx.x] = s_u_vector[0 * blockDim.x + threadIdx.x] *
s_u_vector[0 * blockDim.x + threadIdx.x];
                                s_u_vector[0 * blockDim.x + threadIdx.x] *= 2.0f;
                                s_u_vector[1 * blockDim.x + threadIdx.x] *= 2.0f;
                                s_u_vector[2 * blockDim.x + threadIdx.x] *= 2.0f;
                                s_u_vector[3 * blockDim.x + threadIdx.x] *= -0.6666666666667f;
                                s_u_vector[4 * blockDim.x + threadIdx.x] *= -1.3333333333f;
                                s_u_vector[5 * blockDim.x + threadIdx.x] *= 0.444444444f*l_vector[0][0];
                                s_u_vector[6 * blockDim.x + threadIdx.x] *= 1.11111111111f*l_vector[0][1];
                                s_u_vector[7 * blockDim.x + threadIdx.x] *= -0.25f;
                                del -= 1.0f;
                                // Fill components of C_Matrix **************************Adjust for AF_V_M
                                CMatrix[0] = (AF_V_M[threadIdx.x] * ((AF_1*((2.0f*s_u_vector[3 * blockDim.x +
threadIdx.x] * l_vector[1][0]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[1][0])))
                                    + (AF_2*((2.0f*s_u_vector[4 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[0
* blockDim.x + threadIdx.x])*l_vector[1][0]) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][0])))
```

```
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][0]) + (del*s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[1][0])))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 * blockDim.x)]);
                                CMatrix[1] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
(l_vector[1][0] + l_vector[1][1] + (2.0f*l_vector[0][0] * CB[2 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x
+ threadIdx.x] * l_vector[1][0] * l_vector[1][1])))
                                        + (AF_2*((2.0f*s_u_vector[2 * blockDim.x + threadIdx.x]) + (s_u_vector[4 *
blockDim.x + threadIdx.x] * (((l_vector[0][0] - CB[0 * blockDim.x + threadIdx.x])*l_vector[1][1])
                                                + ((l_vector[0][0] - CB[1 * blockDim.x + threadIdx.x])*l_vector[1][0]) +
(2.0f*l_vector[0][1] * CB[2 * blockDim.x + threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][1])))
                                        + (AF_k*((del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][1]) + (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x + threadIdx.x])))
                                                + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][1]))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)]);
                                CMatrix[2] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
(l_vector[1][0] + l_vector[1][2] + (2.0f*l_vector[0][0] * CB[1 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x
+ threadIdx.x] * l_vector[1][0] * l_vector[1][2])))
                                        + (AF_2*((2.0f*s_u_vector[2 * blockDim.x + threadIdx.x]) + (s_u_vector[4 *
blockDim.x + threadIdx.x] * (((l_vector[0][0] - CB[0 * blockDim.x + threadIdx.x])*l_vector[1][2])
                                                + ((l_vector[0][0] - CB[2 * blockDim.x + threadIdx.x])*l_vector[1][0]) +
(2.0f*l_vector[0][1] * CB[1 * blockDim.x + threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][2])))
                                        + (AF_k*((del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][2]) + (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x + threadIdx.x])))
                                                + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[1][2]))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[3] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
(l_vector[2][0] - (2.0f*l_vector[0][0] * CB[3 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x + threadIdx.x] *
l_vector[1][0] * l_vector[2][0])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][0] *
(l_vector[0][0] - CB[0 * blockDim.x + threadIdx.x])) - (CB[3 * blockDim.x + threadIdx.x] * l_vector[1][0]) -
(2.0f*l_vector[0][1] * CB[3 * blockDim.x + threadIdx.x])))
                                                + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[2][0])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[2][0]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[2][0]) -
                                        (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[4] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][1]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[2][1])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][1] *
(l_vector[0][0] - CB[0 * blockDim.x + threadIdx.x])) - (CB[4 * blockDim.x + threadIdx.x] * l_vector[1][0])))
                                                + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[2][1])))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][0] *
l_vector[2][1]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[2][1])))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[5] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][2]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[2][2])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][2] *
(l_vector[0][0] - CB[0 * blockDim.x + threadIdx.x])) - (CB[5 * blockDim.x + threadIdx.x] * l_vector[1][0])))
```

```
                                        + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][0] *
l_vector[2][2]))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][0] *
l_vector[2][2]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][0] * l_vector[2][2])))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)]);
                                CMatrix[6] = (AF_V_M[threadIdx.x] * ((AF_1*((2.0f*s_u_vector[3 * blockDim.x +
threadIdx.x] * l_vector[1][1]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[1][1])))
                                        + (AF_2*((2.0f*s_u_vector[4 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[1
* blockDim.x + threadIdx.x])*l_vector[1][1]) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[1][1])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[1][1]) + (del*s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[1][1])))))
                                        + (AF_E*AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)]);
                                CMatrix[7] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
(l_vector[1][1] + l_vector[1][2] + (2.0f*l_vector[0][0] * CB[0 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x
+ threadIdx.x] * l_vector[1][1] * l_vector[1][2])))
                                        + (AF_2*((2.0f*s_u_vector[2 * blockDim.x + threadIdx.x]) + (s_u_vector[4 *
blockDim.x + threadIdx.x] * (((l_vector[0][0] - CB[1 * blockDim.x + threadIdx.x])*l_vector[1][2])
                                        + ((l_vector[0][0] - CB[2 * blockDim.x + threadIdx.x])*l_vector[1][1]) +
(2.0f*l_vector[0][1] * CB[0 * blockDim.x + threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[1][2])))
                                        + (AF_k*((del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[1][2]) + (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[0 * blockDim.x + threadIdx.x])))
                                        + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[1][2]))))
                                        + (AF_E*AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[8] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][0]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[2][0])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][0] *
(l_vector[0][0] - CB[1 * blockDim.x + threadIdx.x])) - (CB[3 * blockDim.x + threadIdx.x] * l_vector[1][1])))
                                        + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[2][0])))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][1] *
l_vector[2][0]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[2][0])))))
                                        + (AF_E*AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[9] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x] *
(l_vector[2][1] - (2.0f*l_vector[0][0] * CB[4 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x + threadIdx.x] *
l_vector[1][1] * l_vector[2][1])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][1] *
(l_vector[0][0] - CB[1 * blockDim.x + threadIdx.x])) - (CB[4 * blockDim.x + threadIdx.x] * l_vector[1][1]) -
(2.0f*l_vector[0][1] * CB[4 * blockDim.x + threadIdx.x])))
                                        + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[2][1])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[2][1]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[2][1]) -
                                        (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[10] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[2][2]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[2][2])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][2] *
(l_vector[0][0] - CB[1 * blockDim.x + threadIdx.x])) - (CB[5 * blockDim.x + threadIdx.x] * l_vector[1][1])))
                                        + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][1] *
l_vector[2][2])))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][1] *
l_vector[2][2]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][1] * l_vector[2][2])))))
```

```
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)]);
                                CMatrix[11] = (AF_V_M[threadIdx.x] * ((AF_1*((2.0f*s_u_vector[3 * blockDim.x +
threadIdx.x] * l_vector[1][2]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[1][2])))
                                        + (AF_2*((2.0f*s_u_vector[4 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[2
* blockDim.x + threadIdx.x])*l_vector[1][2]) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[1][2])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[1][2]) + (del*s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[1][2])))))
                                        + (AF_E*AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[12] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[2][0]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[2][0])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][0] *
(l_vector[0][0] - CB[2 * blockDim.x + threadIdx.x])) - (CB[3 * blockDim.x + threadIdx.x] * l_vector[1][2])))
                                                + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[2][0])))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][2] *
l_vector[2][0]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[2][0])))))
                                        + (AF_E*AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[13] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[2][1]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[2][1])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][1] *
(l_vector[0][0] - CB[2 * blockDim.x + threadIdx.x])) - (CB[4 * blockDim.x + threadIdx.x] * l_vector[1][2])))
                                                + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[2][1])))
                                        + (AF_k*((s_u_vector[7 * blockDim.x + threadIdx.x] * del*l_vector[1][2] *
l_vector[2][1]) + (s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[2][1])))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (2 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[14] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* (l_vector[2][2] - (2.0f*l_vector[0][0] * CB[5 * blockDim.x + threadIdx.x]))) + (s_u_vector[5 * blockDim.x + threadIdx.x]
* l_vector[1][2] * l_vector[2][2])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((l_vector[2][2] *
(l_vector[0][0] - CB[2 * blockDim.x + threadIdx.x])) - (CB[5 * blockDim.x + threadIdx.x] * l_vector[1][2]) -
(2.0f*l_vector[0][1] * CB[5 * blockDim.x + threadIdx.x])))
                                                + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[2][2])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[1][2] *
l_vector[2][2]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[1][2] * l_vector[2][2]) -
                                        (2.0f*s_u_vector[0 * blockDim.x + threadIdx.x] * CB[5 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[15] = (AF_V_M[threadIdx.x] * ((AF_1*((-s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[0 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][0] *
l_vector[2][0])))
                                        + (AF_2*((-s_u_vector[4 * blockDim.x + threadIdx.x] * ((2.0f*CB[3 * blockDim.x
+ threadIdx.x] * l_vector[2][0]) + (l_vector[0][1] * CB[0 * blockDim.x + threadIdx.x]))) +
                                        (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][0] * l_vector[2][0]) -
s_u_vector[2 * blockDim.x + threadIdx.x]))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][0] *
l_vector[2][0]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][0] * l_vector[2][0]) -
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[0 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[16] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[5 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][1] *
l_vector[2][0])))
```

```
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((-CB[4 * blockDim.x +
threadIdx.x] * l_vector[2][0]) - (CB[3 * blockDim.x + threadIdx.x] * l_vector[2][1]) + (l_vector[0][1] * CB[5 * blockDim.x
+ threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][1] * l_vector[2][0])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][1] *
l_vector[2][0]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][1] * l_vector[2][0]) +
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[5 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[17] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[4 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][0])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((-CB[5 * blockDim.x +
threadIdx.x] * l_vector[2][0]) - (CB[3 * blockDim.x + threadIdx.x] * l_vector[2][2]) + (l_vector[0][1] * CB[4 * blockDim.x
+ threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][0])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][0]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][0]) +
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (1 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[18] = (AF_V_M[threadIdx.x] * ((AF_1*((-s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[1 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][1] *
l_vector[2][1])))
                                        + (AF_2*((-s_u_vector[4 * blockDim.x + threadIdx.x] * ((2.0f*CB[4 * blockDim.x
+ threadIdx.x] * l_vector[2][1]) + (l_vector[0][1] * CB[1 * blockDim.x + threadIdx.x]))) +
                                        (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][1] * l_vector[2][1]) -
s_u_vector[2 * blockDim.x + threadIdx.x]))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][1] *
l_vector[2][1]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][1] * l_vector[2][1]) -
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[1 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[19] = (AF_V_M[threadIdx.x] * ((AF_1*((s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[3 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][1])))
                                        + (AF_2*((s_u_vector[4 * blockDim.x + threadIdx.x] * ((-CB[5 * blockDim.x +
threadIdx.x] * l_vector[2][1]) - (CB[4 * blockDim.x + threadIdx.x] * l_vector[2][2]) + (l_vector[0][1] * CB[3 * blockDim.x
+ threadIdx.x]))) + (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][1])))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][1]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][1]) +
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (2 * blockDim.x)]);
                                CMatrix[20] = (AF_V_M[threadIdx.x] * ((AF_1*((-s_u_vector[3 * blockDim.x + threadIdx.x]
* l_vector[0][0] * CB[2 * blockDim.x + threadIdx.x]) + (s_u_vector[5 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][2])))
                                        + (AF_2*((-s_u_vector[4 * blockDim.x + threadIdx.x] * ((2.0f*CB[5 * blockDim.x
+ threadIdx.x] * l_vector[2][2]) + (l_vector[0][1] * CB[2 * blockDim.x + threadIdx.x]))) +
                                        (s_u_vector[6 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][2]) -
s_u_vector[2 * blockDim.x + threadIdx.x]))
                                        + (AF_k*((s_u_vector[8 * blockDim.x + threadIdx.x] * l_vector[2][2] *
l_vector[2][2]) + (del*((s_u_vector[7 * blockDim.x + threadIdx.x] * l_vector[2][2] * l_vector[2][2]) -
                                        (s_u_vector[0 * blockDim.x + threadIdx.x] * CB[2 * blockDim.x +
threadIdx.x])))))))
                                        + (AF_E*AF_l[threadIdx.x + (0 * blockDim.x)] * AF_l[threadIdx.x + (0 *
blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)] * AF_l[threadIdx.x + (1 * blockDim.x)]);
                                s_u_vector[3 * blockDim.x + threadIdx.x] = s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[0][0] * 0.5f;
```

```
                                s_u_vector[4 * blockDim.x + threadIdx.x] = s_u_vector[4 * blockDim.x + threadIdx.x] *
l_vector[0][1] * 0.5f;
                                del *= AF_k;
                                //r_strain is actually stress
                                r_strain[0] = (AF_V_M[threadIdx.x] * ((AF_1*(s_u_vector[1 * blockDim.x + threadIdx.x] +
(s_u_vector[3 * blockDim.x + threadIdx.x] * l_vector[1][0])))
                                        + (AF_2*((s_u_vector[2 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[0 *
blockDim.x + threadIdx.x])) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[1][0])) + (del*l_vector[1][0]))
                                        + (AF_V*(fib_force*AF_l[0 * blockDim.x + threadIdx.x] * AF_l[0 * blockDim.x +
threadIdx.x]));
                                r_strain[1] = (AF_V_M[threadIdx.x] * ((AF_1*(s_u_vector[1 * blockDim.x + threadIdx.x] +
(s_u_vector[3 * blockDim.x + threadIdx.x] * l_vector[1][1])))
                                        + (AF_2*((s_u_vector[2 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[1 *
blockDim.x + threadIdx.x])) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[1][1])) + (del*l_vector[1][0]))
                                        + (AF_V*(fib_force*AF_l[1 * blockDim.x + threadIdx.x] * AF_l[1 * blockDim.x +
threadIdx.x]));
                                r_strain[2] = (AF_V_M[threadIdx.x] * ((AF_1*(s_u_vector[1 * blockDim.x + threadIdx.x] +
(s_u_vector[3 * blockDim.x + threadIdx.x] * l_vector[1][2])))
                                        + (AF_2*((s_u_vector[2 * blockDim.x + threadIdx.x] * (l_vector[0][0] - CB[2 *
blockDim.x + threadIdx.x])) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[1][2])) + (del*l_vector[1][0]))
                                        + (AF_V*(fib_force*AF_l[2 * blockDim.x + threadIdx.x] * AF_l[2 * blockDim.x +
threadIdx.x]));
                                r_strain[3] = ((AF_V_M[threadIdx.x] * ((AF_1*s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][0])
                                        + (AF_2*((-s_u_vector[2 * blockDim.x + threadIdx.x] * CB[3 * blockDim.x +
threadIdx.x]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[2][0])) + (del*l_vector[2][0]))
                                        + (AF_V*(fib_force*AF_l[1 * blockDim.x + threadIdx.x] * AF_l[2 * blockDim.x +
threadIdx.x])));
                                r_strain[4] = ((AF_V_M[threadIdx.x] * ((AF_1*s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][1])
                                        + (AF_2*((-s_u_vector[2 * blockDim.x + threadIdx.x] * CB[4 * blockDim.x +
threadIdx.x]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[2][1])) + (del*l_vector[2][1]))
                                        + (AF_V*(fib_force*AF_l[0 * blockDim.x + threadIdx.x] * AF_l[2 * blockDim.x +
threadIdx.x])));
                                r_strain[5] = ((AF_V_M[threadIdx.x] * ((AF_1*s_u_vector[3 * blockDim.x + threadIdx.x] *
l_vector[2][2])
                                        + (AF_2*((-s_u_vector[2 * blockDim.x + threadIdx.x] * CB[5 * blockDim.x +
threadIdx.x]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * l_vector[2][2])) + (del*l_vector[2][2]))
                                        + (AF_V*(fib_force*AF_l[0 * blockDim.x + threadIdx.x] * AF_l[1 * blockDim.x +
threadIdx.x])));
                                for (int i = 0; i < Ssize; i++) s_u_vector[i*blockDim.x + threadIdx.x] = r_strain[i];

                                break;
                        case(13): // Posterior elements
                                        // CMatrix is symmetric, so only half of the matrix is stored
                                del = E_post / ((1.0f + nu_post)*(1.0f - (2.0f*nu_post)));
                                CMatrix[0] = del*(1.0f - nu_post);
                                CMatrix[1] = del*nu_post;
                                CMatrix[2] = del*nu_post;
                                CMatrix[3] = 0;
                                CMatrix[4] = 0;
                                CMatrix[5] = 0;
                                CMatrix[6] = del*(1.0f - nu_post);
                                CMatrix[7] = del*nu_post;
                                CMatrix[8] = 0;
                                CMatrix[9] = 0;
                                CMatrix[10] = 0;
                                CMatrix[11] = del*(1.0f - nu_post);
                                CMatrix[12] = 0;
                                CMatrix[13] = 0;
                                CMatrix[14] = 0;
```

```
		CMatrix[15] = del*(0.5f - nu_post);
		CMatrix[16] = 0;
		CMatrix[17] = 0;
		CMatrix[18] = del*(0.5f - nu_post);
		CMatrix[19] = 0;
		CMatrix[20] = del*(0.5f - nu_post);

		//Adjust strain for shear strain
		r_strain[3] *= 2.0f;
		r_strain[4] *= 2.0f;
		r_strain[5] *= 2.0f;
		{
			for (int i = 0; i < Ssize; i++) {
				sum = 0.0f;
				// below Cmatrix diagonal (columns)
				kind = i;
				for (int k = 0; k < i; k++) {
					sum += CMatrix[kind] * r_strain[k];
					kind += C_cols - k - 1;
				}
				// above CMatrix diagonal (rows)
				kind = i*C_cols - i*(i - 1) / 2;
				for (int k = i; k < C_cols; k++) {
					sum += CMatrix[kind] * r_strain[k];
					kind++;
				}
				s_u_vector[i*blockDim.x + threadIdx.x] = sum;
			}
		}
		break;
case(14): // Cartilage Endplate
			// CMatrix is symmetric, so only half of the matrix is stored
		del = E_end / ((1.0f + nu_end)*(1.0f - (2.0f*nu_end)));
		CMatrix[0] = del*(1.0f - nu_end);
		CMatrix[1] = del*nu_end;
		CMatrix[2] = del*nu_end;
		CMatrix[3] = 0;
		CMatrix[4] = 0;
		CMatrix[5] = 0;
		CMatrix[6] = del*(1.0f - nu_end);
		CMatrix[7] = del*nu_end;
		CMatrix[8] = 0;
		CMatrix[9] = 0;
		CMatrix[10] = 0;
		CMatrix[11] = del*(1.0f - nu_end);
		CMatrix[12] = 0;
		CMatrix[13] = 0;
		CMatrix[14] = 0;
		CMatrix[15] = del*(0.5f - nu_end);
		CMatrix[16] = 0;
		CMatrix[17] = 0;
		CMatrix[18] = del*(0.5f - nu_end);
		CMatrix[19] = 0;
		CMatrix[20] = del*(0.5f - nu_end);

		//Adjust strain for shear strain
		r_strain[3] *= 2.0f;
		r_strain[4] *= 2.0f;
		r_strain[5] *= 2.0f;
		{
			for (int i = 0; i < Ssize; i++) {
```

```
                                        sum = 0.0f;
                                        // below Cmatrix diagonal (columns)
                                        kind = i;
                                        for (int k = 0; k < i; k++) {
                                                sum += CMatrix[kind] * r_strain[k];
                                                kind += C_cols - k - 1;
                                        }
                                        // above CMatrix diagonal (rows)
                                        kind = i*C_cols - i*(i - 1) / 2;
                                        for (int k = i; k < C_cols; k++) {
                                                sum += CMatrix[kind] * r_strain[k];
                                                kind++;
                                        }
                                        s_u_vector[i*blockDim.x + threadIdx.x] = sum;
                                }
                        }
                        break;
                default:
                        break;
                }

                AF_V_M[threadIdx.x] = volumes[ind];

                // Compute CBLpart1 and store in register memory using symmetric matrix indexing
                {
                        for (int i = 0; i < C_cols; i++) {
                                for (int j = 0; j < tet_element_dof; j++) {
                                        sum = 0.0f;
                                        // below Cmatrix diagonal (columns)
                                        kind = i;
                                        for (int k = 0; k < i; k++) {
                                                sum += CMatrix[kind] * BL[k*tet_element_dof + j];
                                                kind += C_cols - k - 1;
                                        }
                                        // above CMatrix diagonal (rows)
                                        kind = i*C_cols - i*(i - 1) / 2;
                                        for (int k = i; k < C_cols; k++) {
                                                sum += CMatrix[kind] * BL[k*tet_element_dof + j];
                                                kind++;
                                        }
                                        CB[((i*tet_element_dof + j)*blockDim.x) + threadIdx.x] = sum;
                                }
                        }
                }
                // Compute KL
                {
                        for (int i = 0; i < tet_element_dof; i++) {
                                for (int j = i; j < tet_element_dof; j++) { // j=i for sym
                                        sum = 0.0f;
                                        kind = dof_ind[(ind*ematrix_ref_size) + i*tet_element_dof + j];
                                        for (int k = 0; k < C_cols; k++)
                                                sum += BL[k*tet_element_dof + i] * CB[((k*tet_element_dof +
j)*blockDim.x) + threadIdx.x];
                                        sum *= AF_V_M[threadIdx.x];
                                        atomicAdd(&K_matrix[kind], sum);
                                }
                        }
                }
                // Also calculate nodal force vector for each element
                {
                        for (int j = 0; j < tet_element_dof; j++) {
```

```
                                        sum = 0.0f;
                                        for (int k = 0; k < C_cols; k++)
                                                sum += BL[k*tet_element_dof + j] * s_u_vector[k*blockDim.x +
threadIdx.x];
                                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (j*blockDim.x) +
threadIdx.x] = sum*AF_V_M[threadIdx.x];
                                }
                        }
                        // Same procedure to calculate KNL
                        // Compute BNLT*S
                        {
                                BL[0] = (s_u_vector[0 * blockDim.x + threadIdx.x] * r_IntDeriv[0]) + (s_u_vector[5 *
blockDim.x + threadIdx.x] * r_IntDeriv[1]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[2]);
                                BL[1] = (s_u_vector[5 * blockDim.x + threadIdx.x] * r_IntDeriv[0]) + (s_u_vector[1 *
blockDim.x + threadIdx.x] * r_IntDeriv[1]) + (s_u_vector[3 * blockDim.x + threadIdx.x] * r_IntDeriv[2]);
                                BL[2] = (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[0]) + (s_u_vector[3 *
blockDim.x + threadIdx.x] * r_IntDeriv[1]) + (s_u_vector[2 * blockDim.x + threadIdx.x] * r_IntDeriv[2]);
                                BL[3] = (s_u_vector[0 * blockDim.x + threadIdx.x] * r_IntDeriv[3]) + (s_u_vector[5 *
blockDim.x + threadIdx.x] * r_IntDeriv[4]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[5]);
                                BL[4] = (s_u_vector[5 * blockDim.x + threadIdx.x] * r_IntDeriv[3]) + (s_u_vector[1 *
blockDim.x + threadIdx.x] * r_IntDeriv[4]) + (s_u_vector[3 * blockDim.x + threadIdx.x] * r_IntDeriv[5]);
                                BL[5] = (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[3]) + (s_u_vector[3 *
blockDim.x + threadIdx.x] * r_IntDeriv[4]) + (s_u_vector[2 * blockDim.x + threadIdx.x] * r_IntDeriv[5]);
                                BL[6] = (s_u_vector[0 * blockDim.x + threadIdx.x] * r_IntDeriv[6]) + (s_u_vector[5 *
blockDim.x + threadIdx.x] * r_IntDeriv[7]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[8]);
                                BL[7] = (s_u_vector[5 * blockDim.x + threadIdx.x] * r_IntDeriv[6]) + (s_u_vector[1 *
blockDim.x + threadIdx.x] * r_IntDeriv[7]) + (s_u_vector[3 * blockDim.x + threadIdx.x] * r_IntDeriv[8]);
                                BL[8] = (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[6]) + (s_u_vector[3 *
blockDim.x + threadIdx.x] * r_IntDeriv[7]) + (s_u_vector[2 * blockDim.x + threadIdx.x] * r_IntDeriv[8]);
                                BL[9] = (s_u_vector[0 * blockDim.x + threadIdx.x] * r_IntDeriv[9]) + (s_u_vector[5 *
blockDim.x + threadIdx.x] * r_IntDeriv[10]) + (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[11]);
                                BL[10] = (s_u_vector[5 * blockDim.x + threadIdx.x] * r_IntDeriv[9]) + (s_u_vector[1 *
blockDim.x + threadIdx.x] * r_IntDeriv[10]) + (s_u_vector[3 * blockDim.x + threadIdx.x] * r_IntDeriv[11]);
                                BL[11] = (s_u_vector[4 * blockDim.x + threadIdx.x] * r_IntDeriv[9]) + (s_u_vector[3 *
blockDim.x + threadIdx.x] * r_IntDeriv[10]) + (s_u_vector[2 * blockDim.x + threadIdx.x] * r_IntDeriv[11]);
                        }
                        {
                                // Compute KNL (using CB as intermediary) and add to KL
                                CB[(0 * blockDim.x) + threadIdx.x] = ((BL[0] * r_IntDeriv[0]) + (BL[1] * r_IntDeriv[1]) +
(BL[2] * r_IntDeriv[2]))*AF_V_M[threadIdx.x];
                                CB[(1 * blockDim.x) + threadIdx.x] = ((BL[0] * r_IntDeriv[3]) + (BL[1] * r_IntDeriv[4]) +
(BL[2] * r_IntDeriv[5]))*AF_V_M[threadIdx.x];//=4
                                CB[(2 * blockDim.x) + threadIdx.x] = ((BL[0] * r_IntDeriv[6]) + (BL[1] * r_IntDeriv[7]) +
(BL[2] * r_IntDeriv[8]))*AF_V_M[threadIdx.x];//=8
                                CB[(3 * blockDim.x) + threadIdx.x] = ((BL[0] * r_IntDeriv[9]) + (BL[1] * r_IntDeriv[10]) +
(BL[2] * r_IntDeriv[11]))*AF_V_M[threadIdx.x];//=12
                                CB[(5 * blockDim.x) + threadIdx.x] = ((BL[3] * r_IntDeriv[3]) + (BL[4] * r_IntDeriv[4]) +
(BL[5] * r_IntDeriv[5]))*AF_V_M[threadIdx.x];
                                CB[(6 * blockDim.x) + threadIdx.x] = ((BL[3] * r_IntDeriv[6]) + (BL[4] * r_IntDeriv[7]) +
(BL[5] * r_IntDeriv[8]))*AF_V_M[threadIdx.x];//=9
                                CB[(7 * blockDim.x) + threadIdx.x] = ((BL[3] * r_IntDeriv[9]) + (BL[4] * r_IntDeriv[10]) +
(BL[5] * r_IntDeriv[11]))*AF_V_M[threadIdx.x];//=13
                                CB[(10 * blockDim.x) + threadIdx.x] = ((BL[6] * r_IntDeriv[6]) + (BL[7] * r_IntDeriv[7]) +
(BL[8] * r_IntDeriv[8]))*AF_V_M[threadIdx.x];
                                CB[(11 * blockDim.x) + threadIdx.x] = ((BL[6] * r_IntDeriv[9]) + (BL[7] * r_IntDeriv[10]) +
(BL[8] * r_IntDeriv[11]))*AF_V_M[threadIdx.x];//=14
                                CB[(15 * blockDim.x) + threadIdx.x] = ((BL[9] * r_IntDeriv[9]) + (BL[10] * r_IntDeriv[10])
+ (BL[11] * r_IntDeriv[11]))*AF_V_M[threadIdx.x];

                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 0]], CB[(0 * blockDim.x) +
threadIdx.x]);
```

```
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 3]], CB[(1 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 6]], CB[(2 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 9]], CB[(3 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 13]], CB[(0 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 16]], CB[(1 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 19]], CB[(2 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 22]], CB[(3 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 26]], CB[(0 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 29]], CB[(1 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 32]], CB[(2 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 35]], CB[(3 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 36]], CB[(1 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 39]], CB[(5 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 42]], CB[(6 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 45]], CB[(7 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 49]], CB[(1 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 52]], CB[(5 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 55]], CB[(6 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 58]], CB[(7 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 62]], CB[(1 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 65]], CB[(5 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 68]], CB[(6 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 71]], CB[(7 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 72]], CB[(2 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 75]], CB[(6 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 78]], CB[(10 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 81]], CB[(11 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 85]], CB[(2 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 88]], CB[(6 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 91]], CB[(10 * blockDim.x) +
threadIdx.x]);
```

```
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 94]], CB[(11 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 98]], CB[(2 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 101]], CB[(6 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 104]], CB[(10 * blockDim.x) +
threadIdx.x]);
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 107]], CB[(11 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 108]], CB[(3 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 111]], CB[(7 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 114]], CB[(11 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 117]], CB[(15 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 121]], CB[(3 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 124]], CB[(7 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 127]], CB[(11 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 130]], CB[(15 * blockDim.x) +
threadIdx.x]);
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 134]], CB[(3 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 137]], CB[(7 * blockDim.x) +
threadIdx.x]); // remove for sym
                                //atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 140]], CB[(11 * blockDim.x) +
threadIdx.x]); // remove for sym
                                atomicAdd(&K_matrix[dof_ind[(ind*ematrix_ref_size) + 143]], CB[(15 * blockDim.x) +
threadIdx.x]);
                        }
                }
                // Calculate stiffness matrix for ligaments
                //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; (ind >= solid_element_count) && (ind <
(solid_element_count + lig_count)); ind += blockDim.x * gridDim.x){
                else if ((type_ind[ind] == 25))
                {
                        register float lig_linstiffness;
                        register float lig_nonlinstiffness;
                        register float r_lig_cosines[global_dof];
                        register float r_lig_force;
                        register float r_lig_E;
                        register float r_lig_lengths0 = lig_lengths0[ind];
                        register float r_lengths;
                        register float r_lig_disp;
                        register float x_diff = 0;
                        register float y_diff = 0;
                        register float z_diff = 0;
                        register float sum = 0.0f;
                        register int vb_node = 0;
                        register int ground_node = 0;
                        // Calculate the current lig length
                        vb_node = d_element_ind[(ind*nodes_per_tet)];
                        ground_node = d_element_ind[(ind*nodes_per_tet) + 1];

                        x_diff = node_loc[ground_node*global_dof] - node_loc[vb_node*global_dof];
                        y_diff = node_loc[ground_node*global_dof + 1] - node_loc[vb_node*global_dof + 1];
```

```
z_diff = node_loc[ground_node*global_dof + 2] - node_loc[vb_node*global_dof + 2];

r_lig_cosines[0] = x_diff / r_lig_lengths0;
r_lig_cosines[1] = y_diff / r_lig_lengths0;
r_lig_cosines[2] = z_diff / r_lig_lengths0;

r_lengths = (x_diff*x_diff) + (y_diff*y_diff) + (z_diff*z_diff);
r_lig_disp = sqrtf(r_lengths) - r_lig_lengths0;

r_lig_E = 0.0f;
r_lig_force = 0.0f;
switch (body_ind[ind])
{
case(5): //ALL
        if (r_lig_disp < -0.000001f) {
                r_lig_E = 0.0f;
                r_lig_disp = 0.0f;
                r_lig_force = 0.0f;
        }
        else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 0.27f)) {
                r_lig_E = 44.0741f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
        }
        else if ((r_lig_disp >= 0.27f) && (r_lig_disp < 0.51f)) {
                r_lig_E = 202.4167f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.27f)*r_lig_E) + (11.9f / 4.0f);
        }
        else if ((r_lig_disp >= 0.51f) && (r_lig_disp < 0.92f)) {
                r_lig_E = 280.2439f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.51f)*r_lig_E) + (60.48f / 4.0f);
        }
        else {
                r_lig_E = 1091.375f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.92f)*r_lig_E) + (175.38f / 4.0f);
        }
        lig_linstiffness = r_lig_E;
        break;
case(6): // PLL
        if (r_lig_disp < -0.000001f) {
                r_lig_E = 0.0f;
                r_lig_disp = 0.0f;
                r_lig_force = 0.0f;
        }
        else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 0.9f)) {
                r_lig_E = 0.6667f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
        }
        else if ((r_lig_disp >= 0.9f) && (r_lig_disp < 1.0f)) {
                r_lig_E = 12.0f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.9f)*r_lig_E) + (0.6f / 4.0f);
        }
        else if ((r_lig_disp >= 1.0f) && (r_lig_disp < 1.2f)) {
                r_lig_E = 66.0f / 4.0f;
                r_lig_force = ((r_lig_disp - 1.0f)*r_lig_E) + (1.8f / 4.0f);
        }
        else if ((r_lig_disp >= 1.2f) && (r_lig_disp < 1.4f)) {
                r_lig_E = 345.0f / 4.0f;
                r_lig_force = ((r_lig_disp - 1.2f)*r_lig_E) + (15.0f / 4.0f);
        }
        else {
                r_lig_E = 300.0f / 4.0f;
```

```
                r_lig_force = ((r_lig_disp - 1.4f)*r_lig_E) + (84.0f / 4.0f);
        }
        lig_linstiffness = r_lig_E;
        break;
case(7): //SSL
        if (r_lig_disp < -0.000001f) {
                r_lig_E = 0.0f;
                r_lig_disp = 0.0f;
                r_lig_force = 0.0f;
        }
        else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 7.9987f)) {
                r_lig_E = 1.499994f;
                r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
        }
        else if ((r_lig_disp >= 7.9987f) && (r_lig_disp < 9.9984f)) {
                r_lig_E = 3.179977f;
                r_lig_force = ((r_lig_disp - 7.9987f)*r_lig_E) + 11.998f;
        }
        else if ((r_lig_disp >= 9.9984f) && (r_lig_disp < 13.998f)) {
                r_lig_E = 20.39879f;
                r_lig_force = ((r_lig_disp - 9.9984f)*r_lig_E) + 18.357f;
        }
        else {
                r_lig_E = 49.95609f;
                r_lig_force = ((r_lig_disp - 13.998f)*r_lig_E) + 99.944f;
        }
        lig_linstiffness = r_lig_E;
        break;
case(8): // ISL
        if (r_lig_disp < -0.000001f) {
                r_lig_E = 0.0f;
                r_lig_disp = 0.0f;
                r_lig_force = 0.0f;
        }
        else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 2.155f)) {
                r_lig_E = 0.846088f / 4.0f;
                r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
        }
        else if ((r_lig_disp >= 2.155f) && (r_lig_disp < 3.1232f)) {
                r_lig_E = 0.885478f / 4.0f;
                r_lig_force = ((r_lig_disp - 2.155f)*r_lig_E) + (1.82332f / 4.0f);
        }
        else if ((r_lig_disp >= 3.1232f) && (r_lig_disp < 4.6848f)) {
                r_lig_E = 8.820031f / 4.0f;
                r_lig_force = ((r_lig_disp - 3.1232f)*r_lig_E) + (2.68064f / 4.0f);
        }
        else {
                r_lig_E = 14.42259f / 4.0f;
                r_lig_force = ((r_lig_disp - 4.6848f)*r_lig_E) + (16.454f / 4.0f);
        }
        lig_linstiffness = r_lig_E;
        break;
case(9): // TL
        if (r_lig_disp < -0.000001f) {
                r_lig_E = 0.0f;
                r_lig_disp = 0.0f;
                r_lig_force = 0.0f;
        }
        else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 5.069f)) {
                r_lig_E = 0.300059f;
                r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
```

```
		}
		else if ((r_lig_disp >= 5.069f) && (r_lig_disp < 6.489f)) {
			r_lig_E = 3.209155f;
			r_lig_force = ((r_lig_disp - 5.069f)*r_lig_E) + 1.521f;
		}
		else {
			r_lig_E = 16.08076f;
			r_lig_force = ((r_lig_disp - 6.489f)*r_lig_E) + 6.078f;
		}
		lig_linstiffness = r_lig_E;
		break;
case(10): // LF (ligamentum flavum)
		if (r_lig_disp < -0.000001f) {
			r_lig_E = 0.0f;
			r_lig_disp = 0.0f;
			r_lig_force = 0.0f;
		}
		else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 3.0f)) {
			r_lig_E = 0.033333f;
			r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
		}
		else if ((r_lig_disp >= 3.0f) && (r_lig_disp < 3.5f)) {
			r_lig_E = 9.8f;
			r_lig_force = ((r_lig_disp - 3.0f)*r_lig_E) + 0.1f;
		}
		else if ((r_lig_disp >= 3.5f) && (r_lig_disp < 4.5f)) {
			r_lig_E = 5.0f;
			r_lig_force = ((r_lig_disp - 3.5f)*r_lig_E) + 5.0f;
		}
		else if ((r_lig_disp >= 4.5f) && (r_lig_disp < 5.0f)) {
			r_lig_E = 30.0f;
			r_lig_force = ((r_lig_disp - 4.5f)*r_lig_E) + 10.0f;
		}
		else {
			r_lig_E = 70.0f;
			r_lig_force = ((r_lig_disp - 5.0f)*r_lig_E) + 25.0f;
		}
		lig_linstiffness = r_lig_E;
		break;
case(11): // FL (Facet capsulary ligament)
		if (r_lig_disp < -0.000001f) {
			r_lig_E = 0.0f;
			r_lig_disp = 0.0f;
			r_lig_force = 0.0f;
		}
		else if ((r_lig_disp >= -0.000001f) && (r_lig_disp < 1.0f)) {
			r_lig_E = 0.1f;
			r_lig_force = ((r_lig_disp - 0.0f)*r_lig_E) + 0.0f;
		}
		else if ((r_lig_disp >= 1.0f) && (r_lig_disp < 2.0f)) {
			r_lig_E = 2.9f;
			r_lig_force = ((r_lig_disp - 1.0f)*r_lig_E) + 0.1f;
		}
		else if ((r_lig_disp >= 2.0f) && (r_lig_disp < 2.4f)) {
			r_lig_E = 7.5f;
			r_lig_force = ((r_lig_disp - 2.0f)*r_lig_E) + 3.0f;
		}
		else if ((r_lig_disp >= 2.4f) && (r_lig_disp < 2.7f)) {
			r_lig_E = 30.0f;
			r_lig_force = ((r_lig_disp - 2.4f)*r_lig_E) + 6.0f;
		}
```

```
		else {
			r_lig_E = 33.3333f;
			r_lig_force = ((r_lig_disp - 2.7f)*r_lig_E) + 15.0f;
		}
		lig_linstiffness = r_lig_E;
		break;
default:
		break;
}

lig_nonlinstiffness = r_lig_force / r_lig_lengths0;
// Ligament linear stiffness matrix ***SYMMETRICAL***
sum = (lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[0]) + lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 0]], sum);
sum = lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[1];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 1]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 6]], sum); // remove for sym
sum = lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 2]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 12]], sum); // remove for sym
sum = (-1.0f*lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[0]) - lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 3]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 18]], sum); // remove for sym
sum = -1.0f*lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[1];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 4]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 24]], sum); // remove for sym
sum = -1.0f*lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 5]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 30]], sum); // remove for sym
sum = (lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[1]) + lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 7]], sum);
sum = lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 8]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 13]], sum); // remove for sym
sum = -1.0f*lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[1];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 9]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 19]], sum); // remove for sym
sum = (-1.0f*lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[1]) - lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 10]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 25]], sum); // remove for sym
sum = -1.0f*lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 11]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 31]], sum); // remove for sym
sum = (lig_linstiffness*r_lig_cosines[2] * r_lig_cosines[2]) + lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 14]], sum);
sum = -1.0f*lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 15]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 20]], sum); // remove for sym
sum = -1.0f*lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 16]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 26]], sum); // remove for sym
sum = (-1.0f*lig_linstiffness*r_lig_cosines[2] * r_lig_cosines[2]) - lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 17]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 32]], sum); // remove for sym
sum = (lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[0]) + lig_nonlinstiffness;
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 21]], sum);
sum = lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[1];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 22]], sum);
//atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 27]], sum); // remove for sym
sum = lig_linstiffness*r_lig_cosines[0] * r_lig_cosines[2];
atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 23]], sum);
```

```
                        //atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 33]], sum); // remove for sym
                        sum = (lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[1]) + lig_nonlinstiffness;
                        atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 28]], sum);
                        sum = lig_linstiffness*r_lig_cosines[1] * r_lig_cosines[2];
                        atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 29]], sum);
                        //atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 34]], sum); // remove for sym
                        sum = (lig_linstiffness*r_lig_cosines[2] * r_lig_cosines[2]) + lig_nonlinstiffness;
                        atomicAdd(&K_matrix[dof_ind[ind*ematrix_ref_size + 35]], sum);

                        // ligament nodal force vector
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (0 * blockDim.x) + threadIdx.x] = -r_lig_force *
r_lig_cosines[0];
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (1 * blockDim.x) + threadIdx.x] = -r_lig_force *
r_lig_cosines[1];
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (2 * blockDim.x) + threadIdx.x] = -r_lig_force *
r_lig_cosines[2];
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (3 * blockDim.x) + threadIdx.x] = r_lig_force *
r_lig_cosines[0];
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (4 * blockDim.x) + threadIdx.x] = r_lig_force *
r_lig_cosines[1];
                        b_vector[(tet_element_dof*blockDim.x*blockIdx.x) + (5 * blockDim.x) + threadIdx.x] = r_lig_force *
r_lig_cosines[2];

        }

        else if (type_ind[ind] != -1) printf("SOMETHING IS WRONG IN SOLID_STIFFNESS AT ind = %i with
type_ind[ind] == %i\n", ind, type_ind[ind]);

}
//**************************CUDA functions used within the
solver*******************************************************************************
__global__ void VectorAssembly(float* vector_elements, float* vector_global, int* element_node_index, int* type_ind,
int* body_ind, int* share_element_ind, bool* share_bool_ind, int node_count) {
        register int global_vec_ind = 0;
        register int local_vec_ind = 0;
        register bool local_bool_ind = false;
        __shared__ float local_assembly[BLOCK_SIZE*nodes_per_tet*global_dof];
        // Initialize shared memory with zeros
        for (int i = threadIdx.x; i < BLOCK_SIZE*nodes_per_tet*global_dof; i += blockDim.x) {
                local_assembly[i] = 0.0f;
        }
        __syncthreads();

        register int ind = blockDim.x*blockIdx.x + threadIdx.x;

        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < solid_element_count; ind += blockDim.x *
gridDim.x){
        if (type_ind[ind] == 24) { // change back to 24
                {
                        for (int a = 0; a < nodes_per_tet; a++) {
                                local_vec_ind = share_element_ind[(ind*nodes_per_tet) + a];
                                for (int i = 0; i < global_dof; i++) {
                                        atomicAdd(&local_assembly[(local_vec_ind*global_dof) + i],
vector_elements[(tet_element_dof*blockDim.x*blockIdx.x) + (((a*global_dof) + i)*blockDim.x) + threadIdx.x]);

                                }
                        }
                }
                __syncthreads();
                {
                        for (int a = 0; a < nodes_per_tet; a++) {
```

```
                                        local_bool_ind = share_bool_ind[(ind*nodes_per_tet) + a];
                                        if (local_bool_ind == true) {
                                                global_vec_ind = element_node_index[(ind*nodes_per_tet) + a];
                                                local_vec_ind = share_element_ind[(ind*nodes_per_tet) + a];
                                                for (int i = 0; i < global_dof; i++) {
                                                        atomicAdd(&vector_global[(global_vec_ind*global_dof) + i],
local_assembly[(local_vec_ind*global_dof) + i]);
                                                }
                                        }
                                }
                        }
                }
                else if (type_ind[ind] == 25) {
                        {
                                for (int a = 0; a < nodes_per_lig; a++) {
                                        local_vec_ind = share_element_ind[(ind*nodes_per_tet) + a];
                                        for (int i = 0; i < global_dof; i++) {
                                                atomicAdd(&local_assembly[(local_vec_ind*global_dof) + i],
vector_elements[(tet_element_dof*blockDim.x*blockIdx.x) + (((a*global_dof) + i)*blockDim.x) + threadIdx.x]);

                                        }
                                }
                        }
                        __syncthreads();
                        {
                                for (int a = 0; a < nodes_per_lig; a++) {
                                        local_bool_ind = share_bool_ind[(ind*nodes_per_tet) + a];
                                        if (local_bool_ind == true) {
                                                local_vec_ind = share_element_ind[(ind*nodes_per_tet) + a];
                                                global_vec_ind = element_node_index[(ind*nodes_per_tet) + a];
                                                for (int i = 0; i < global_dof; i++) {
                                                        atomicAdd(&vector_global[(global_vec_ind*global_dof) + i],
local_assembly[(local_vec_ind*global_dof) + i]);
                                                }
                                        }
                                }
                        }
                }
}
__global__ void VectorDisassembly(float* global_vector, float* element_vector, int* element_nodes_index, int* type_ind,
int* body_ind, int node_count) {
        register int global_vec_ind = 0;

        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < solid_element_count; ind += blockDim.x *
gridDim.x){
        int ind = blockIdx.x*blockDim.x + threadIdx.x;
        if ((type_ind[ind] == 24)) { //change back to 24
                {
                        for (int a = 0; a < nodes_per_tet; a++) {
                                global_vec_ind = element_nodes_index[(ind*nodes_per_tet) + a];
                                for (int i = 0; i < global_dof; i++) {
                                        element_vector[(tet_element_dof*blockDim.x*blockIdx.x) +
(((a*global_dof) + i)*blockDim.x) + threadIdx.x] = global_vector[(global_vec_ind*global_dof) + i];
                                }
                        }
                }
        }
        else if ((type_ind[ind] == 25)) {
                {
                        for (int a = 0; a < nodes_per_lig; a++) {
                                global_vec_ind = element_nodes_index[(ind*nodes_per_tet) + a];
```

```
                                        for (int i = 0; i < global_dof; i++)
                                                element_vector[(tet_element_dof*blockDim.x*blockIdx.x) +
(((a*global_dof) + i)*blockDim.x) + threadIdx.x] = global_vector[(global_vec_ind*global_dof) + i];
                                }
                        }
                }
}
__global__ void VectorSubtraction(float* vector1, float* vector2, float* vector_result, int total_dof) {

        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < total_dof; ind += blockDim.x * gridDim.x){
        int ind = blockDim.x*blockIdx.x + threadIdx.x;
        if (ind < total_dof) {
                vector_result[ind] = vector1[ind] - vector2[ind];
        }
}
__global__ void ScalarOne(float* scalar) {
        *scalar = 1.0f;
}
__global__ void fillzeros(float* fill, int max_elements) {

        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < max_elements; ind += blockDim.x * gridDim.x){
        int ind = blockDim.x*blockIdx.x + threadIdx.x;
        if (ind < max_elements) {
                fill[ind] = 0.0f;
        }
}
__global__ void fillones(float* fill, int max_elements) {
        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < max_elements; ind += blockDim.x * gridDim.x){
        int ind = blockDim.x*blockIdx.x + threadIdx.x;
        if (ind < max_elements) {
                {
                        for (int i = 0; i < tet_element_dof; i++)
                                fill[(tet_element_dof*blockDim.x*blockIdx.x) + (i*blockDim.x) + threadIdx.x] =
0.0f;
                }
        }
}
// Update the solution vector
__global__ void UpdateSolNodes(float* x_coordinates, float* u_coordinates, float* u_vector, int node_size) {
        //for (int ind = blockIdx.x * blockDim.x + threadIdx.x; ind < node_size; ind += blockDim.x * gridDim.x){
        int ind = blockDim.x*blockIdx.x + threadIdx.x;
        register float r_u_vector;
        if (ind < node_size) {
                r_u_vector = u_vector[ind];
                u_coordinates[ind] += r_u_vector;
                x_coordinates[ind] += r_u_vector;
        }
}
// Load application function for non-remote-point method
__global__ void ForcePressure(float* b_vector, float* force, int* f_nodes, int f_node_count, int node_count) {
        int ind = blockDim.x*blockIdx.x + threadIdx.x;

        if (ind < node_count) {
                for (int i = 0; i < f_node_count; i++) {
                        if (ind == f_nodes[i]) {
                                for (int i = 0; i < global_dof; i++) b_vector[ind*global_dof + i] = force[i] /
(float)f_node_count;
                        }
                }
        }
}
```

```
// Load application function for non-remote-point method
__global__ void MomentPressure(float* b_vector, float* nodes_x, int* f_nodes, float* ma_params, int f_node_count, int
node_count) {
	int ind = blockDim.x*blockIdx.x + threadIdx.x;

	if (ind < node_count) {
		for (int i = 0; i < f_node_count; i++) {
			if (ind == f_nodes[i]) {
				//if (nodes_x[ind*global_dof + 2] < ma_params[0]) b_vector[ind*global_dof + 2] =
ma_params[1] * (nodes_x[i*global_dof + 2] - ma_params[0]);
				//else if (nodes_x[ind*global_dof + 2] > ma_params[0]) b_vector[ind*global_dof +
2] = ma_params[2] * (nodes_x[i*global_dof + 2] - ma_params[0]);
				if (nodes_x[ind*global_dof + 2] < ma_params[0]) { // 2 fe 0 lb
					if (nodes_x[ind*global_dof + 0] > ma_params[1])
b_vector[ind*global_dof + 1] = -ma_params[2]; // 0 fe 2 lb
					else b_vector[ind*global_dof + 1] = -ma_params[3];
				}
				else {
					if (nodes_x[ind*global_dof + 0] > ma_params[1])
b_vector[ind*global_dof + 1] = ma_params[4]; // 0 fe 2 lb
					else b_vector[ind*global_dof + 1] = ma_params[5];
				}
			}
		}
	}

}
__global__ void ApplyBC(float* K_matrix, float* b_vector, int* BC_nodes, int* BCapply_ind, float* BCapply_stiff, float
BC_value, int BC_node_count, int BC_stiff_count, int Ksize,
	int total_node_count) {
	int ind = blockDim.x*blockIdx.x + threadIdx.x;

	if (ind < BC_stiff_count) {
		K_matrix[BCapply_ind[ind]] = BCapply_stiff[ind];
	}
	if (ind < BC_node_count) {
		for (int i = 0; i < global_dof; i++) {
			b_vector[BC_nodes[ind] * global_dof + i] = BC_value;
		}
	}
}
__global__ void HydrostaticStress(float* h_stress, float* IntDeriv, float* volumes, float* u_vector, float* node_loc, int*
d_element_ind, int* type_ind, int* body_ind) {
	int ind = blockIdx.x*blockDim.x + threadIdx.x;
	const int strain_size = 6;
	const float E_NP = 1.0f;
	const float nu_NP = 0.49f;
	const int C_cols = 6;
	if ((type_ind[ind] == 24) && (body_ind[ind] == 2)) {
		// register memory for thread
		register float r_h_stress = 0.0f;
		register float del = E_NP / ((1.0f + nu_NP)*(1.0f - (2.0f*nu_NP)));; // also used for J3 Mooney-Rivlin
calc
		register float sum = 0.0f;
		register float l_vector[3][3]; // also holds the invar_deriv calcs for Mooney_Rivlin
		register float r_u_vector[tet_element_dof];
		register float r_x_vector[tet_element_dof];
		register float r_IntDeriv[tet_element_dof];
		register float r_strain[strain_size];
		register float r_stress_vector[strain_size];
		register float F_tensor[3][3];
```

```
		register float r_stress_PK2[3][3];
		register float r_stress_Cauchy[3][3];
		register float CMatrix[C_cols][C_cols] = {
			{ del*(1.0f - nu_NP), del*nu_NP, del*nu_NP, 0.0f, 0.0f, 0.0f },
			{ del*nu_NP, del*(1.0f - nu_NP), del*nu_NP, 0.0f, 0.0f, 0.0f },
			{ del*nu_NP, del*nu_NP, del*(1.0f - nu_NP), 0.0f, 0.0f, 0.0f },
			{ 0.0f, 0.0f, 0.0f, del*(0.5f - nu_NP), 0.0f, 0.0f },
			{ 0.0f, 0.0f, 0.0f, 0.0f, del*(0.5f - nu_NP), 0.0f },
			{ 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, del*(0.5f - nu_NP) }
		};

		// Move global memory to register/shared memory
		{
			for (int k = 0; k < tet_element_dof; k++) {
				r_u_vector[k] = u_vector[(tet_element_dof*blockDim.x*blockIdx.x) +
(k*blockDim.x) + threadIdx.x];
				r_IntDeriv[k] = IntDeriv[(tet_element_dof*blockDim.x*blockIdx.x) +
(k*blockDim.x) + threadIdx.x];
			}
			for (int i = 0; i < nodes_per_tet; i++) {
				for (int j = 0; j < global_dof; j++) r_x_vector[i*global_dof + j] =
node_loc[d_element_ind[ind*nodes_per_tet + i] * global_dof + j];
			}
		}
		// Calculate l_vector and F_tensor.  Could use F=I+l_vector instead if speed is an issue
		{
			for (int i = 0; i < global_dof; i++) {
				for (int j = 0; j < global_dof; j++) {
					l_vector[i][j] = 0.0f;
					F_tensor[i][j] = 0.0f;
					for (int k = 0; k < nodes_per_tet; k++) {
						l_vector[i][j] += (r_IntDeriv[global_dof*k + j] *
r_u_vector[global_dof*k + i]);
						F_tensor[i][j] += (r_IntDeriv[global_dof*k + j] *
r_x_vector[k*global_dof + i]);
					}
				}
			}
		}

		// Compute strain from BL and u
		r_strain[0] = l_vector[0][0];
		r_strain[1] = l_vector[1][1];
		r_strain[2] = l_vector[2][2];
		r_strain[3] = 0.5f*(l_vector[1][2] + l_vector[2][1]);
		r_strain[4] = 0.5f*(l_vector[0][2] + l_vector[2][0]);
		r_strain[5] = 0.5f*(l_vector[0][1] + l_vector[1][0]);
		for (int k = 0; k < global_dof; k++) {
			r_strain[0] += 0.5f*(l_vector[k][0] * l_vector[k][0]);
			r_strain[1] += 0.5f*(l_vector[k][1] * l_vector[k][1]);
			r_strain[2] += 0.5f*(l_vector[k][2] * l_vector[k][2]);
			r_strain[3] += 0.5f*(l_vector[k][1] * l_vector[k][2]);
			r_strain[4] += 0.5f*(l_vector[k][0] * l_vector[k][2]);
			r_strain[5] += 0.5f*(l_vector[k][0] * l_vector[k][1]);
		}

		//Adjust strain for shear strain
		r_strain[3] *= 2.0f;
		r_strain[4] *= 2.0f;
		r_strain[5] *= 2.0f;
		{
```

```
                        for (int i = 0; i < C_cols; i++) {
                                sum = 0.0f;
                                // below Cmatrix diagonal (columns)
                                for (int k = 0; k < C_cols; k++) {
                                        sum += CMatrix[i][k] * r_strain[k];
                                }
                                r_stress_vector[i] = sum;
                        }
                }
                r_stress_PK2[0][0] = r_stress_vector[0]; r_stress_PK2[1][1] = r_stress_vector[1]; r_stress_PK2[2][2] =
r_stress_vector[2];
                r_stress_PK2[0][1] = r_stress_vector[5]; r_stress_PK2[1][0] = r_stress_vector[5];
                r_stress_PK2[0][2] = r_stress_vector[4]; r_stress_PK2[2][0] = r_stress_vector[4];
                r_stress_PK2[1][2] = r_stress_vector[3]; r_stress_PK2[2][1] = r_stress_vector[3];
                del = (F_tensor[0][0] * F_tensor[1][1] * F_tensor[2][2]) + (F_tensor[0][1] * F_tensor[1][2] *
F_tensor[2][0]) + (F_tensor[0][2] * F_tensor[1][0] * F_tensor[2][1])
                        - (F_tensor[0][2] * F_tensor[1][1] * F_tensor[2][0]) - (F_tensor[0][1] * F_tensor[1][0] *
F_tensor[2][2]) - (F_tensor[0][0] * F_tensor[1][2] * F_tensor[2][1]);
                del = 1.0f / del;
                for (int i = 0; i < global_dof; i++) {
                        for (int j = 0; j < global_dof; j++) {
                                sum = 0.0f;
                                for (int k = 0; k < global_dof; k++) {
                                        for (int l = 0; l < global_dof; l++) {
                                                sum += F_tensor[i][k] * r_stress_PK2[k][l] * F_tensor[j][l];
                                        }
                                }
                                r_stress_Cauchy[i][j] = sum*del;
                        }
                }
                //sum = 0.0f;
                //for (int i = 0; i < global_dof; i++) sum += r_stress_Cauchy[i][i];
                r_h_stress = 0.333333333333333f * (r_stress_Cauchy[0][0] + r_stress_Cauchy[1][1] +
r_stress_Cauchy[2][2]);
                h_stress[ind] = r_h_stress;
        }
        else h_stress[ind] = 0.0f;
}
// Main function
int main(int argc, char *argv[])
{
        // Vectors to collect data from files
        vector<float> node_loc_vec;
        vector<int> element_i_vec;
        vector<int> element_j_vec;
        vector<int> csr_column_vec;
        vector<int> csr_row_vec;
        vector<int> element_mat_vec;
        vector<int> element_type_vec;
        vector<int> element_sec_vec;
        vector<int> share_element_vec;
        vector<bool> share_bool_vec;
        vector<int> BC_node_vec;
        vector<int> fib_ind_vec;
        vector<int> f_ind_vec;
        // Open the files that have the nodes and elements
        ifstream all_node_file("NODE_LOC_BEFORE.txt");
        ifstream share_bool_file("SHARED_GLOBAL_ASSEMBLY_IND.txt");
        ifstream fib_ind_file("AF_IND_PRESSURE.txt");
        ifstream f_ind_file("F_NODES.txt"); // from ANSYS (therefore base 1) so minus 1
```

```
                                                                        // No need to minus 1 - already
handled in the other program
        string line;
        float nbuffer1;
        float nbuffer2, nbuffer3, nbuffer4;
        int bbuffer1, bbuffer2, bbuffer3, bbuffer4;
        //*******************************Use vectors to read from files, then copy vectors into
arrays************************
        // collect all nodes
        if (all_node_file.is_open())
        {
                while (getline(all_node_file, line))
                {
                        all_node_file >> nbuffer1 >> nbuffer2 >> nbuffer3 >> nbuffer4;
                        node_loc_vec.push_back(nbuffer2);
                        node_loc_vec.push_back(nbuffer3);
                        node_loc_vec.push_back(nbuffer4);
                }
                all_node_file.close();
        }
        else cout << "Unable to open file1." << endl;

        vector<int> element_i_buff;
        ifstream element_i_file("ELEMENT_IND_I.binary", ios::binary);
        element_i_file.unsetf(ios::skipws);
        streampos fileSize;
        element_i_file.seekg(0, ios::end);
        fileSize = element_i_file.tellg();
        element_i_file.seekg(0, ios::beg);
        element_i_buff.resize(fileSize / sizeof(int));
        element_i_file.read((char *)element_i_buff.data(), fileSize);

        for (int i = 0; i < (int)element_i_buff.size() / 8; i++) {
                element_i_vec.push_back(element_i_buff[i * 8]);
                element_i_vec.push_back(element_i_buff[i * 8 + 1]);
                element_i_vec.push_back(element_i_buff[i * 8 + 2]);
                element_i_vec.push_back(element_i_buff[i * 8 + 3]);
                element_mat_vec.push_back(element_i_buff[i * 8 + 4]);
                element_type_vec.push_back(element_i_buff[i * 8 + 5]);
                element_sec_vec.push_back(element_i_buff[i * 8 + 7]);
        }

        ifstream element_j_file("ELEMENT_IND_J.binary", ios::binary);
        element_j_file.unsetf(ios::skipws);
        element_j_file.seekg(0, ios::end);
        fileSize = element_j_file.tellg();
        element_j_file.seekg(0, ios::beg);
        element_j_vec.resize(fileSize / sizeof(int));
        element_j_file.read((char *)element_j_vec.data(), fileSize);

        ifstream csr_column_file("CSR_COLUMN.binary", ios::binary);
        csr_column_file.unsetf(ios::skipws);
        csr_column_file.seekg(0, ios::end);
        fileSize = csr_column_file.tellg();
        csr_column_file.seekg(0, ios::beg);
        csr_column_vec.resize(fileSize / sizeof(int));
        csr_column_file.read((char *)csr_column_vec.data(), fileSize);
```

```
ifstream csr_row_file("CSR_ROW.binary", ios::binary);
csr_row_file.unsetf(ios::skipws);
csr_row_file.seekg(0, ios::end);
fileSize = csr_row_file.tellg();
csr_row_file.seekg(0, ios::beg);
csr_row_vec.resize(fileSize / sizeof(int));
csr_row_file.read((char *)csr_row_vec.data(), fileSize);

ifstream share_element_file("SHARED_ASSEMBLY_IND.binary", ios::binary);
share_element_file.unsetf(ios::skipws);
share_element_file.seekg(0, ios::end);
fileSize = share_element_file.tellg();
share_element_file.seekg(0, ios::beg);
share_element_vec.resize(fileSize / sizeof(int));
share_element_file.read((char *)share_element_vec.data(), fileSize);

if (share_bool_file.is_open())
{
        while (getline(share_bool_file, line))
        {
                share_bool_file >> bbuffer1 >> bbuffer2 >> bbuffer3 >> bbuffer4;
                share_bool_vec.push_back(bbuffer1 != 0);
                share_bool_vec.push_back(bbuffer2 != 0);
                share_bool_vec.push_back(bbuffer3 != 0);
                share_bool_vec.push_back(bbuffer4 != 0);
        }
        share_bool_file.close();
}
else cout << "Unable to open share bool file." << endl;

ifstream BC_node_file("BC_NODES_PRESSURE.binary", ios::binary);
BC_node_file.unsetf(ios::skipws);
BC_node_file.seekg(0, ios::end);
fileSize = BC_node_file.tellg();
BC_node_file.seekg(0, ios::beg);
BC_node_vec.resize(fileSize / sizeof(int));
BC_node_file.read((char *)BC_node_vec.data(), fileSize);

// collect nodes for BC application
if (f_ind_file.is_open())
{
        while (getline(f_ind_file, line))
        {
                f_ind_file >> nbuffer1 >> nbuffer2 >> nbuffer3 >> nbuffer4;
                f_ind_vec.push_back((int)nbuffer1 - 1);
        }
        f_ind_file.close();
}
else cout << "Unable to open F node file." << endl;
// Collect the array for the annulus fibres
if (fib_ind_file.is_open()) {
        while (getline(fib_ind_file, line)) {
                fib_ind_file >> nbuffer2 >> nbuffer3;
                fib_ind_vec.push_back((int)nbuffer2);
                fib_ind_vec.push_back((int)nbuffer3);
        }
        fib_ind_file.close();
}
else cout << "Unable to open fib node file." << endl;

// Copy vectors into arrays
```

```
	//for (int i = 0; i < (int)element_ki_vec.size(); i++) { printf("element_ki_vec[%i + %i] = %i\n", i / nodes_per_tet, i % nodes_per_tet, element_ki_vec[i]); cin.get(); }

	// Element index organized and optimized for the different element types
	int* element_i = new int[element_i_vec.size()];
	int* body_ind = new int[element_mat_vec.size()]; // material ID for each element
	int* sec_ind = new int[element_sec_vec.size()];
	int* type_ind = new int[element_type_vec.size()];
	solid_element_count = 0;
	lig_count = 0;
	blank_count = 0;
	total_element_count = 0;
	while (total_element_count < element_mat_vec.size()) {
		if (element_type_vec[total_element_count] == 24) { // 24
			element_i[total_element_count*nodes_per_tet] = element_i_vec[total_element_count*nodes_per_tet];
			element_i[total_element_count*nodes_per_tet + 1] = element_i_vec[total_element_count*nodes_per_tet + 1];
			element_i[total_element_count*nodes_per_tet + 2] = element_i_vec[total_element_count*nodes_per_tet + 2];
			element_i[total_element_count*nodes_per_tet + 3] = element_i_vec[total_element_count*nodes_per_tet + 3];
			if (element_mat_vec[total_element_count] == 3) { // 3
				body_ind[total_element_count] = 1; // cancellous bone
			}
			else if (element_mat_vec[total_element_count] == 26) { // 3
				body_ind[total_element_count] = 26; // cortical bone
			}
			else if ((element_mat_vec[total_element_count] == 24)) { //24
				body_ind[total_element_count] = 3; // annulus fibrosus
				if ((element_sec_vec[total_element_count] == 15) || (element_sec_vec[total_element_count] == 19)) {
					sec_ind[total_element_count] = 1;
				}
				else if ((element_sec_vec[total_element_count] == 14) || (element_sec_vec[total_element_count] == 4)) {
					sec_ind[total_element_count] = 2;
				}
				else if ((element_sec_vec[total_element_count] == 1) || (element_sec_vec[total_element_count] == 16)) {
					sec_ind[total_element_count] = 3;
				}
				else if ((element_sec_vec[total_element_count] == 5) || (element_sec_vec[total_element_count] == 17)) {
					sec_ind[total_element_count] = 4;
				}
				else {
					cout << "Section numbers for annulus fibrosus don't match: please adjust section numbers" << endl;
					cin.get();
				}
			}
			else if ((element_mat_vec[total_element_count] == 4) || (element_mat_vec[total_element_count] == 1)) { // 4
				body_ind[total_element_count] = 13; // posterior elements
			}
			else if (element_mat_vec[total_element_count] == 5) { //5
				body_ind[total_element_count] = 14; // cartilage endplate
			}
			else if (element_mat_vec[(total_element_count)] == 2) { // change this back to 2
				body_ind[(total_element_count)] = 2;
```

```
                        }
                        else {
                                cout << "Material numbers don't match, please adjust material numbers" << endl;
                                cin.get();
                        }
                        solid_element_count++;
                }
                else if (element_type_vec[total_element_count] == 25) {
                        element_i[total_element_count*nodes_per_tet] =
element_i_vec[total_element_count*nodes_per_tet];
                        element_i[total_element_count*nodes_per_tet + 1] =
element_i_vec[total_element_count*nodes_per_tet + 1];
                        element_i[total_element_count*nodes_per_tet + 2] = -1;
                        element_i[(total_element_count*nodes_per_tet) + 3] = -1;
                        if (element_sec_vec[total_element_count] == 50) { //35 ALL 50
                                body_ind[total_element_count] = 5;
                        }
                        else if (element_sec_vec[total_element_count] == 51) {//36 PLL 51
                                body_ind[total_element_count] = 6;
                        }
                        else if (element_sec_vec[total_element_count] == 52) {//37 SSL 52
                                body_ind[total_element_count] = 7;
                        }
                        else if (element_sec_vec[total_element_count] == 53) {//38 ISL 53
                                body_ind[total_element_count] = 8;
                        }
                        else if ((element_sec_vec[total_element_count] == 54) ||
(element_sec_vec[total_element_count] == 55)) {//40 39 TL 54 55
                                body_ind[total_element_count] = 9;
                        }
                        else if ((element_sec_vec[total_element_count] == 56) ||
(element_sec_vec[total_element_count] == 57)) {//41 42 LF 56 57
                                body_ind[total_element_count] = 10;
                        }
                        else if ((element_sec_vec[total_element_count] == 58) ||
(element_sec_vec[total_element_count] == 59)) {//43 44 FL 58 59
                                body_ind[total_element_count] = 11;
                        }
                        else {
                                cout << "Material numbers didn't match: Please renumber the material numbers" <<
endl;
                                cin.get();
                        }
                        lig_count++;
                }
                else if ((element_type_vec[total_element_count] == -1)) {
                        element_i[(total_element_count)*nodes_per_tet] = -1;
                        element_i[(total_element_count)*nodes_per_tet + 1] = -1;
                        element_i[(total_element_count)*nodes_per_tet + 2] = -1;
                        element_i[(total_element_count)*nodes_per_tet + 3] = -1;
                        body_ind[total_element_count] = -1;
                        blank_count++;
                }
                else {
                        cout << "Error: element type number doesn't exist" << endl;
                        cin.get();
                }
                type_ind[total_element_count] = element_type_vec[total_element_count];
                total_element_count++;
        }
```

```
int* element_j = new int[(total_element_count)*ematrix_ref_size];
for (int i = 0; i < total_element_count; i++) {
		for (int j = 0; j < ematrix_ref_size; j++) {
			element_j[(i*ematrix_ref_size) + j] = element_j_vec[(i*ematrix_ref_size) + j];
		}
}

csr_row_size = (int)csr_row_vec.size();
int* csr_row = new int[csr_row_size];
for (int i = 0; i < csr_row_size; i++) {
		csr_row[i] = csr_row_vec[i];
}
A_non_zeros = (int)csr_column_vec.size();
int* csr_col = new int[A_non_zeros];
for (int i = 0; i < A_non_zeros; i++) {
		csr_col[i] = csr_column_vec[i];
}

F_node_count = (int)f_ind_vec.size();
int* F_ind = new int[F_node_count];
for (int i = 0; i < F_node_count; i++) {
		F_ind[i] = f_ind_vec[i];
}
// Node locations
//auto max_rotdof_node = max_element(std::begin(shell_rotdof_vec), std::end(shell_rotdof_vec));
//total_shellnode_count = *max_rotdof_node + 1;
int node_loc_size = (int)node_loc_vec.size(); // adjust here for adding force_elements
total_node_count = ((int)node_loc_vec.size() / global_dof); // and here
total_dof = node_loc_size;
float* nodes_x = new float[total_dof];
for (int i = 0; i < (int)node_loc_size; i++) {
		nodes_x[i] = node_loc_vec[i];
}

BC_node_count = (int)BC_node_vec.size();
int* BC_node_ind = new int[BC_node_count + 1];
for (int i = 0; i < BC_node_count; i++) {
		BC_node_ind[i] = BC_node_vec[i];
}
int* share_element_ind = new int[share_element_vec.size()];
for (int i = 0; i < (int)share_element_vec.size(); i++) {
		share_element_ind[i] = share_element_vec[i];
}
bool* share_bool_ind = new bool[share_bool_vec.size()];
for (int i = 0; i < share_bool_vec.size(); i++) {
		share_bool_ind[i] = share_bool_vec[i];
}
int* fib_ind = new int[(int)fib_ind_vec.size()];
for (int i = 0; i < fib_ind_vec.size(); i++) {
		fib_ind[i] = (int)fib_ind_vec[i];
}

// Prep an indexing array for applying the boundary conditions
vector<int> BCapply_ind0_vec;
vector<int> BCapply_ind1;
vector<float> BCapply_stiff_vec;
//force_BC for applying BC to the force elements <- 0 for 0 force and 1 for all other forces
float* h_force = new float[force_node_dof];
h_force[0] = 0.0f;
h_force[1] = 0.0f;
h_force[2] = 0.0f; //1000000.0f or 600.0f
```

```
		h_force[3] = -7652.0f; //7652.0f
		h_force[4] = 0.0f;
		h_force[5] = 0.0f; // 7100.0f

		// Find moment application parameters ******FOR FLEXION/EXTENSION
ONLY********************************************************** 0=2 and 2=0
		float* ma_para = new float[6]; // includes (in order) z_mid_line, x_mid_line, m_left_ant, m_right_ant,
m_left_post, m_right_post
		float y_length_ant = nodes_x[F_ind[0] * global_dof + 2], y_length_post = nodes_x[F_ind[0] * global_dof + 2],
d_length = 0.0f, m_force[2] = { 0.0f,0.0f }, r_force = 0.0f; //"..global_dof+2" fe "+0" lb
		float z_length_left_ave = 0.0f, z_length_right_ave = 0.0f, z_ratio; //x_length_post_ave/ant_ave for bending
		float f_ave_left_ant, f_ave_right_ant, f_ave_left_post, f_ave_right_post;
		vector<int> F_nodes_left_ant;
		vector<int> F_nodes_right_ant;
		vector<int> F_nodes_left_post;
		vector<int> F_nodes_right_post;
		ma_para[0] = 0.0f;
		ma_para[1] = 0.0f;
		for (int i = 0; i < F_node_count; i++) {
			ma_para[0] += nodes_x[F_ind[i] * global_dof + 2]; //2 fe 0 lb
			ma_para[1] += nodes_x[F_ind[i] * global_dof + 0]; //0 fe 2 lb
			if (nodes_x[F_ind[i] * global_dof + 2] < y_length_ant) y_length_ant = nodes_x[F_ind[i] * global_dof +
2]; // negative z-direction (anterior) 2 fe 0 lb
			else if (nodes_x[F_ind[i] * global_dof + 2] >= y_length_post) y_length_post = nodes_x[F_ind[i] *
global_dof + 2]; // positive z-direction (posterior) 2 fe 0 lb
		}
		ma_para[0] = ma_para[0] / F_node_count;
		ma_para[1] = ma_para[1] / F_node_count;
		for (int i = 0; i < F_node_count; i++) {
			if (nodes_x[F_ind[i] * global_dof + 0] > ma_para[1]) z_length_left_ave += (nodes_x[F_ind[i] *
global_dof + 0] - ma_para[1]); // 0 fe 2 lb
			else if (nodes_x[F_ind[i] * global_dof + 0] <= ma_para[1]) z_length_right_ave += (ma_para[1] -
nodes_x[F_ind[i] * global_dof + 0]); // 0 fe 2 lb
		}
		y_length_ant = y_length_ant - ma_para[0];
		y_length_post = y_length_post - ma_para[0];
		z_length_left_ave = z_length_left_ave / F_node_count;
		z_length_right_ave = z_length_right_ave / F_node_count;
		//z_ratio = z_length_right_ave / z_length_left_ave;
		z_ratio = 0.639f; // 0.639f for flexion/extension 1.800f for lateral bending
		for (int i = 0; i < F_node_count; i++) {
			if (nodes_x[F_ind[i] * global_dof + 2] < ma_para[0]) { // 2 fe 0 lb
				if (nodes_x[F_ind[i] * global_dof + 0] < ma_para[1]) F_nodes_right_ant.push_back(F_ind[i]);
// 0 fe 2 lb
				else F_nodes_left_ant.push_back(F_ind[i]);

			}
			else if (nodes_x[F_ind[i] * global_dof + 2] >= ma_para[0]) { // 2 fe 0 lb
				if (nodes_x[F_ind[i] * global_dof + 0] < ma_para[1])
F_nodes_right_post.push_back(F_ind[i]); //0 fe 2 lb
				else F_nodes_left_post.push_back(F_ind[i]);
			}
		}
		d_length = (abs(y_length_ant) + abs(y_length_post)) / 4.0f;
		r_force = h_force[3] / (2.0f*d_length); // 3 fe 5 lb
		m_force[0] = (r_force*z_ratio) / (1.0f + z_ratio); // left because z_length_right_ave in ratio numerator
		m_force[1] = r_force - m_force[0];
		f_ave_left_ant = m_force[0] / F_nodes_left_ant.size();
		f_ave_right_ant = m_force[1] / F_nodes_right_ant.size();
		f_ave_left_post = m_force[0] / F_nodes_left_post.size();
		f_ave_right_post = m_force[1] / F_nodes_right_post.size();
```

```
	ma_para[2] = f_ave_left_ant;
	ma_para[3] = f_ave_right_ant;
	ma_para[4] = f_ave_left_post;
	ma_para[5] = f_ave_right_post;
	// Check resultant moment and force
	float sum = 0.0f;
	float F_result = 0.0f;
	float M_result[2] = { 0.0f,0.0f };
	for (int i = 0; i < (int)F_nodes_left_ant.size(); i++) {
		sum = -ma_para[2];
		M_result[0] += sum*(nodes_x[F_nodes_left_ant[i] * global_dof + 2] - ma_para[0]); // 2 fe 0 lb
		M_result[1] += sum*(nodes_x[F_nodes_left_ant[i] * global_dof + 0] - ma_para[1]); // 0 fe 2 lb
		F_result += sum;
	}
	for (int i = 0; i < (int)F_nodes_right_ant.size(); i++) {
		sum = -ma_para[3];
		M_result[0] += sum*(nodes_x[F_nodes_right_ant[i] * global_dof + 2] - ma_para[0]); // 2 fe 0 lb
		M_result[1] += sum*(nodes_x[F_nodes_right_ant[i] * global_dof + 0] - ma_para[1]); // 0 fe 2 lb
		F_result += sum;
	}
	for (int i = 0; i < (int)F_nodes_left_post.size(); i++) {
		sum = ma_para[4];
		M_result[0] += sum*(nodes_x[F_nodes_left_post[i] * global_dof + 2] - ma_para[0]); // 2 fe 0 lb
		M_result[1] += sum*(nodes_x[F_nodes_left_post[i] * global_dof + 0] - ma_para[1]); // 0 fe 2 lb
		F_result += sum;
	}
	for (int i = 0; i < (int)F_nodes_right_post.size(); i++) {
		sum = ma_para[5];
		M_result[0] += sum*(nodes_x[F_nodes_right_post[i] * global_dof + 2] - ma_para[0]); // 2 fe 0 lb
		M_result[1] += sum*(nodes_x[F_nodes_right_post[i] * global_dof + 0] - ma_para[1]); // 0 fe 2 lb
		F_result += sum;
	}
	printf("\n\n**Check the resultant moment and force.\nF_result = %f\nh_force[3] = %f while M_result = %f,
M_result_dir = %f\n",
		F_result, h_force[3], M_result[0], M_result[1]);
	//cin.get();

	// Indexing arrays for applying the boundary conditions
	bool BC_flag = false;
	for (int i = 0; i < total_node_count; i++) {
		BC_flag = false;
		for (int j = 0; j < BC_node_count; j++) {
			if (i == BC_node_ind[j]) {
				BC_flag = true; break;
			}
		}
		if (BC_flag == true) {
			for (int j = 0; j < global_dof; j++) {
				for (int k = csr_row[i*global_dof + j]; k < csr_row[i*global_dof + j + 1]; k++) {
					if ((i*global_dof + j) == csr_col[k]) {
						BCapply_ind1.push_back(k);
					}
					else {
						BCapply_ind0_vec.push_back(k);
						BCapply_stiff_vec.push_back(0.0f);
					}
				}
			}
		}
		else {
			for (int j = 0; j < global_dof; j++) {
```

```
                                        for (int k = csr_row[i*global_dof + j]; k < csr_row[i*global_dof + j + 1]; k++) {
                                                for (int l = 0; l < BC_node_count; l++) {
                                                        if (csr_col[k] == (BC_node_ind[l] * global_dof)) {
                                                                for (int m = 0; m < global_dof; m++) {
                                                                        BCapply_ind0_vec.push_back(k + m);
                                                                        BCapply_stiff_vec.push_back(0.0f);
                                                                }
                                                                break;
                                                        }
                                                }
                                        }
                                }
                        }
                }

                for (int i = 0; i < (int)BCapply_ind1.size(); i++) {
                        BCapply_ind0_vec.push_back(BCapply_ind1[i]);
                        BCapply_stiff_vec.push_back(1.0f);
                }
                BC_stiff_count = (int)BCapply_ind0_vec.size();
                int* BCapply_ind = new int[BC_stiff_count];
                float* BCapply_stiff = new float[BC_stiff_count];
                for (int i = 0; i < BC_stiff_count; i++) {
                        BCapply_ind[i] = BCapply_ind0_vec[i];
                        BCapply_stiff[i] = BCapply_stiff_vec[i];
                }

                GRID_SIZE_NON_ZEROS = (A_non_zeros / BLOCK_SIZE) + 1;
                GRID_SIZE1 = (total_element_count / BLOCK_SIZE);
                GRID_SIZE2 = (solid_element_count / BLOCK_SIZE);
                GRID_SIZE12 = (lig_count / BLOCK_SIZE);
                GRID_SIZE0 = (total_dof / BLOCK_SIZE) + 1;
                GRID_SIZE01 = ((total_node_count) / BLOCK_SIZE) + 1;
                GRID_SIZE_BC = (BC_stiff_count / BLOCK_SIZE) + 1;

                cout << "File read successful: nodes and elements acquired." << endl;
                //system("pause");

                // cuda Error checking while performing calculations
                cudaError_t cudaStatus = RealTimeSim(nodes_x, element_i, element_j, csr_row, csr_col, share_element_ind,
ma_para, share_bool_ind, type_ind, body_ind, sec_ind,
                        BC_node_ind, F_ind, BCapply_ind, BCapply_stiff, fib_ind, h_force);
                if (cudaStatus != cudaSuccess) {
                        fprintf(stderr, "\nRealTimeSim failed!\n");
                        cin.get();
                        return 1;
                }

                // Show success!
                cout << "Simulation success!!" << endl;

                // cudaDeviceReset must be called before exiting in order for profiling and
                // tracing tools such as Nsight and Visual Profiler to show complete traces.
                cudaStatus = cudaDeviceReset();
                if (cudaStatus != cudaSuccess) {
                        fprintf(stderr, "cudaDeviceReset failed!");
                        return 1;
                }

                // Write updated nodes to file
```

```
	cout << "Program Complete" << endl;
	cin.get();

	return 0;
}

// Proper pardiso_error checking will make debugging easier
cudaError_t RealTimeSim(float* nodes_x, int* element_i, int* element_j, int* csr_row, int* csr_col, int*
share_element_ind, float* ma_params,
	bool* share_bool_ind, int* type_ind, int* body_ind, int* sec_ind, int* BC_node_ind, int* f_ind, int*
BCapply_ind, float* BCapply_stiff, int* fib_ind, float* h_force)
{
	cudaError_t cudaStatus;

	// device variables to move data from host
	float* d_nodes_x = NULL;
	float* d_nodes_x_prev = NULL;
	float* d_nodes_x0 = NULL;
	int* d_element_i = NULL;
	int* d_element_j = NULL;
	int* d_csr_row = NULL;
	int* d_csr_col = NULL;
	int* d_share_element_ind = NULL;
	bool* d_share_bool_ind = NULL;
	int* d_type_ind = NULL;
	int* d_body_ind = NULL;
	int* d_sec_ind = NULL;
	int* d_BC_node_ind = NULL;
	int* d_F_ind = NULL;
	int* d_BCapply_ind = NULL;
	float* d_BCapply_stiff = NULL;
	int* d_fib_ind = NULL;
	float* d_ma_params = NULL;
	float* d_h_stress = NULL;
	float* h_h_stress = NULL;
	// Initialize stiffness matrices and force vectors
	const int solid_BLsize = 72;
	float* solid_inter_constants = NULL;
	float* solid_element_volumes = NULL;
	float* solid_BL0 = NULL;
	float* matrix_A = NULL;
	float* matrix_A_prev = NULL;
	float* lig_lengths0 = NULL;
	float* fib_lengths0_2 = NULL;
	// boundary condition and forces
	float BC_value = 0.0f;
	float* d_force = NULL;
	float* d_local_coords0 = NULL;
	float* d_local_coords = NULL;
	float *h_force_step = new float[global_dof * 2];
	float *h_step_size = new float[global_dof * 2];
	float *h_previous_step = new float[global_dof * 2];
	for (int i = 0; i < (global_dof * 2); i++) {
		h_force_step[i] = h_force[i];
		h_step_size[i] = h_force[i];
		h_previous_step[i] = 0.0f;
	}
	float* h_ma_step = new float[6];
	float* h_ma_force = new float[6];
	float* h_ma_previous = new float[6];
```

```
	h_ma_force[0] = ma_params[0]; h_ma_force[1] = ma_params[1]; h_ma_step[0] = ma_params[0]; h_ma_step[1] =
ma_params[1];
	for (int i = 2; i < 6; i++) { h_ma_force[i] = substep_size * ma_params[i]; h_ma_step[i] = substep_size *
ma_params[i]; } // *substep_size breaks force into two substeps (see non_substep = substep_size)
	h_ma_previous[0] = ma_params[0]; h_ma_previous[1] = ma_params[1];
	for (int i = 2; i < 6; i++) h_ma_previous[i] = 0.0f;
	float* d_force_step = NULL;
	// Initialize variables for preconditioned gradient solver
	float* u_global = NULL;
	float* u_sol_global = NULL;
	float* u_sol_global_prev = NULL;
	float* b_global = NULL;
	float* r_global = NULL;
	float* f_global = NULL;
	float* f_global_prev = NULL;
	float* u_elements = NULL;
	float* u_sol_elements = NULL;
	float* f_elements = NULL;
	//float etol_d = 1.0f;
	int NR_itr = 0;
	float substep = 0.0f;
	float non_substep = substep_size; // breaks force into two substeps
	int total_steps = int( 1.0f / substep_size);
	float* x_substeps = NULL;
	x_substeps = (float*)malloc(total_dof*total_steps * sizeof(float));
	int step_count = 0;
	int NR_itr_max = 10;
	int total_itr = 0;
	float* del_u_norm = NULL;
	float* u_norm = NULL;
	float* rf_diff_norm = NULL;
	float* ext_force_norm = NULL;
	float* h_rf_diff_norm = NULL;
	h_rf_diff_norm = (float*)malloc(sizeof(float));
	float* h_rf_diff_norm2 = NULL;
	h_rf_diff_norm2 = (float*)malloc(sizeof(float));
	float* h_ext_force_norm = NULL;
	h_ext_force_norm = (float*)malloc(sizeof(float));
	float* h_del_u_norm = NULL;
	h_del_u_norm = (float*)malloc(sizeof(float));
	float* h_del_u_norm2 = NULL;
	h_del_u_norm2 = (float*)malloc(sizeof(float));
	float* h_u_norm = NULL;
	h_u_norm = (float*)malloc(sizeof(float));
	h_h_stress = (float*)malloc(total_element_count * sizeof(float));
	cublasHandle_t handle;
	cudaEvent_t start, stop, start_matrix, stop_matrix, start_solve, stop_solve;
	float millisecond = 0.0f;
	float millisecond_matrix = 0.0f;
	float millisecond_solve = 0.0f;
	float* hf_matrix_A = NULL;
	hf_matrix_A = (float*)malloc(A_non_zeros * sizeof(float));
	float* hf_b_global = NULL;
	hf_b_global = (float*)malloc(2 * total_dof * sizeof(float));
	float* hf_u_global = NULL;
	hf_u_global = (float*)malloc(total_dof * sizeof(float));

	//***for calculating location and orientation of the remote node
	float remote_node_before[global_dof] = { 0.0f, 0.0f, 0.0f };
	for (int j = 0; j < global_dof; j++) {
		remote_node_before[j] = 0.0f;
```

```
		for (int i = 0; i < F_node_count; i++) remote_node_before[j] += nodes_x[f_ind[i] * global_dof + j];
	}
	for (int j = 0; j < global_dof; j++) remote_node_before[j] = remote_node_before[j] / F_node_count;
	float* remote_loc_coord_before = new float[F_node_count*global_dof];
	for (int i = 0; i < F_node_count; i++) {
		for (int j = 0; j < global_dof; j++) remote_loc_coord_before[i*global_dof + j] = nodes_x[f_ind[i] *
global_dof + j] - remote_node_before[j];
	}

	// PARDISO variables
	//*******************************REMEMBER THAT PARDISO IS INDEX BASE
ONE**********************************************************
	MKL_INT mtype = 2;	/* -2 Real symmetric indefinite; 2 symmetric positive definite; 1 Real structurally
symmetric 11 Real nonsymmetric */
	MKL_INT n_total = total_dof;
	MKL_INT* p_csr_row = new MKL_INT[total_dof + 1];
	MKL_INT* p_csr_col = new MKL_INT[A_non_zeros];
	for (int i = 0; i < total_dof + 1; i++) {
		p_csr_row[i] = csr_row[i] + 1;
	}
	for (int i = 0; i < A_non_zeros; i++) {
		p_csr_col[i] = csr_col[i] + 1;
	}
	MKL_INT nrhs = 1;	/* Number of right hand sides. */
	void *pt[64]; /* Internal solver memory pointer pt, */
					/* Pardiso control parameters. */
	MKL_INT iparm[64];
	MKL_INT maxfct, mnum, phase, pardiso_error, msglvl;
	/* Auxiliary variables. */
	MKL_INT p_i;
	float ddum;		/* Float dummy */
	MKL_INT idum;		/* Integer dummy. */
							/* ---------------------------------------------------------------------- */
							/* .. Setup Pardiso control parameters. */
							/* ---------------------------------------------------------------------- */
	for (p_i = 0; p_i < 64; p_i++)
	{
		iparm[p_i] = 0;
	}
	iparm[0] = 1;		/* No solver default */
	iparm[1] = 2;		/* Fill-in reordering from METIS */
	iparm[3] = 0;		/* No iterative-direct algorithm */
	iparm[4] = 0;		/* No user fill-in reducing permutation */
	iparm[5] = 0;		/* Write solution into x */
	iparm[6] = 0;		/* Not in use */
	iparm[7] = 2;		/* Max numbers of iterative refinement steps */
	iparm[8] = 0;		/* Not in use */
	iparm[9] = 13;		/* Perturb the pivot elements with 1E-13 */
	iparm[10] = 1;		/* Use nonsymmetric permutation and scaling MPS */
	iparm[11] = 0;		/* Not in use */
	iparm[12] = 0;		/* Maximum weighted matching algorithm is switched-off (default for symmetric). Try
iparm[12] = 1 in case of inappropriate accuracy */
	iparm[13] = 0;		/* Output: Number of perturbed pivots */
	iparm[14] = 0;		/* Not in use */
	iparm[15] = 0;		/* Not in use */
	iparm[16] = 0;		/* Not in use */
	iparm[17] = -1;		/* Output: Number of nonzeros in the factor LU */
	iparm[18] = -1;		/* Output: Mflops for LU factorization */
	iparm[19] = 0;		/* Output: Numbers of CG Iterations */
	iparm[27] = 1;			/* for single precision Intel MKL PARDISO */
	maxfct = 1;		/* Maximum number of numerical factorizations. */
```

```
	mnum = 1;	/* Which factorization to use. */
	msglvl = 0;		/* Print statistical information in file */
	pardiso_error = 0;		/* Initialize pardiso_error flag */
/* ------------------------------------------------
------------------- */
/* .. Initialize the internal solver memory
pointer. This is only */
/* necessary for the FIRST call of the
PARDISO solver. */
/* ------------------------------------------------
------------------- */
	for (p_i = 0; p_i < 64; p_i++)
	{
		pt[p_i] = 0;
	}

	// Choose which GPU to run on, change this on a multi-GPU system.
	cudaStatus = cudaSetDevice(0);
	//memory checks;
	size_t free_byte;
	size_t total_byte;

	cudaStatus = cudaMemGetInfo(&free_byte, &total_byte);
	double free_db = (double)free_byte;
	double total_db = (double)total_byte;
	double used_db = total_db - free_db;
	printf("GPU memory usage before cudaMalloc: used = %f MB, free = %f MB, total = %f MB\n", used_db /
1024.0 / 1024.0, free_db / 1024.0 / 1024.0, total_db / 1024.0 / 1024.0);
	// allocate GPU buffers
	cudaStatus = cudaMalloc(&d_nodes_x, total_dof * sizeof(float));
	cudaStatus = cudaMalloc(&d_nodes_x_prev, total_dof * sizeof(float));
	cudaStatus = cudaMalloc(&d_nodes_x0, total_dof * sizeof(float));
	cudaStatus = cudaMalloc(&d_element_i, ((total_element_count*nodes_per_tet)) * sizeof(int));
	cudaStatus = cudaMalloc(&d_element_j, ((total_element_count*ematrix_ref_size)) * sizeof(int));
	cudaStatus = cudaMalloc(&d_csr_row, csr_row_size * sizeof(int));
	cudaStatus = cudaMalloc(&d_csr_col, A_non_zeros * sizeof(int));
	cudaStatus = cudaMalloc(&d_share_element_ind, GRID_SIZE1*BLOCK_SIZE*nodes_per_tet * sizeof(int));
	cudaStatus = cudaMalloc(&d_share_bool_ind, GRID_SIZE1*BLOCK_SIZE*nodes_per_tet * sizeof(bool));
	cudaStatus = cudaMalloc(&d_type_ind, total_element_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_body_ind, total_element_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_sec_ind, total_element_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_BC_node_ind, BC_node_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_F_ind, F_node_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_BCapply_ind, BC_stiff_count * sizeof(int));
	cudaStatus = cudaMalloc(&d_BCapply_stiff, BC_stiff_count * sizeof(float));
	cudaStatus = cudaMalloc(&d_fib_ind, total_element_count*nodes_per_fib * sizeof(int));
	cudaStatus = cudaMalloc(&d_ma_params, 6 * sizeof(float));
	cudaStatus = cudaMalloc(&d_h_stress, total_element_count * sizeof(float));

	// Move memory from host to device
	cudaStatus = cudaMemcpy(d_nodes_x, nodes_x, total_dof * sizeof(float), cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_nodes_x_prev, nodes_x, total_dof * sizeof(float), cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_nodes_x0, nodes_x, total_dof * sizeof(float), cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_element_i, element_i, ((total_element_count*nodes_per_tet)) * sizeof(int),
cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_element_j, element_j, ((total_element_count*ematrix_ref_size)) * sizeof(int),
cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_csr_row, csr_row, csr_row_size * sizeof(int), cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_csr_col, csr_col, A_non_zeros * sizeof(int), cudaMemcpyHostToDevice);
	cudaStatus = cudaMemcpy(d_share_element_ind, share_element_ind,
GRID_SIZE1*BLOCK_SIZE*nodes_per_tet * sizeof(int), cudaMemcpyHostToDevice);
```

```
		cudaStatus = cudaMemcpy(d_share_bool_ind, share_bool_ind, GRID_SIZE1*BLOCK_SIZE*nodes_per_tet *
sizeof(bool), cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_type_ind, type_ind, total_element_count * sizeof(int), cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_body_ind, body_ind, total_element_count * sizeof(int),
cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_sec_ind, sec_ind, total_element_count * sizeof(int), cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_BC_node_ind, BC_node_ind, BC_node_count * sizeof(int),
cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_F_ind, f_ind, F_node_count * sizeof(int), cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_BCapply_ind, BCapply_ind, BC_stiff_count * sizeof(int),
cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_BCapply_stiff, BCapply_stiff, BC_stiff_count * sizeof(float),
cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_fib_ind, fib_ind, total_element_count * nodes_per_fib * sizeof(int),
cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_ma_params, h_ma_force, 6 * sizeof(float), cudaMemcpyHostToDevice);

		// Matrices for precalculations
		cudaStatus = cudaMalloc(&solid_inter_constants, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)) * sizeof(float));
		cudaStatus = cudaMalloc(&solid_element_volumes, total_element_count * sizeof(float));
		cudaStatus = cudaMalloc(&solid_BL0, ((total_element_count*solid_BLsize) + (BLOCK_SIZE*solid_BLsize)) *
sizeof(float));
		cudaStatus = cudaMalloc(&lig_lengths0, total_element_count * sizeof(float));
		cudaStatus = cudaMalloc(&matrix_A, A_non_zeros * sizeof(float));
		cudaStatus = cudaMalloc(&matrix_A_prev, A_non_zeros * sizeof(float));
		cudaStatus = cudaMalloc(&fib_lengths0_2, total_element_count * sizeof(float));

		// Vectors for preconditioned gradient solver
		cudaStatus = cudaMalloc(&u_global, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&b_global, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&r_global, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&f_global, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&f_global_prev, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&u_sol_global, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&u_sol_global_prev, total_dof * sizeof(float));
		cudaStatus = cudaMalloc(&u_elements, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)) * sizeof(float));
		cudaStatus = cudaMalloc(&u_sol_elements, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)) * sizeof(float));
		cudaStatus = cudaMalloc(&f_elements, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)) * sizeof(float));
		cudaStatus = cudaMalloc(&del_u_norm, sizeof(float));
		cudaStatus = cudaMalloc(&ext_force_norm, sizeof(float));
		cudaStatus = cudaMalloc(&u_norm, sizeof(float));
		cudaStatus = cudaMalloc(&rf_diff_norm, sizeof(float));

		cudaStatus = cudaMalloc(&d_force, force_node_dof * sizeof(float));
		cudaStatus = cudaMalloc(&d_force_step, force_node_dof * sizeof(float));
		cudaStatus = cudaPeekAtLastError();
		cudaStatus = cudaMemcpy(d_force, h_force, force_node_dof * sizeof(float), cudaMemcpyHostToDevice);
		cudaStatus = cudaMemcpy(d_force_step, h_force_step, force_node_dof * sizeof(float),
cudaMemcpyHostToDevice);
		cudaStatus = cudaPeekAtLastError();
		cudaStatus = cudaMalloc(&d_local_coords0, total_element_count*tet_element_dof * sizeof(float));
		cudaStatus = cudaMalloc(&d_local_coords, total_element_count*tet_element_dof * sizeof(float));

		cudaStatus = cudaGetLastError();
		if (cudaStatus != cudaSuccess) {
			fprintf(stderr, "solid cudaMalloc failed!");
			goto Error;
```

```
	}

	cudaStatus = cudaMemGetInfo(&free_byte, &total_byte);
	free_db = (double)free_byte;
	total_db = (double)total_byte;
	used_db = total_db - free_db;
	printf("GPU memory usage after cudaMalloc: used = %f MB, free = %f MB, total = %f MB\n", used_db / 1024.0 /
1024.0, free_db / 1024.0 / 1024.0, total_db / 1024.0 / 1024.0);

	// Get cublas ready for solver
	cublasCreate(&handle);
	cublasSetPointerMode(handle, CUBLAS_POINTER_MODE_DEVICE);
	// Get time recording ready
	cudaEventCreate(&start);
	cudaEventCreate(&stop);
	cudaEventCreate(&start_matrix);
	cudaEventCreate(&stop_matrix);
	cudaEventCreate(&start_solve);
	cudaEventCreate(&stop_solve);

	//*****************************************************************************************************
*****************************************************************************
	//*****************************************************************************************************
*****************************************************************************
	//
	                                                    SOLVER PREP
	//*****************************************************************************************************
*****************************************************************************
	//*****************************************************************************************************
*****************************************************************************
	printf("\nPreparing the model...\n******USING %i SOLVER******\n", mtype);
	//cin.get();

	// Precalculations for stiffness matrix
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(u_sol_global, total_dof);
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(u_sol_global_prev, total_dof);
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(u_global, total_dof);
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(b_global, total_dof);
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(f_global_prev, total_dof);
	fillones << < GRID_SIZE1, BLOCK_SIZE >> >(f_elements, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)));  // actually filling with zeros

	VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_sol_global, u_sol_elements, d_element_i,
d_type_ind, d_body_ind, total_node_count);

	VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_global, u_elements, d_element_i, d_type_ind,
d_body_ind, total_node_count);
	FibLengths0 << < GRID_SIZE1, BLOCK_SIZE >> >(fib_lengths0_2, d_nodes_x, d_fib_ind, d_body_ind,
total_element_count);

	PreStiffCalculations << < GRID_SIZE1, BLOCK_SIZE >> >(d_element_i, d_type_ind, d_body_ind, d_nodes_x,
solid_inter_constants, solid_element_volumes, solid_BL0, lig_lengths0,
		d_local_coords, d_local_coords0, total_node_count);

	//*****************************If force application was done like a
pressure*********************************
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(r_global, total_dof);
	//ForcePressure << <GRID_SIZE01, BLOCK_SIZE >> >(r_global, d_force_step, d_F_ind, F_node_count,
total_node_count);
	MomentPressure << <GRID_SIZE01, BLOCK_SIZE >> > (r_global, d_nodes_x0, d_F_ind, d_ma_params,
F_node_count, total_node_count);
```

```
		// Calculate stiffness matrix and force vector including preconditioner
		fillzeros << <GRID_SIZE_NON_ZEROS, BLOCK_SIZE >> >(matrix_A, A_non_zeros);

		fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(f_global, total_dof);
		SolidStiffness << < GRID_SIZE1, BLOCK_SIZE >> >(matrix_A, solid_BL0, solid_inter_constants,
solid_element_volumes, d_nodes_x, u_sol_global, u_sol_elements, u_elements, f_elements, f_global,
			d_BC_node_ind, d_element_i, d_type_ind, d_element_j, d_body_ind, d_sec_ind, BC_value,
lig_lengths0, d_fib_ind, fib_lengths0_2, d_local_coords, d_local_coords0, total_node_count);
		cublasScopy(handle, A_non_zeros, matrix_A, 1, matrix_A_prev, 1);

		VectorAssembly << < GRID_SIZE1, BLOCK_SIZE >> >(f_elements, f_global, d_element_i, d_type_ind,
d_body_ind, d_share_element_ind, d_share_bool_ind, total_node_count);

		VectorSubtraction << <GRID_SIZE0, BLOCK_SIZE >> >(r_global, f_global, b_global, total_dof);

		cublasSnrm2(handle, total_dof, b_global, 1, ext_force_norm);
		cudaMemcpy(h_ext_force_norm, ext_force_norm, sizeof(float), cudaMemcpyDeviceToHost);
		cout << "\nRF limit = " << *h_ext_force_norm << endl;
		cublasSnrm2(handle, total_dof, b_global, 1, ext_force_norm);

		ApplyBC << <GRID_SIZE_BC, BLOCK_SIZE >> >(matrix_A, b_global, d_BC_node_ind, d_BCapply_ind,
d_BCapply_stiff, BC_value, BC_node_count, BC_stiff_count, A_non_zeros, total_node_count);

		cudaMemcpy(hf_matrix_A, matrix_A, A_non_zeros * sizeof(float), cudaMemcpyDeviceToHost);
		phase = 11;
		PARDISO(pt, &maxfct, &mnum, &mtype, &phase, &n_total, hf_matrix_A, p_csr_row, p_csr_col, &idum,
&nrhs, iparm, &msglvl, &ddum, &ddum, &pardiso_error);
		if (pardiso_error != 0)
		{
			printf("\nERROR during symbolic factorization: %d\n", pardiso_error);
			goto Error;
		}
		printf("\nReordering completed ... ");
		printf("\nNumber of nonzeros in factors = %d", iparm[17]);
		printf("\nNumber of factorization MFLOPS = %d\n", iparm[18]);

		//*************************************************************************************************
*************************************************************************
		//*************************************************************************************************
*************************************************************************
		//
							SOLVER
		//*************************************************************************************************
*************************************************************************
		//*************************************************************************************************
*************************************************************************

		printf("\nBeginning to solve...\n");
		//cin.get();

		cudaEventRecord(start, 0);

		cudaEventRecord(start_solve, 0);
		cudaMemcpy(hf_b_global, b_global, n_total * sizeof(float), cudaMemcpyDeviceToHost);
		phase = 23;
		PARDISO(pt, &maxfct, &mnum, &mtype, &phase,
			&n_total, hf_matrix_A, p_csr_row, p_csr_col, &idum, &nrhs, iparm, &msglvl, hf_b_global,
hf_u_global, &pardiso_error);
		cudaMemcpy(u_global, hf_u_global, n_total * sizeof(float), cudaMemcpyHostToDevice);
		cudaEventRecord(stop_solve, 0);
```

```
	cudaEventSynchronize(stop_solve);
	cudaEventElapsedTime(&millisecond_solve, start_solve, stop_solve);

	//*********************************************First
iteration*****************************************************************
	cudaEventRecord(start_matrix, 0);

	NR_itr = 1;

	UpdateSolNodes << < GRID_SIZE0, BLOCK_SIZE >> >(d_nodes_x, u_sol_global, u_global, total_dof);

	VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_sol_global, u_sol_elements, d_element_i,
d_type_ind, d_body_ind, total_node_count);
	VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_global, u_elements, d_element_i, d_type_ind,
d_body_ind, total_node_count);

	cublasSnrm2(handle, total_dof, u_global, 1, del_u_norm);
	cudaMemcpy(h_del_u_norm, del_u_norm, sizeof(float), cudaMemcpyDeviceToHost);
	// Calculate stiffness matrix and force vector including preconditioner
	fillzeros << <GRID_SIZE_NON_ZEROS, BLOCK_SIZE >> >(matrix_A, A_non_zeros);
	fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(f_global, total_dof);
	SolidStiffness << < GRID_SIZE1, BLOCK_SIZE >> >(matrix_A, solid_BL0, solid_inter_constants,
solid_element_volumes, d_nodes_x, u_sol_global, u_sol_elements, u_elements, f_elements, f_global,
		d_BC_node_ind, d_element_i, d_type_ind, d_element_j, d_body_ind, d_sec_ind, BC_value,
lig_lengths0, d_fib_ind, fib_lengths0_2, d_local_coords, d_local_coords0, total_node_count);

	fillones << < GRID_SIZE1, BLOCK_SIZE >> >(u_sol_elements, ((total_element_count*tet_element_dof) +
(BLOCK_SIZE*tet_element_dof)));  // actually filling with zeros
	VectorAssembly << < GRID_SIZE1, BLOCK_SIZE >> > (f_elements, f_global, d_element_i, d_type_ind,
d_body_ind, d_share_element_ind, d_share_bool_ind, total_node_count);

	VectorSubtraction << <GRID_SIZE0, BLOCK_SIZE >> >(r_global, f_global, b_global, total_dof);

	ApplyBC << <GRID_SIZE_BC, BLOCK_SIZE >> >(matrix_A, b_global, d_BC_node_ind, d_BCapply_ind,
d_BCapply_stiff, BC_value, BC_node_count, BC_stiff_count, A_non_zeros, total_node_count);

	cublasSnrm2(handle, total_dof, b_global, 1, rf_diff_norm);
	cublasSnrm2(handle, total_dof, u_sol_global, 1, u_norm);
	cudaMemcpy(h_rf_diff_norm, rf_diff_norm, sizeof(float), cudaMemcpyDeviceToHost);
	cudaMemcpy(h_u_norm, u_norm, sizeof(float), cudaMemcpyDeviceToHost);

	cout << "\nU iteration = " << *h_del_u_norm << endl;
	cout << "RF iteration = " << *h_rf_diff_norm << endl;
	if (isnan(*h_del_u_norm) || isnan(*h_rf_diff_norm)) goto Error;

	NR_itr++;

	cudaEventRecord(stop_matrix, 0);
	cudaEventSynchronize(stop_matrix);
	cudaEventElapsedTime(&millisecond_matrix, start_matrix, stop_matrix);
	printf("Elapsed time for matrix build is: %f\n", millisecond_matrix / 1000.0f);
	printf("Elapsed time for matrix solve is: %f\n", millisecond_solve / 1000.0f);

	*h_rf_diff_norm2 = 10000.0f;
	*h_del_u_norm2 = 100000.0f;
	total_itr = 1;
	// Running in two substeps
	while (substep < 1.0f) {
		NR_itr = 1;
		//*****************************************************************Next
iterations*****************************************************************
```

```
                    //while ((((*h_del_u_norm > 20.0f) && (*h_del_u_norm2 > 20.0f)) && ((*h_rf_diff_norm > 110.0f) ||
(*h_rf_diff_norm2 > 125.0f))) && (NR_itr < NR_itr_max)) {
                    //while ((NR_itr < NR_itr_max)) {
                    //while ((((*h_del_u_norm / *h_u_norm) > etol_d) || (*h_rf_diff_norm > h_etol_f)) && (NR_itr <
NR_itr_max)){
                    while (((*h_del_u_norm > 21.0f) || (*h_rf_diff_norm > 200.0f)) && (NR_itr < NR_itr_max)){

                            *h_rf_diff_norm2 = *h_rf_diff_norm;
                            *h_del_u_norm2 = *h_del_u_norm;

                            cudaEventRecord(start_solve, 0);
                            cudaMemcpy(hf_matrix_A, matrix_A, A_non_zeros * sizeof(float),
cudaMemcpyDeviceToHost);
                            cudaMemcpy(hf_b_global, b_global, n_total * sizeof(float), cudaMemcpyDeviceToHost);
                            phase = 23;
                            PARDISO(pt, &maxfct, &mnum, &mtype, &phase,
                                    &n_total, hf_matrix_A, p_csr_row, p_csr_col, &idum, &nrhs, iparm, &msglvl,
hf_b_global, hf_u_global, &pardiso_error);
                            cudaMemcpy(u_global, hf_u_global, n_total * sizeof(float), cudaMemcpyHostToDevice);
                            cudaEventRecord(stop_solve, 0);
                            cudaEventSynchronize(stop_solve);
                            cudaEventElapsedTime(&millisecond_solve, start_solve, stop_solve);

                            //*****************************************************************end K del_u =
R-F ***********************************************************************************
                            cudaEventRecord(start_matrix, 0);

                            UpdateSolNodes << < GRID_SIZE0, BLOCK_SIZE >> >(d_nodes_x, u_sol_global,
u_global, total_dof);
                            fillones << < GRID_SIZE1, BLOCK_SIZE >> >(f_elements,
((total_element_count*tet_element_dof) + (BLOCK_SIZE*tet_element_dof)));  // actually filling with zeros
                            VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_sol_global, u_sol_elements,
d_element_i, d_type_ind, d_body_ind, total_node_count);
                            VectorDisassembly << < GRID_SIZE1, BLOCK_SIZE >> >(u_global, u_elements,
d_element_i, d_type_ind, d_body_ind, total_node_count);

                            cublasSnrm2(handle, total_dof, u_global, 1, del_u_norm);
                            cudaMemcpy(h_del_u_norm, del_u_norm, sizeof(float), cudaMemcpyDeviceToHost);
                            // Calculate stiffness matrix and force vector including preconditioner
                            fillzeros << <GRID_SIZE_NON_ZEROS, BLOCK_SIZE >> >(matrix_A, A_non_zeros);
                            fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(f_global, total_dof);
                            SolidStiffness << < GRID_SIZE1, BLOCK_SIZE >> >(matrix_A, solid_BL0,
solid_inter_constants, solid_element_volumes, d_nodes_x, u_sol_global, u_sol_elements, u_elements, f_elements, f_global,
                                    d_BC_node_ind, d_element_i, d_type_ind, d_element_j, d_body_ind, d_sec_ind,
BC_value, lig_lengths0, d_fib_ind, fib_lengths0_2, d_local_coords, d_local_coords0,
                                    total_node_count);

                            VectorAssembly << < GRID_SIZE1, BLOCK_SIZE >> >(f_elements, f_global, d_element_i,
d_type_ind, d_body_ind, d_share_element_ind, d_share_bool_ind, total_node_count);
                            VectorSubtraction << <GRID_SIZE0, BLOCK_SIZE >> >(r_global, f_global, b_global,
total_dof);

                            ApplyBC << <GRID_SIZE_BC, BLOCK_SIZE >> >(matrix_A, b_global, d_BC_node_ind,
d_BCapply_ind, d_BCapply_stiff, BC_value, BC_node_count, BC_stiff_count, A_non_zeros, total_node_count);

                            cublasSnrm2(handle, total_dof, b_global, 1, rf_diff_norm);
                            cublasSnrm2(handle, total_dof, u_sol_global, 1, u_norm);
                            cudaMemcpy(h_rf_diff_norm, rf_diff_norm, sizeof(float), cudaMemcpyDeviceToHost);
                            cudaMemcpy(h_u_norm, u_norm, sizeof(float), cudaMemcpyDeviceToHost);

                            cout << "\nU iteration = " << *h_del_u_norm << endl;
```

```
                        cout << "RFu iteration = " << *h_rf_diff_norm << endl;

                        NR_itr++;
                        total_itr++;

                        cudaEventRecord(stop_matrix, 0);
                        cudaEventSynchronize(stop_matrix);
                        cudaEventElapsedTime(&millisecond_matrix, start_matrix, stop_matrix);
                        printf("Elapsed time for matrix build is: %f\n", millisecond_matrix / 1000.0f);
                        printf("Elapsed time for matrix solve is: %f\n", millisecond_solve / 1000.0f);
                }

                //if (NR_itr == NR_itr_max) {
                //      cout << "\nMax NR iterations reached" << endl;
                //}

                h_ma_force[2] += h_ma_step[2]; h_ma_force[3] += h_ma_step[3]; h_ma_force[4] += h_ma_step[4];
h_ma_force[5] += h_ma_step[5];
                cout << "*******************SUBSTEP CONVERGED**************" << endl;
                substep += non_substep;
                cudaMemcpy(&x_substeps[step_count*total_dof], d_nodes_x, total_dof*sizeof(float),
cudaMemcpyDeviceToHost);
                step_count++;
                *h_rf_diff_norm = 10000.0f;
                *h_del_u_norm = 10000.0f;
                *h_rf_diff_norm2 = 100000.0f;
                *h_del_u_norm2 = 100000.0f;
                //cudaMemcpy(d_force_step, h_force_step, force_node_dof * sizeof(float),
cudaMemcpyHostToDevice);
                cudaMemcpy(d_ma_params, h_ma_force, 6 * sizeof(float), cudaMemcpyHostToDevice);
                fillzeros << < GRID_SIZE0, BLOCK_SIZE >> >(r_global, total_dof);
                //ForcePressure << <GRID_SIZE01, BLOCK_SIZE >> >(r_global, d_force_step, d_F_ind,
F_node_count, total_node_count);
                MomentPressure << <GRID_SIZE01, BLOCK_SIZE >> > (r_global, d_nodes_x0, d_F_ind,
d_ma_params, F_node_count, total_node_count);
                VectorSubtraction << <GRID_SIZE0, BLOCK_SIZE >> >(r_global, f_global, b_global, total_dof);
                ApplyBC << <GRID_SIZE_BC, BLOCK_SIZE >> >(matrix_A, b_global, d_BC_node_ind,
d_BCapply_ind, d_BCapply_stiff, BC_value, BC_node_count, BC_stiff_count, A_non_zeros, total_node_count);

                cublasSnrm2(handle, total_dof, r_global, 1, ext_force_norm);
                cudaMemcpy(h_ext_force_norm, ext_force_norm, sizeof(float), cudaMemcpyDeviceToHost);
                //cout << "\nRF limit = " << *h_ext_force_norm << endl;
                cublasSnrm2(handle, total_dof, b_global, 1, ext_force_norm);

        }


Error:
        //if (mkl_error != MKL_DSS_SUCCESS) printf("Solver returned pardiso_error code %d\n", mkl_error);
        cudaMemcpy(nodes_x, d_nodes_x, total_dof * sizeof(float), cudaMemcpyDeviceToHost);
        cudaMemcpy(hf_u_global, u_sol_global, total_dof * sizeof(float), cudaMemcpyDeviceToHost);

        // Record time
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&millisecond, start, stop);

        printf("\nTotal number of iterations: %i\n", total_itr);
        printf("Elapsed time is: %f\n", millisecond / 1000.0f);
        //system("pause");
```

```
	// Determine hydrostatic stress in the nucleus pulposus
	HydrostaticStress << <GRID_SIZE1, BLOCK_SIZE >> > (d_h_stress, solid_inter_constants,
solid_element_volumes, u_sol_elements, d_nodes_x, d_element_i, d_type_ind, d_body_ind);
	cudaMemcpy(h_h_stress, d_h_stress, total_element_count * sizeof(float), cudaMemcpyDeviceToHost);

	// F_ELEM file for determining remote node disp and angle
	float* remote_node_after = NULL;
	remote_node_after = (float*)malloc(total_steps*2 * global_dof * sizeof(float));
	float* f_angles = new float[global_dof*F_node_count];
	float n_length = 0.0f;
	float n_dir_before[3] = { 0.0f, 0.0f, 0.0f };
	float n_dir_after[3] = { 0.0f, 0.0f, 0.0f };
	float n_dir_angle[3] = { 0.0f, 0.0f, 0.0f };
	float l_vector[3][3];
	float sum;
	float del;
	float gamma;
	float remote_node_after_coords[global_dof] = { 0.0f, 0.0f, 0.0f };
	for (int step = 0; step < total_steps; step++) {
		for (int j = 0; j < global_dof; j++) {
			remote_node_after_coords[j] = 0.0f;
			for (int i = 0; i < F_node_count; i++) remote_node_after_coords[j] +=
x_substeps[step*total_dof + f_ind[i] * global_dof + j];
		}
		for (int j = 0; j < global_dof; j++) remote_node_after_coords[j] = remote_node_after_coords[j] /
F_node_count;
		float* remote_loc_coord_after = new float[F_node_count*global_dof];
		for (int i = 0; i < F_node_count; i++) {
			for (int j = 0; j < global_dof; j++) remote_loc_coord_after[i*global_dof + j] =
x_substeps[step*total_dof + f_ind[i] * global_dof + j] - remote_node_after_coords[j];
		}
		for (int i = 0; i < global_dof; i++) remote_node_after[step*6 + i] = remote_node_after_coords[i] -
remote_node_before[i]; // Due to translation


                                                // Rotation is calculated from LOCAL COORDINATES NOT
NORMAL VECTORS
		for (int i = 0; i < F_node_count; i++) {
			n_length = sqrtf((remote_loc_coord_before[i*global_dof + 0] *
remote_loc_coord_before[i*global_dof + 0]) + (remote_loc_coord_before[i*global_dof + 1] *
remote_loc_coord_before[i*global_dof + 1])
				+ (remote_loc_coord_before[i*global_dof + 2] *
remote_loc_coord_before[i*global_dof + 2]));
			n_dir_before[0] = remote_loc_coord_before[i*global_dof + 0] / n_length;
			n_dir_before[1] = remote_loc_coord_before[i*global_dof + 1] / n_length;
			n_dir_before[2] = remote_loc_coord_before[i*global_dof + 2] / n_length;
			n_length = sqrtf((remote_loc_coord_after[i*global_dof + 0] *
remote_loc_coord_after[i*global_dof + 0]) + (remote_loc_coord_after[i*global_dof + 1] *
remote_loc_coord_after[i*global_dof + 1])
				+ (remote_loc_coord_after[i*global_dof + 2] *
remote_loc_coord_after[i*global_dof + 2]));
			n_dir_after[0] = remote_loc_coord_after[i*global_dof + 0] / n_length;
			n_dir_after[1] = remote_loc_coord_after[i*global_dof + 1] / n_length;
			n_dir_after[2] = remote_loc_coord_after[i*global_dof + 2] / n_length;
			n_dir_angle[0] = (n_dir_before[1] * n_dir_after[2]) - (n_dir_before[2] * n_dir_after[1]);
			n_dir_angle[1] = (n_dir_before[2] * n_dir_after[0]) - (n_dir_before[0] * n_dir_after[2]);
			n_dir_angle[2] = (n_dir_before[0] * n_dir_after[1]) - (n_dir_before[1] * n_dir_after[0]);
			sum = sqrtf((n_dir_angle[0] * n_dir_angle[0]) + (n_dir_angle[1] * n_dir_angle[1]) +
(n_dir_angle[2] * n_dir_angle[2]));
			del = (n_dir_before[0] * n_dir_after[0]) + (n_dir_before[1] * n_dir_after[1]) +
(n_dir_before[2] * n_dir_after[2]);
```

```
                                gamma = (1.0f - del) / (sum*sum);
                                l_vector[0][0] = 1.0f - (gamma*(n_dir_angle[2] * n_dir_angle[2] + n_dir_angle[1] *
n_dir_angle[1]));
                                l_vector[0][1] = -(n_dir_angle[2]) + (gamma*n_dir_angle[1] * n_dir_angle[0]);
                                l_vector[0][2] = (n_dir_angle[1]) + (gamma*n_dir_angle[0] * n_dir_angle[2]);
                                l_vector[1][0] = (n_dir_angle[2]) + (gamma*n_dir_angle[0] * n_dir_angle[1]);
                                l_vector[1][1] = 1.0f - (gamma*(n_dir_angle[0] * n_dir_angle[0] + n_dir_angle[2] *
n_dir_angle[2]));
                                l_vector[1][2] = -(n_dir_angle[0]) + (gamma*n_dir_angle[1] * n_dir_angle[2]);
                                l_vector[2][0] = -(n_dir_angle[1]) + (gamma*n_dir_angle[0] * n_dir_angle[2]);
                                l_vector[2][1] = (n_dir_angle[0]) + (gamma*n_dir_angle[1] * n_dir_angle[2]);
                                l_vector[2][2] = 1.0f - (gamma*(n_dir_angle[0] * n_dir_angle[0] + n_dir_angle[1] *
n_dir_angle[1]));
                                sum = acosf((l_vector[0][0] + l_vector[1][1] + l_vector[2][2] - 1.0f) / 2.0f);
                                gamma = sqrtf(((l_vector[2][1] - l_vector[1][2])*(l_vector[2][1] - l_vector[1][2])) +
((l_vector[0][2] - l_vector[2][0])*(l_vector[0][2] - l_vector[2][0]))
                                        + ((l_vector[1][0] - l_vector[0][1])*(l_vector[1][0] - l_vector[0][1])));
                                f_angles[i*global_dof + 0] = sum*((l_vector[2][1] - l_vector[1][2]) / gamma);
                                f_angles[i*global_dof + 1] = sum*((l_vector[0][2] - l_vector[2][0]) / gamma);
                                f_angles[i*global_dof + 2] = sum*((l_vector[1][0] - l_vector[0][1]) / gamma);
                        }
                        int max_node[3] = { 0,0,0 };
                        for (int j = 0; j < global_dof; j++) {
                                sum = f_angles[0 * global_dof + j];
                                for (int i = 1; i < F_node_count; i++) {
                                        if (abs(sum) < abs(f_angles[i*global_dof + j])) {
                                                sum = f_angles[i*global_dof + j];
                                                max_node[j] = i; // Find maximum rotation vector (in negative direction).
Max occurs in x-direction
                                        }
                                }
                        }
                        for (int j = 0; j < global_dof; j++) remote_node_after[step*6 + global_dof + j] = f_angles[max_node[0]
* global_dof + j];
                }

                ////Get local coords of slave node and store in l_vector
                //float local_loc[global_dof];
                //float* f_angle_disp = new float[global_dof*F_node_count];
                //float* f_disp = new float[global_dof*F_node_count];
                //sum = sqrtf((remote_node_after[3] * remote_node_after[3]) + (remote_node_after[4] * remote_node_after[4]) +
(remote_node_after[5] * remote_node_after[5]));
                //if (sum != 0.0f) {
                //      del = (sinf(sum*0.5f) / (0.5f*sum));
                //      sum = sinf(sum) / sum;
                //}
                //else {
                //      sum = 0.0f;
                //      del = 0.0f;
                //}
                //del *= del;
                //l_vector[0][0] = 1.0f - (0.5f*del*(remote_node_after[4] * remote_node_after[4] + remote_node_after[5] *
remote_node_after[5]));
                //l_vector[0][1] = -(sum*remote_node_after[5]) + (0.5f*del*remote_node_after[4] * remote_node_after[3]);
                //l_vector[0][2] = (sum*remote_node_after[4]) + (0.5f*del*remote_node_after[3] * remote_node_after[5]);
                //l_vector[1][0] = (sum*remote_node_after[5]) + (0.5f*del*remote_node_after[4] * remote_node_after[3]);
                //l_vector[1][1] = 1.0f - (0.5f*del*(remote_node_after[5] * remote_node_after[5] + remote_node_after[3] *
remote_node_after[3]));
                //l_vector[1][2] = -(sum*remote_node_after[3]) + (0.5f*del*remote_node_after[4] * remote_node_after[5]);
                //l_vector[2][0] = -(sum*remote_node_after[4]) + (0.5f*del*remote_node_after[3] * remote_node_after[5]);
                //l_vector[2][1] = (sum*remote_node_after[3]) + (0.5f*del*remote_node_after[5] * remote_node_after[4]);
```

```
	//l_vector[2][2] = 1.0f - (0.5f*del*(remote_node_after[3] * remote_node_after[3] + remote_node_after[4] *
remote_node_after[4]));
	//int count = 0;
	//for (int i = 0; i < F_node_count; i++) {
	//		for (int k = 0; k < global_dof; k++) {
	//			local_loc[k] = 0.0f;
	//			for (int l = 0; l < global_dof; l++) {
	//				local_loc[k] += l_vector[k][l] * remote_loc_coord_before[i*global_dof + l];
	//			}
	//			f_angle_disp[i*global_dof + k] = local_loc[k] - remote_loc_coord_before[i*global_dof + k];
	//			f_disp[i*global_dof + k] = remote_node_after[k] + f_angle_disp[i*global_dof + k];
	//			sum = f_disp[i*global_dof + k] - hf_u_global[f_ind[i] * global_dof + k];
	//			if (abs(sum) > 0.5f) {
	//				//printf("Angle doesn't work for f_node i = %i and j = %i\n", i, k);
	//				count++;
	//			}
	//		}
	//}
	//printf("Inaccurate count = %i\n", count);

	/* ---------------------------------------------------------------------- */
	/* .. Termination and release of memory for PARDISO. */
	/* ---------------------------------------------------------------------- */
	phase = -1;		/* Release internal memory. */
	PARDISO(pt, &maxfct, &mnum, &mtype, &phase,
		&n_total, &ddum, p_csr_row, p_csr_col, &idum, &nrhs,
		iparm, &msglvl, &ddum, &ddum, &pardiso_error);

	cublasDestroy(handle);

	// cudaFree() each variable if pardiso_error occurs
	cudaFree(d_nodes_x);
	cudaFree(d_nodes_x_prev);
	cudaFree(d_nodes_x0);
	cudaFree(d_element_i);
	cudaFree(d_element_j);
	cudaFree(d_csr_col);
	cudaFree(d_csr_row);
	cudaFree(d_share_element_ind);
	cudaFree(d_share_bool_ind);
	cudaFree(d_type_ind);
	cudaFree(d_body_ind);
	cudaFree(d_sec_ind);
	cudaFree(d_BC_node_ind);
	cudaFree(d_BCapply_ind);
	cudaFree(d_BCapply_stiff);
	cudaFree(d_fib_ind);
	cudaFree(solid_inter_constants);
	cudaFree(solid_BL0);
	cudaFree(solid_element_volumes);
	cudaFree(lig_lengths0);
	cudaFree(matrix_A);
	cudaFree(fib_lengths0_2);
	cudaFree(u_global);
	cudaFree(r_global);
	cudaFree(f_global);
	cudaFree(f_global_prev);
	cudaFree(u_sol_global);
	cudaFree(u_sol_global_prev);
	cudaFree(b_global);
	cudaFree(u_elements);
```

```
cudaFree(u_sol_elements);
cudaFree(f_elements);
cudaFree(u_norm);
cudaFree(rf_diff_norm);
cudaFree(del_u_norm);
cudaFree(ext_force_norm);
cudaFree(d_force);
cudaFree(d_force_step);
cudaFree(d_local_coords0);

//ofstream outfile_sol_orig("u_sol_global6_orig.txt");
ofstream outfile_sol_orig("u_substeps.txt");
if (outfile_sol_orig.is_open()) {
	outfile_sol_orig << " " << endl;
	for (int i = 0; i < total_node_count; i++) {
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << i;
		for (int j = 0; j < global_dof; j++) {
			outfile_sol_orig.width(15);
			outfile_sol_orig << left << hf_u_global[global_dof*i + j];
		}
		outfile_sol_orig << endl;
	}
	for (int k = 0; k < total_steps; k++) {
		outfile_sol_orig << left << "Step = " << k << endl;
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << total_node_count;
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 0];
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 1];
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 2];
		outfile_sol_orig << endl;
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << total_node_count + 1;
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 3];
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 4];
		outfile_sol_orig.width(15);
		outfile_sol_orig << left << remote_node_after[k * 6 + 5];
		outfile_sol_orig << endl;
	}
	outfile_sol_orig.close();
}
else cout << "Unable to create force_node_steps file" << endl;

ofstream outfile_nodes("NODE_LOC_AFTER2.txt");
if (outfile_nodes.is_open()) {
	outfile_nodes << " " << endl;
	for (int i = 0; i < total_node_count; i++) {
		outfile_nodes.width(15);
		outfile_nodes << right << i + 1;
		for (int j = 0; j < global_dof; j++) {
			outfile_nodes.width(15);
			outfile_nodes << right << nodes_x[global_dof*i + j];
		}
		outfile_nodes << endl;
	}
	outfile_nodes.close();
```

```
	}
	else cout << "Unable to create NODES_LOC_AFTER file" << endl;

	ofstream outfile_stress("HYDRO_STRESS.txt");
	if (outfile_stress.is_open()) {
		outfile_stress << " " << endl;
		for (int i = 0; i < total_element_count; i++) {
			outfile_stress.width(15);
			outfile_stress << right << i;
			outfile_stress.width(15);
			outfile_stress << right << h_h_stress[i];
			outfile_stress << endl;
		}
		outfile_stress.close();
	}
	else cout << "Unable to create NODES_LOC_AFTER file" << endl;

	return cudaStatus;
}
```