Comput. Methods Appl. Mech. Engrg. 199 (2010) 3305–3314

Contents lists available at ScienceDirect

Comput. Methods Appl. Mech. Engrg.

journal homepage: www.elsevier.com/locate/cma

Real-time nonlinear finite element computations on GPU – Application to neurosurgical simulation

Grand Roman Joldes *, Adam Wittek, Karol Miller

Intelligent Systems for Medicine Laboratory, School of Mechanical Engineering, The University of Western Australia, Perth, Australia

ARTICLE INFO

Article history:
Received 15 June 2009
Received in revised form 9 April 2010
Accepted 30 June 2010
Available online 30 July 2010

Keywords:
Non-rigid image registration
Biomechanical models
Dynamic relaxation
Graphics processing unit
CUDA

ABSTRACT

Application of biomechanical modeling techniques in the area of medical image analysis and surgical simulation implies two conflicting requirements: accurate results and high solution speeds. Accurate results can be obtained only by using appropriate models and solution algorithms. In our previous papers we have presented algorithms and solution methods for performing accurate nonlinear finite element analysis of brain shift (which includes mixed mesh, different non-linear material models, finite deformations and brain–skull contacts) in less than a minute on a personal computer for models having up to 50,000 degrees of freedom. In this paper we present an implementation of our algorithms on a graphics processing unit (GPU) using the new NVIDIA Compute Unified Device Architecture (CUDA) which leads to more than 20 times increase in the computation speed. This makes possible the use of meshes with more elements, which better represent the geometry, are easier to generate, and provide more accurate results.

1. Introduction

The objective of our work is to significantly increase the efficacy and efficiency of image-guided neurosurgery by including realistic computation of brain deformations, based on a fully non-linear biomechanical model, in a system to improve intra-operative visualisation, navigation and monitoring. The system will create an augmented reality visualisation of the intra-operative configuration of the patient's brain merged with high resolution pre-operative imaging data, including diffusion tensor imaging and functional magnetic resonance imaging, in order to better localise the tumor and critical healthy tissues.

The focus of this paper is on problems arising in image-guided neurosurgery. In this context it is very important to be able to predict the effect of certain procedures on the position of pathologies and critical healthy areas in the brain. The most typical example is the prediction of a displacement field within the brain after opening the skull (so called "brain shift" estimation). A neurosurgeon is interested in the final, deformed position of the brain. The displacement field within the brain predicted by a biomechanical model can be used to morph high quality preoperative images (usually MRI) in order to show the current (intra-operative) position of the brain components. The computation must be done intra-operatively; therefore it is subject to stringent time constraints, which practically means that the results should be available to an operating surgeon in less than one minute [1–4]. The actual deformation computation should take only a fraction of this time, as there are other activities that need to be done beforehand (such as the acquisition of the deformed brain surface in the area of craniotomy and the registration of this surface with its pre-operative position, leading to the extraction of displacements that are used for driving the deformation of the model) [4].

Some of our previous work focused on developing efficient and accurate solution algorithms for nonlinear finite element models. In [5] we have shown how the Total Lagrangian formulation can be combined with explicit integration in order to obtain a very efficient time stepping algorithm. Because of their numerical efficiency, the linear tetrahedron and the linear under-integrated hexahedron are the preferred elements when constructing the mesh. We developed fast algorithms for handling the numerical problems associated with these elements: an anti-hourglassing algorithm for the linear under-integrated hexahedron is presented in [6] and a non-locking linear tetrahedron element is developed in [7]. We also developed an algorithm for handling the brain–skull interaction (contact) in [8]. Combining all these algorithms and using dynamic relaxation for the computation of the steady state solution, we have been able to perform brain shift simulations on standard 3 GHz dual-core personal computer in less than a minute for models having up to 50,000 degrees of freedom [9].

One of the main problems in the development of a biomechanical model is the generation of the mesh. Many algorithms are now available for fast and accurate automatic mesh generation using tetrahedral elements, but not for automatic hexahedral mesh generation [10–12]. Some template based meshing algorithms

* Corresponding author. Tel.: +61 8 6488 3125; fax: +61 8 6488 1024.
E-mail addresses: grandj@mech.uwa.edu.au (G.R. Joldes), adwit@mech.uwa.edu.au (A. Wittek), kmiller@mech.uwa.edu.au (K. Miller).

0045-7825/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.cma.2010.06.037

can be used for meshing different organs using hexahedrons [13–15], but these types of algorithms only work for healthy organs. In case of severe pathologies (such as a brain tumor), such algorithms can not be used, as the shape, size and position of the pathology is unpredictable. This is one reason why many authors proposed the use of tetrahedral meshes for their models [1,2,16–18]. In order to automate the simulation process, mixed meshes (having both hexahedral and tetrahedral elements) with predominantly hexahedral elements are the most convenient.

For the same number of nodes, a tetrahedral mesh has about five times more elements than a hexahedral one. It is therefore desirable to mesh as much of the volume as possible using hexahedral elements, in order to reduce the computational effort. This, combined with a limit on the number of elements in the mesh, usually means manual intervention during the meshing process, which makes meshing difficult and time consuming.

The possibility to use meshes with increased number of elements has several advantages: increased percentage of the volume can be meshed using hexahedral elements, better representation of the geometry, more accurate results, and easier mesh generation. This unfortunately leads to an increased computational time, which means we can no longer satisfy the time constraints associated with surgery using such models. As the algorithms we developed are already very efficient and the CPU does not offer sufficient computation power and memory bandwidth to solve this problem, we turned our attention to parallel computations on GPUs. GPUs offer high computation power and increased memory bandwidth at a relatively low cost.

In the past years there has been an increased interest in using the power of graphics processing units (GPUs), with their parallel architecture, for general purpose computations. The GPU is used as a coprocessor for the CPU for executing sections of the code that can run in parallel. Before the introduction of CUDA, general purpose computations on GPUs (GPGPU) were done by recasting the computations in graphic terms and using the graphics pipeline [19], therefore a scientific or general-purpose computation often requires a concerted effort by experts in both computer graphics and in the particular scientific or engineering domain. With the introduction of CUDA, in November 2006, NVIDIA proposed a new parallel programming model and instruction set for their GPUs that can be used for performing general purpose computations. CUDA comes with a software environment that allows developers to use C as a high level programming language. A minimum set of keywords are used to extent the C language in order to: identify the code that must be run on the GPU as parallel threads, identify each thread (and the block of threads it belongs to) and to organize and transfer the data in the different GPU memory spaces. CUDA also exposes the internal architecture of the GPU and allows direct access to its internal resources. The programmer has more control over the internal hardware resources of the GPU, but this comes at the expense of an increased programming effort compared to a CPU implementation.

An implementation of our basic nonlinear Total Lagrangian Explicit Dynamics algorithm using the GPGPU framework (the graphics pipeline) has been presented in [20]. The implementation shows up to 16 times speed gains compared with the corresponding CPU implementation, but it has several limitations: it can only handle linear locking tetrahedrons, a single material type, and no contacts, has no time step control and does not compute the steady state solution.

In this paper we use CUDA to implement a suite of nonlinear Finite Element algorithms for brain shift computation. The implementation can handle areas with different non-linear materials, different element types (linear hexahedron with hourglass control, linear tetrahedron and non-locking tetrahedron) and contacts between brain and skull. We present CUDA and the related terminology in Section 2. The finite element algorithm, as implemented on CPU, is presented in Section 3. In Section 4 we discuss the way the data-parallel parts of the algorithm can be transferred and executed on a GPU using CUDA in order to improve the computation efficiency. A performance evaluation is done in Section 5 and discussion and conclusions are presented in Section 6.

2. NVIDIA CUDA – a general purpose parallel computing architecture

The detailed presentation of CUDA is available in the Programming Guide provided by NVIDIA [21]. In this section we only present some of the main characteristics and terminology related to CUDA.

The GPU has a highly parallel, multithreaded, many core processor architecture. This is well suited for problems that can be expressed as data-parallel computations with high arithmetic intensity, where the same program is executed on many data elements in parallel. CUDA is a general purpose parallel computing architecture that allows the development of application software that transparently scales with the number of processor cores in the GPU. It achieves this scalability by using three key abstractions – a hierarchy of thread groups, shared memory and barrier synchronization – that are exposed to the programmer as a minimal set of language extensions. These abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel and then into smaller pieces that can be solved cooperatively in parallel.

The code executed on GPU is called a kernel. When a kernel is invoked (from the CPU), it is executed N times in parallel by N different CUDA threads (N is specified during the kernel invocation). Threads are organized into a grid of blocks, with each block identified by a block index, while each thread within the block is identified by a thread index. Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. This is possible because each block of threads runs on the same multiprocessor on the GPU. The number of threads per block is restricted by the limited memory and register resources of a processor core. Thread blocks are required to execute independently, therefore they cannot cooperate among themselves.

The CPU running the program that launches the kernels is called the *host*, while the GPU acts as a coprocessor running the CUDA threads, called the *device*. The host and the device maintain their own DRAM, referred as the host memory and the device memory (global, constant, and texture memory are implemented in the device memory).

CUDA threads may access data from different device memory spaces during their execution: registers, local, shared, global, constant, and texture memory. The properties of each of these memory spaces are presented in Table 1. The register and shared memory are located on chip (on each multiprocessor), allowing very fast access, while the other memory spaces are located on the device memory and have higher latency and lower throughput. The constant and texture memory are read only and cached for faster access.

Table 1
Properties of the different device memory spaces.

Memory	Access	Scope	Lifetime
Register	Read/write	One thread	Thread
Local	Read/write	One thread	Thread
Shared	Read/write	All threads in a block	Block of threads
Global	Read/write	All threads and host	Host allocation
Constant	Read	All threads and host	Host allocation
Texture	Read	All threads and host	Host allocation

Each device has a compute capability identified by a major revision number and a minor revision number. The minor revision number corresponds to incremental improvements to the core architecture, ranging from support for atomic operations in global memory (compute capability 1.1) to support for double-precision floating-point numbers (compute capability 1.3). We performed our computations on a NVIDIA Tesla C870 computing board, which has compute capability 1.0. It has 16 multiprocessors with eight scalar processor cores each and single-precision floating point operations. Therefore, 128 processor cores can run code in parallel and can perform fast data exchanges using the device memory.

The CUDA architecture is built around a scalable array of multi-threaded streaming processors. The threads of a thread block execute concurrently on one multiprocessor, in groups of 32 parallel threads called warps. Individual threads in a warp start together at the same program address but are otherwise free to branch and execute independently. Full efficiency is realized when all threads in a warp agree on their execution path, otherwise the different branches in a warp are executed serially. Because all threads of a block are run on the same multiprocessor they can exchange data using the fast on-chip shared memory. A grid of blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing. The number of blocks a multiprocessor can process at once depends on how many registers per thread and how much shared memory per block are required for a given kernel.

Because of the hardware implementation, there are a series of performance guidelines that must be observed when programming a GPU using CUDA. The most important ones refer to: memory transfers between host and device, memory latency when accessing global and local memory, global memory access pattern (to ensure that memory accesses by threads of a half-warp can be coalesced into a single memory transaction), shared memory access patterns (to avoid memory bank conflicts), and execution configuration (number of threads per block and number of thread blocks specified for a kernel launch). The details on these guidelines are included in the CUDA Programming Guide and should be observed during the software implementation in order to obtain maximum performance.

3. The finite element algorithms for brain shift computation

In this section we present the main algorithms used for solving the finite element problem. This is not an exhaustive presentation, as the details can be found in our previous papers [5–9,22,23].

3.1. Integration of the equations of continuum mechanics

Various spatial discretization schemes are possible while using the finite element method [24]. In the development of our finite element algorithms we used the Total Lagrangian formulation, where all variables are referred to the original configuration of the system. We use Second-Piola Kirchoff stress and Green–Lagrange strains, which are appropriate for handling geometric nonlinearities (finite deformations).

The integration of equilibrium equations in the time domain is done using explicit methods [25]. By using a lumped (diagonal) mass matrix, the equations of motion can be decoupled and no system of equations must be solved. Computations are done at the element level eliminating the need for assembling the stiffness matrix of the entire model.

A detailed description of the Total Lagrange Explicit Dynamics [TLED] algorithm is presented in [5]. The main benefits of the TLED algorithm are:

- allows pre-computing of many variables involved (e.g. derivatives with respect to spatial coordinates, hourglass control parameters),
- no accumulation of errors – increase stability for quasistatic solutions,
- Second-Piola Kirchoff stress and Green strain are used – appropriate for handling geometric non-linearities,
- easy implementation of the material law for hyper-elastic materials using the deformation gradient,
- straightforward treatment of non-linearities,
- no iterations required for a time step,
- no system of equations needs to be solved,
- low computational cost for each time step,
- low internal memory requirements.

3.2. Computation grid: elements used in the finite element mesh

Because of the computation time requirement, the mesh must be constructed using low order elements that are not computationally intensive, such as the linear tetrahedron or the linear under-integrated hexahedron. The standard formulation of the linear tetrahedral element exhibits artificial stiffening, referred to in the literature as volumetric locking [25] when used for incompressible (or almost incompressible) continua such as brain and other soft tissues. To reduce locking special countermeasures must be employed and therefore hexahedral elements are preferred when modeling the behaviour of soft organs.

The under-integrated hexahedral elements require the use of an hourglass control algorithm in order to eliminate the instabilities, known as zero energy modes, which arise from the one-point integration [26]. Special algorithms for handling hourglass control for the hexahedral elements must be implemented.

3.3. Hourglass control

Starting from the algorithm proposed by Flanagan and Belytschko we proved that the Total Lagrangian formulation is also recommended from the point of view of efficient hourglass control implementation, as many quantities involved can be pre-computed. We have shown in [6] that the hourglass control forces for each element can be computed (in matrix form) as:

$$ {}^{t}_{0}\mathbf{F}^{Hg} = k_0 \mathbf{Y}_0 \mathbf{Y}^{T} {}^{t}_{0}\mathbf{u}, \tag{1} $$

where k is a constant that depends on the element geometry and material properties, $\mathbf{Y}$ is the matrix of hourglass shape vectors and $\mathbf{u}$ is the matrix of current displacements. The notation from [25] is used, where the left superscript represents the current time and the left subscript represents the time of the reference configuration, which is 0 for Total Lagrangian. In Eq. (1) all quantities except $\mathbf{u}$ are constant and can be pre-computed, making the hourglass control mechanism very efficient from the computational point of view.

3.4. Non-locking tetrahedral elements

In modeling of incompressible continua, artificial stiffening (often referred to as volumetric locking) afflicts many standard elements including the linear tetrahedral element. This phenomenon occurs also for nearly incompressible materials and therefore introducing slight compressibility does not solve the problem.

A number of improved linear tetrahedral elements with anti-locking features have been proposed by different authors [27–30]. The average nodal pressure (ANP) tetrahedral element proposed in [27] is computationally inexpensive and provides much better results for nearly incompressible materials compared

to the standard tetrahedral element. Nevertheless, one shortcoming of the ANP element and its implementation in a finite element code is the handling of interfaces between different materials. We extended the formulation of the ANP element so that all elements in a mesh are treated in a similar way, requiring no special handling of the interface elements.

The ANP element has the same deviatoric component of the strain energy as the standard tetrahedral element and a modified volumetric component. The modified volumetric component of the strain energy is computed in such a way that the element pressure for an element e is given as the average of the nodal pressures for the nodes belonging to that element:

$$\bar{p}^{(e)} = \frac{1}{4}\sum_{a=1}^{4} p_a^{(e)}. \tag{2}$$

A weakness of the standard ANP element is that it treats different materials separately. Instead of considering different nodal pressure for different material types we make the assumption that the nodal pressure is constant over the nodal volume. This assumption derives from the relation that exists between pressure and stress ($p = -\sigma_{ii}/3$) [31] and from the fact that at the interface between two different materials the stress in the materials should be the same. Starting from this assumption, we demonstrated in [7] that the nodal pressure should be computed as:

$$p_a = \frac{\sum_{e=1}^{m_a} p^{(e)} V^{(e)}}{\sum_{e=1}^{m_a} V^{(e)}}, \tag{3}$$

where m_a is the number of elements surrounding node a. The element pressure is computed afterwards in the same manner as for the standard ANP element, using (2). We will call this element the improved average nodal pressure (IANP) tetrahedral element.

3.5. Computation of nodal forces

Because we use single-point integration for all the elements in the mesh (linear tetrahedrons and under-integrated linear hexahedrons) for computational efficiency, the nodal forces for each element are computed as:

$${}^{t}_{0}\mathbf{P}_{\text{int}} = {}^{t}_{0}\mathbf{X} \cdot {}^{t}_{0}\mathbf{S} \cdot \mathbf{B}_0 \cdot V_0, \tag{4}$$

where $\mathbf{P}_{\text{int}}$ is the matrix of nodal forces, $\mathbf{B}_0$ is the matrix of shape function derivatives, $\mathbf{S}$ is the Second Piola Kirchoff stress matrix, $\mathbf{X}$ is the deformation gradient and V_0 is the initial volume. The matrix of shape function derivatives $\mathbf{B}_0$ and the initial volume V_0 are constant and therefore are pre-computed. The stress matrix is computed based on the deformation gradient, using the considered material law (we use the Neo-Hookean material law in our computations).

3.6. Modeling of interactions between different organs: contact algorithm

Many simulations require the treatment of interactions between different parts of the model. In order to handle the brain–skull interaction we developed a very efficient algorithm that treats this interaction as a finite sliding, frictionless contact between a deformable object (the brain) and a rigid surface (the skull). The contact type was chosen based on the anatomical properties of the brain–skull interface.

The main parts of the contact algorithm are: detection of nodes on the brain surface (also called the slave surface) which have penetrated the skull surface (master surface) and the displacement of each slave node that has penetrated the master surface to the closest point on the master surface.

The surfaces of the anatomical structures of segmented brain images are typically discretised using triangles; therefore we consider the skull surface as a triangular mesh. We will call each triangle surface a "face", the vertices – "nodes" and the triangle sides – "edges".

We base our penetration detection algorithm on the closest master node (nearest neighbour) approach [32]. The basic algorithm is as follows:

- For each slave node P:
  - Find the closest master node C (global search).
  - Check the faces and edges surrounding C for penetration (local search).
  - Check additional faces and edges that might be penetrated by P (identified in the master surface analysis stage – because the master surface is rigid this analysis can be done pre-operatively).

To improve the computation speed, following [32], we implemented the global search phase using bucket sort. A detailed description of this searching algorithm is given in [33].

3.7. Dynamic relaxation

The basic dynamic relaxation (DR) algorithm is presented in [34]. The main ideas are the inclusion of a mass proportional damping in the equation of motion that will increase the convergence speed towards the deformed state and the solving of the obtained damped equation using the central difference method (explicit integration).

After including the derivatives defined by the central difference method in the damped equation of motion, the equation that describes the iterations in terms of displacements becomes:

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \beta(\mathbf{q}^n - \mathbf{q}^{n-1}) + \alpha\mathbf{M}^{-1}(\mathbf{f} - \mathbf{P}(\mathbf{q}^n)), \tag{5}$$

$$\alpha = 2h^2/(2 + ch), \quad \beta = (2 - ch)/(2 + ch), \tag{6}$$

where h is a fixed time increment, n indicates the nth time increment, c is the damping coefficient, $\mathbf{M}$ is the mass matrix, $\mathbf{q}$ is the displacement vector, $\mathbf{P}$ is the vector of internal nodal forces and $\mathbf{f}$ is the vector of externally applied forces (volumetric forces, surface forces, nodal forces as well as forces derived from contacts).

The iterative method defined by Eq. (5) is explicit as long as the mass matrix is diagonal. As the mass matrix does not influence the deformed state solution, a lumped mass matrix can be used that maximizes the convergence of the method.

One very important aspect of any FEM algorithm is the termination criterion used. If the criterion is too coarse, then the solution might be too inaccurate and if the criterion is too tight, then time is lost in unnecessary computations. In [23] we propose a new termination criterion that gives information about the absolute error in the solution, particularly suited for our solution method

$$\|\mathbf{q}^{n+1} - \mathbf{q}^*\|_\infty \leqslant \frac{\rho}{1-\rho}\|\mathbf{q}^{n+1} - \mathbf{q}^n\|_\infty \leqslant \delta, \tag{7}$$

where ρ is the spectral radius of a matrix representing the reduction in error and δ is the imposed accuracy. This convergence criterion gives an approximation of the absolute error based on the displacement variation norm from the current iteration.

The parameters ρ, c and h are computed to maximize the rate of convergence to the steady state solution. The algorithms for computing these parameters are presented in [23].

3.8. The complete algorithm

The algorithm we propose for computing the deformed state using DR can be summarized as follows:

(a) Initialization:
- $\mathbf{q}^0 = 0$; $\mathbf{q}^1 = 0$.
- Compute parameters ρ, c, h and the mass matrix $\mathbf{M}$ that provide maximum convergence rate.
- Pre-compute all other needed quantities (shape functions, hourglass shape vectors, initial volumes, etc.).

(b) For every iteration step:
(1) Apply current loading (nodal displacements, external forces).
(2) Compute the nodal forces corresponding to the current displacements, $\mathbf{P}(\mathbf{q}^n)$:
- For each non-locking tetra in the mesh:
  - Compute the element pressure.
- For each node in the mesh:
  - Compute the nodal pressure (corresponding to the non-locking tetra elements) using (3).
- For each element in the mesh:
  - Compute the deformation gradient.
  - If the element is a non-locking tetra, compute the average element pressure using (2) and modify the deformation gradient.
  - Compute the nodal forces using (4).
  - If the element is an under-integrated hexahedron, add the hourglass control forces given by (1).
  - Add the nodal forces to the global nodal force vector $\mathbf{P}$.

(3) For each node in the mesh:
- Compute next displacement vector using (5).

(4) For each slave node:
- Check for penetration of master surface and move the node if penetration is detected.

(5) Compute displacement variation norm, check convergence criteria and finish analysis if met, using (7).

(6) For each element in the mesh:
- Check the maximum eigenvalue and change the mass matrix if needed (mass scaling, to ensure convergence).

The initialization step can be done pre-operatively and therefore we are interested in pre-computing as many parameters as possible in this step. Because the iterative solution is obtained intra-operatively, the computation process must be very efficient.

4. GPU implementation using CUDA

When transforming an application to run on a GPU, first we have to identify the data-parallel parts of it. In our case, we are interested in speeding the iteration process (part b of the program), which has to be performed intra-operatively. The most obvious data-parallel parts of this code are the ones described above using phrases such as "for each element: ..." or "for each node: ...", as each element or node can be seen as a data structure on which the computations are made. Therefore, the first kernels (code that should be run on the GPU) we identify are presented in Table 2.

Table 2
Identified kernels and the data they operate on.

Kernel's function	Data structure to operate on
Compute element pressure	Non-locking tetra elements
Compute nodal pressure (Eq. (3))	Mesh nodes
Compute nodal forces (Eq. (4))	Non-locking tetra elements
Compute nodal forces (Eq. (4))	Linear tetra element
Compute nodal forces (Eqs. (1) and (4))	Under-integrated hexahedral element
Compute new displacements (Eq. (5))	Mesh nodes
Enforce contacts (Section 3.6)	Slave nodes

As discussed in Section 2, there are some performance guidelines that must be considered when transferring the code on the GPU. One very important aspect is to minimize the data transfer between host and device, as these transfers are relatively slow. A second aspect is to minimize the number of kernels that are used, as each kernel invocation implies a small hardware and software overhead. Taking this into consideration, we took the following measures to increase the speed of the implementation:

- Re-organize portions of the code: the computation of maximum eigenvalue (step 6), that is done for each element in the mesh, is done differently for each element type. We moved this code, for each different element type, at the beginning of the kernels that compute the nodal forces. This way we eliminated the need for another three kernel invocations in step 6.
- Minimize data transfers between host and device:
  - Application of loads (step 1), which is done for each (loaded) node, has been moved to the end of the kernel that computes the new displacements. The loads applied are saved on the device memory, and only a parameter specifying the percentage of the load that should be applied in the next step is passed as a parameter to the kernel.
  - The computation of the displacement variation norm (step 5) is done on the GPU. This avoids the transfer of the complete nodal displacements vector to the CPU at each time step, as only the final norm needs to be transferred. This computation is, in fact, a reduction problem, and can not be solved using just one kernel invocation (because the thread blocks can not synchronize their execution). Therefore, in our implementation, the reduction is started in the kernel that computes the new displacements, where a binary tree reduction is done in each block of threads and the partial results are saved in memory. The final step of the reduction is done in the first block of a kernel that computes the nodal forces, where sequential and parallel reduction are combined to obtain the final result. The guidelines from the NVIDIA paper "Optimizing Parallel Reduction in CUDA", which comes with the CUDA development kit, were followed when implementing the reduction algorithm.
  - All the information needed for the computations is transferred to the GPU in the initialization stage (pre-operatively).

Another point that must be considered when transferring code on the GPU is that scatter operations can not be safely done on the GPU (because of the parallel nature of the execution, two threads might try to write the same memory location). Unless the GPU supports atomic read-modify-write operations, all scatter operations must be avoided. Such operations are needed in our code during the assembly of the nodal force vector or the mass vector. The solution to this problem is to transform the scatter operations into gather operations. For example, in case of the nodal force vector computation, the nodal forces computed for each element are saved in the device memory and the assembly of the force vector is done, for each node, in the kernel that computes the new displacements.

An important difference between the CPU and GPU code is data organization. While on a CPU the data is usually grouped together in structures or classes, in the GPU memory the data must be re-organized in order to obtain maximum memory access performances. The data organization and alignment must ensure coalesced memory access as much as possible, especially for memory writes. During memory reads, whenever coalesced access is not possible (for example in case of gather operations) the memory is accessed using textures.

The complete algorithm for computing the steady state solution using the GPU looks as follows:

(a) Initialization:
- $\mathbf{q}^0 = 0$; $\mathbf{q}^1 = 0$.
- Compute parameters ρ, c, h and the mass matrix $\mathbf{M}$ that provide maximum convergence rate.
- Pre-compute all other needed quantities (shape functions, hourglass shape vectors, initial volumes, etc.).
- Transfer all needed data to the GPU memory.

(b) For every iteration step:
  (1) Compute the nodal forces corresponding to the current displacements, $\mathbf{P}(\mathbf{q}^n)$:
    - Run the kernel that computes the element pressure.
    - Run the kernel that computes the nodal pressure.
    - For each element type:
      - Run the kernel that computes the nodal forces and saves them in the GPU memory. These kernels also check the maximum eigenvalue and change the mass of the element if needed (the mass is saved in the GPU memory). One of the kernels finalizes the reduction of the displacement variation norm.
  (2) Read the displacement variation norm from the GPU.
  (3) Compute next displacement vector:
    - Run the kernel that computes the next displacements. This kernel assembles the force vector and mass matrix, computes the new displacements using (5) and applies the load for the next time step. It also computes the displacement variation norm and performs its reduction at block level, saving the intermediate results in the GPU memory.
  (4) Run the kernel that enforces the contacts.
  (5) Check convergence criteria and finish analysis if met, using (7).

(c) Read final displacements from the GPU.

(d) Clean up GPU memory.

Our implementation allows the use of different materials for different parts of the brain. Nevertheless, in our application, we use only one material law (Neo-Hookean). Therefore, the use of different materials is implemented quite easily by saving the different material coefficients in the constant device memory and accessing them for each element based on an index that identifies the material associated with that element.

When choosing the execution configuration, based on the advice given in the programming guide, the number of threads per block should be a multiple of 64. This number is limited by the resources available on the GPU (registers, shared memory) and should allow enough registers for each thread so that no variables are placed in the local memory (because of the high latency of local memory access).

5. Performance evaluation

We evaluated the performance of the GPU implementation by performing several simulations and comparing the results with our best CPU implementation of the same algorithms. The purpose of first three simulations is to evaluate the different parts of the algorithm specific to different element types. In these simulations we deformed a cylinder meshed with a different element type for each simulation and compared the speed and accuracy of the results against the CPU computations. We used a soft, almost incompressible Neo-Hookean material model for all simulations. The applied displacements were 20% of the undeformed height of the cylinder. We performed 3000 time steps for each simulation. The results are presented in Table 3.

From our experiments we can conclude that the computations performed using the GPU can be orders of magnitude faster than the ones on CPU. On the CPU, the computation time varies almost linearly with the number of elements. The relative GPU performance increases with the number of elements. This happens because the computation time, when the GPU is used, can be split into two components: an almost constant component, given by the operations performed on the CPU (program control, kernel invocations, data transfer between host and device) and a variable component (computations performed on the GPU – varies almost linearly with the number of elements). When a low number of elements are used, the constant component represents a significant part of the computation time, degrading the relative performance.

The GPU we used supports only single-precision floating point numbers. The latest generation of GPUs from NVIDIA has support for double-precision floating point numbers, but the operations in double precision are around eight times slower than the operations in single precision, and the memory transfers are two times slower. Because we use the Total Lagrangian formulation, there is no accumulation of errors taking place from one time step to another. This is one very important advantage over the Updated Lagrangian formulation, and it allows us to use single precision when performing the computations. In order to assess how the precision of floating point operations impacts on the convergence of the algorithm, we counted the number of steps needed to reach a given accuracy (imposed using the convergence criteria) on both GPU and CPU. From the results (Table 4) we can see that performing the operations in single precision does not have a high impact on convergence for the accuracy usually required in our simulations (we used an imposed accuracy $\delta = 1.0\text{E}{-4}$). Nevertheless, if the imposed accuracy is chosen too small, the computation error could become a significant part of the displacement variation norm evaluation (Eq. 7), and the convergence criteria might never be satisfied. The maximum difference in nodal displacements between GPU and CPU results was 1.41E−4, which is within the limits set by the imposed accuracy (it should be less than 1.7δ). The repartition of the displacement difference over the mesh is presented in Fig. 1.

In the next experiment we performed a brain shift simulation using a biomechanical model. A human brain consisting of healthy brain tissue, a tumor and ventricles is enclosed inside the skull (Fig. 2). The different parts of the brain (parenchyma, tumor, and ventricle) are modeled using almost incompressible nonlinear materials (Neo-Hookean) with different properties, as presented in Table 5 (for the ventricle the Poisson's ratio was assumed to

Table 3
Computation times for GPU and CPU for different element types.

Deformation	No. of elements	Type of elements	Computation time (s)		Speed up (x)
			CPU	GPU	
Compression	48,000	Hexahedron	172	5.3	32.4
Extension	290,000	IANP tetra	1769	17.1	103.4
Shear	290,000	Linear tetra	1614	12.6	128

Table 4
Number of steps required for convergence on GPU and CPU for an imposed accuracy ($\delta = 1.0\text{E}{-4}$).

Deformation	No. of elements	Type of elements	No. of steps		Difference in displacements (max/mean)
			CPU	GPU	
Compression	48,000	Hexahedron	1457	1250	1.28E−5/6.85E−6
Extension	290,000	ANP tetra	3276	3257	1.41E−4/7.8E−5
Shear	290,000	Linear tetra	1908	1910	3.03E−5/1.8E−5

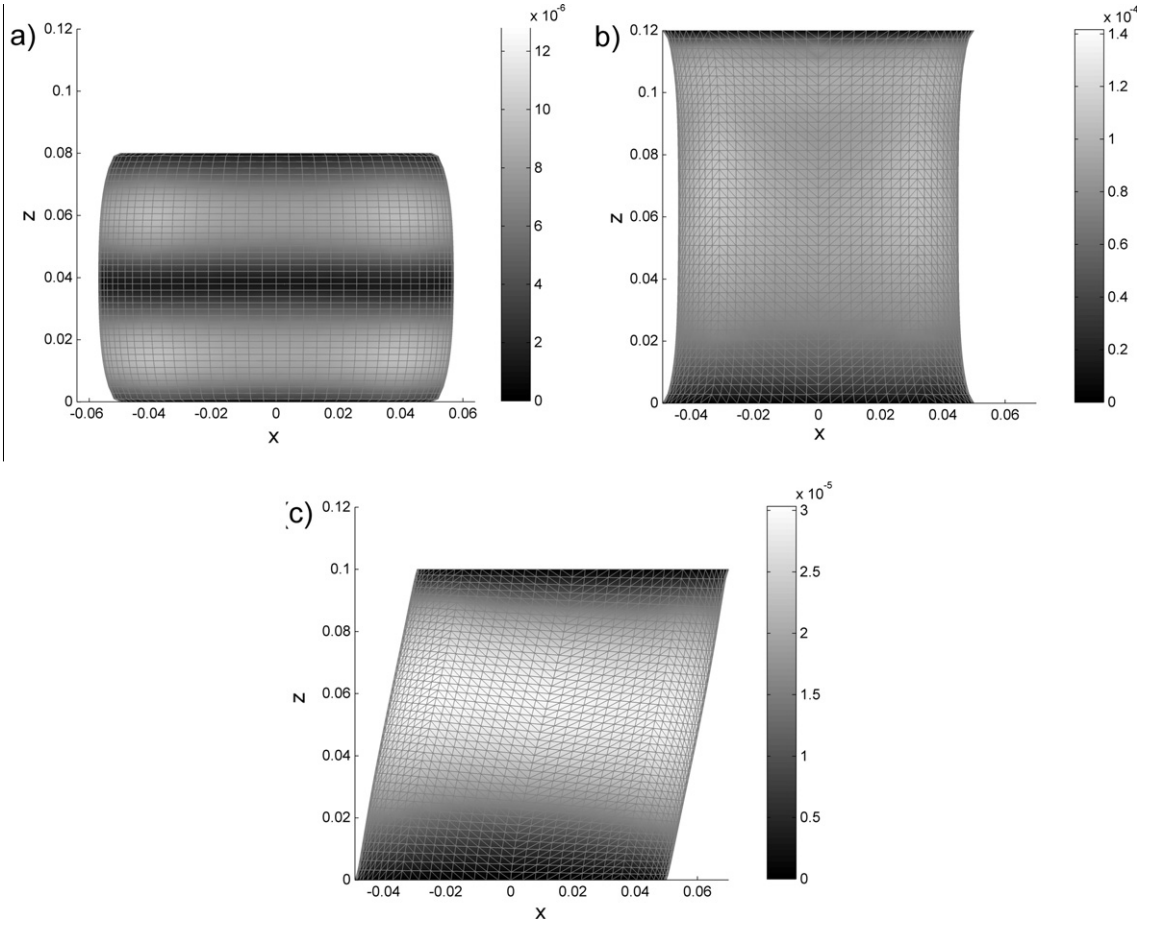

Fig. 1. Deformation of a cylinder: (a) compression; (b) extension; (c) shear. The color codes represent the difference in nodal displacements between the GPU and CPU results for an imposed accuracy δ = 1.0E−4. All dimensions are in meters.

be low in order to account for any fluid leakage). The material model is consistent with our previous work on brain material properties [35–37].

The brain is meshed using all three types of elements we considered (hexahedral elements, linear tetrahedral and improved tetrahedral elements). The hexahedral and tetrahedral elements in the mesh share the same nodes at the interface. There are no wedges or pyramids used as a transition layer; therefore the resulting mesh is non-conforming (the approximation fields are not always continuous between elements of different type). We also performed the simulations on a refined mesh to assess the performance of the algorithms on meshes with higher number of elements. The structure of these two meshes is presented in Table 6. We assume that the initial geometry is known from high quality pre-operative MRI images and we simulate the brain shift by applying displacements on the area of the brain visible during craniotomy, where the displacements can be measured intra-operatively (using a laser range scanner [38] or a stereo vision system [39]). We extracted the required displacements from available intra-operative MRI images. The maximum applied displacement was 10 mm. A very similar model has been used in previous papers for brain shift estimation [4,40,41].

We assume the skull to be rigid and the interaction between the brain and the skull as a frictionless finite sliding contact. All external brain nodes except the displaced ones are included in the contact definition.

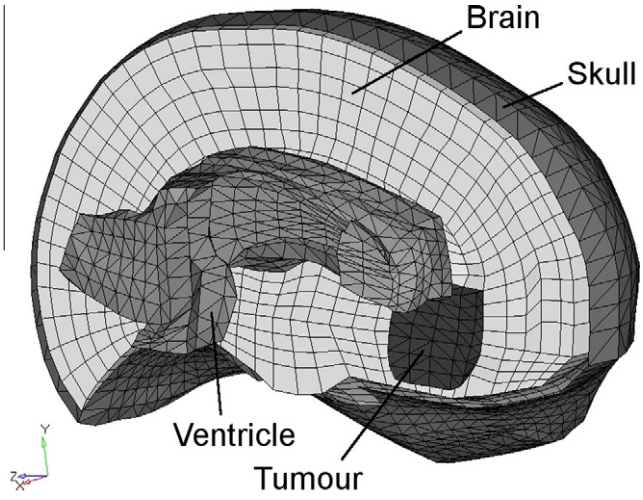

Fig. 2. The mesh used for the brain shift simulation (only the left half of the brain and skull meshes is shown).

Table 5
Material properties for different parts of the brain.

Brain part	Density (kg/m^3)	Young's modulus (Pa)	Poisson's ratio
Brain	1000	2500	0.49
Tumor	1000	7500	0.49
Ventricle	1000	100	0.1

Table 6
Structure of the meshes used.

Mesh	Number of nodes	Number of elements				Skull	
		Hexa	Linear tetras	ANP tetras	Total elements	Number of nodes	Number of tria.
Original	**12,693**	10,596	4831	1398	**16,825**	1993	**3960**
Refined	**95,669**	84,768	32,439	8085	**125,292**	7945	**15,840**

The data in bold highlights the overall size of the meshes used in the model.

Table 7
Computation results for brain shift simulation.

Mesh	No. of steps required for convergence (δ = 1.0E−4)		Run time for 3000 steps (s)		Speed up (x)	Difference in displacements (max/mean)
	CPU	GPU	CPU	GPU		
Original	1887	2103	79.7	3.54	22.5	1.04E−4/8.98E−6
Refined	3120	3091	543.4	19.95	27.2	4.56E−5/2.02E−5

The computation results for the CPU and GPU simulations are presented in Table 7. The simulations need around 2000 steps for the original mesh and 3000 steps for the refined mesh to reach the steady state with sub-millimeter accuracy (δ = 1.0E−4 m). This shows that the convergence rate of the solution decreases with the decrease of the element size, which is a characteristic of the dynamic relaxation solution method. This is one more reason why the computations can no longer be performed in real time on a CPU for an increased number of elements.

The speed-ups increase with the number of elements in the mesh, but are not as large as in the case of cylinder deformation experiments. This happens because the combination of three types of elements in the same mesh, plus contacts, leads to a higher number of kernel invocations at each time step, which increases the constant component of the computation time on GPU. The difference between the CPU and the GPU in the nodal displacement values and the number of steps required for convergence shows that using single-precision floating-point numbers on GPU does not have a significant impact on the results.

In this paper we do not wish to make a validation of the model used, but to demonstrate the efficiency of our algorithms. Partial validation of the model was performed in previous papers with very promising results [4,40,41]. A typical comparison between the deformation computed using our algorithms, the deformation computed using a commercial software package (LS-Dyna), and the deformation extracted from the intra-operative MRI is presented in Fig. 3. The presented algorithms can, in principle, be used with any other finite element model that requires the computation of a steady state solution.

6. Discussion and conclusions

Intra-operative computation of the brain shift using bio-mechanical models can be used to register high-quality pre-operative images to the intra-operative position. This can be considered as a finite element problem for which the steady state solution needs to be found. We have developed algorithms and solution methods that allow us to use complex bio-mechanical models

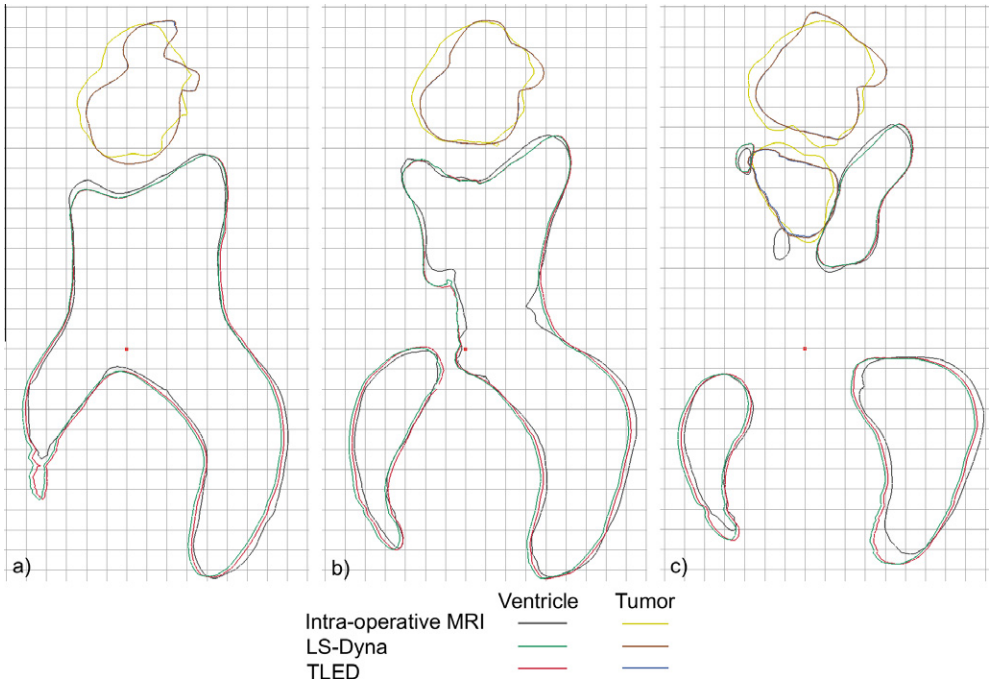

Fig. 3. Brain shift simulation – comparison between the simulation results and the intra-operative MRI. The cutting sections are perpendicular to the superiorly pointing axis, with 0 on the brain's most superior vertex, at distances of (a) −45.5 mm, (b) −50.5 mm and (c) −55.5 mm. Grid lines are 5 mm apart.

(which include geometric and material nonlinearities, finite deformations and contacts) to solve such a problem. These methods allow us to find a solution within the intra-operative constraints of surgery (less than a minute) for models having up to 50,000 degrees of freedom on a personal computer.

GPUs offer high parallel computing power at a low cost. With the release of CUDA, developing general purpose computations on GPU has become much easier. As the internal organization of the GPU is revealed, the programmer can take full advantage of the GPU's computational power. We use CUDA to transfer the data-parallel parts of our computations to the GPU. Our implementation allows us to perform computations more than 20 times faster than on the CPU using a GPU with compute capability 1.0 (having 16 multiprocessors). We can now use models with an increased number of elements to solve brain shift problems within the time constrain of the operating room. This also simplifies the task of meshing such models, as the limitation on the number of elements in the model is almost removed.

The use of single-precision floating-point numbers on the GPU does not have a significant impact on the convergence and accuracy of our solution method. This happens because we use the Total Lagrangian formulation, which does not exhibit accumulation of errors during the time stepping procedure.

The evolution of GPU hardware allows these performances to be enhanced even more. The latest devices, with compute capability 1.3, have more multiprocessors (30 on Tesla C1060), double the number of registers, support for double-precision floating-point numbers and concurrent kernel execution with memory transfers between host and device. These devices are also easier to program, as some of the restrictions on the coalescing of memory transfers have been removed.

Although we implemented only linear hexahedral and tetrahedral elements on the GPU, the code can be extended to handle any other type of finite elements: with a different number of nodes, higher order shape function or higher number of integration points. It can also be extended to handle mesh-free methods, such as the one presented in [42] (integration cells can be processed in parallel instead of finite elements).

Acknowledgments

The financial support of the Australian Research Council (Grants DP0664534, DP0770275 and LX0774754), University of Western Australia (Research Development Award 2009), and National Institute of Health (Grant R03 CA126466) is gratefully acknowledged.

References

[1] M. Ferrant, A. Nabavi, B. Macq, P.M. Black, F.A. Jolesz, R. Kikinis, S.K. Warfield, Serial registration of intraoperative MR images of the brain, Med. Image Anal. 6 (4) (2002) 337–359.
[2] S.K. Warfield, F. Talos, A. Tei, A. Bharatha, A. Nabavi, M. Ferrant, P.M. Black, F.A. Jolesz, R. Kikinis, Real-time registration of volumetric brain MRI by biomechanical simulation of deformation during image guided surgery, Comput. Visual. Sci. 5 (2002) 3–11.
[3] S.K. Warfield, S.J. Haker, I.-F. Talos, C.A. Kemper, N. Weisenfeld, U.J. Mewes, D. Goldberg-Zimring, K.H. Zou, C.-F. Westin, W.M. Wells, C.M.C. Tempany, A. Golby, P.M. Black, F.A. Jolesz, R. Kikinis, Capturing intraoperative deformations: research experience at Brigham and Womens's hospital, Med. Image Anal. 9 (2) (2005) 145–162.
[4] A. Wittek, K. Miller, R. Kikinis, S.K. Warfield, Patient-specific model of brain deformation: application to medical image registration, J. Biomech. 40 (2007) 919–929.
[5] K. Miller, G.R. Joldes, D. Lance, A. Wittek, Total Lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation, Commun. Numer. Methods Engrg. 23 (2007) 121–134.
[6] G.R. Joldes, A. Wittek, K. Miller, An efficient hourglass control implementation for the uniform strain hexahedron using the total Lagrangian formulation, Commun. Numer. Methods Engrg. 24 (2008) 1315–1323.
[7] G.R. Joldes, A. Wittek, K. Miller, Non-locking tetrahedral finite element for surgical simulation, Commun. Numer. Methods Engrg. 25 (7) (2009) 827–836.
[8] G.R. Joldes, A. Wittek, K. Miller, L. Morriss, Realistic and efficient brain–skull interaction model for brain shift computation, in: Computational Biomechanics for Medicine III Workshop, MICCAI, New York, 2008.
[9] G.R. Joldes, A. Wittek, K. Miller, Suite of finite element algorithms for accurate computation of soft tissue deformation for surgical simulation, Med. Image Anal. 13 (6) (2009) 912–919.
[10] S.J. Owen, A survey of unstructured mesh generation technology, in: Seventh International Meshing Roundtable, Dearborn, Michigan, USA, 1998.
[11] M. Viceconti, F. Taddei, Automatic generation of finite element meshes from computed tomography data, Crit. Rev. Biomed. Engrg. 31 (1) (2003) 27–72.
[12] S.J. Owen, Hex-dominant mesh generation using 3D constrained triangulation, Comput.-Aided Design 33 (2001) 211–220.
[13] A.D. Castellano-Smith, T. Hartkens, J. Schnabel, D.R. Hose, H. Liu, W.A. Hall, C.L. Truwit, D.J. Hawkens, D.L.G. Hill, Constructing patient specific models for correcting intraoperative brain deformation, in: Fourth International Conference on Medical Image Computing and Computer Assisted Intervention, MICCAI, Utrecht, The Netherlands, 2001.
[14] B. Couteau, Y. Payan, S. Lavallée, The mesh-matching algorithm: an automatic 3D mesh generator for finite element structures, J. Biomech. 33 (2000) 1005–1009.
[15] V. Luboz, M. Chabanas, P. Swider, Y. Payan, Orbital and maxillofacial computer aided surgery: patient-specific finite element models to predict surgical outcomes, Comput. Methods Biomech. Biomed. Engrg. 8 (4) (2005) 259–265.
[16] M. Ferrant, B. Macq, A. Nabavi, S.K. Warfield, Deformable modeling for characterizing biomedical shape changes, in: Discrete Geometry for Computer Imagery: Ninth International Conference, Springer-Verlag, Uppsala, Sweden, 2000.
[17] O. Clatz, H. Delingette, E. Bardinet, D. Dormont, N. Ayache, Patient specific biomechanical model of the brain: application to Parkinson's disease procedure, in: International Symposium on Surgery Simulation and Soft Tissue Modeling (IS4TM'03), Springer-Verlag, Juan-les-Pins, France, 2003.
[18] O. Clatz, M. Sermesant, P.-Y. Bondiau, H. Delingette, S.K. Warfield, G. Malandain, N. Ayache, Realistic simulation of the 3D growth of brain tumors in MR images coupling diffusion with biomechanical deformation, IEEE Trans. Med. Imag. 24 (10) (2005) 1334–1346.
[19] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Comput. Graph. Forum 26 (1) (2007) 80–113.
[20] Z.A. Taylor, M. Cheng, S. Ourselin, High-speed nonlinear finite element analysis for surgical simulation using graphics processing units, IEEE Trans. Med. Imag. 27 (5) (2008) 650–663.
[21] NVIDIA CUDA – Programming Guide, Version 2.1, NVIDIA Corporation, 2008.
[22] K. Miller, A. Wittek, G. Joldes, A. Horton, T.D. Roy, J. Berger, L. Morriss, Modelling brain deformations for computer-integrated neurosurgery, Commun. Numer. Methods Engrg. 26 (1) (2009) 117–138.
[23] G.R. Joldes, A. Wittek, K. Miller, Computation of intra-operative brain shift using dynamic relaxation, Comput. Methods Appl. Mech. Engrg. 198 (41–44) (2009) 3313–3320.
[24] T. Belytschko, An overview of semidiscretization and time integration procedures, in: T. Belytschko, T.J.R. Hughes (Eds.), Computational Methods for Transient Analysis, North-Holland, Amsterdam, 1983, pp. 1–66.
[25] K.-J. Bathe, Finite Element Procedures, Prentice-Hall, New Jersey, 1996.
[26] D.P. Flanagan, T. Belytschko, A uniform strain hexahedron and quadrilateral with orthogonal hourglass control, Int. J. Numer. Methods Engrg. 17 (1981) 679–706.
[27] J. Bonet, A.J. Burton, A simple averaged nodal pressure tetrahedral element for incompressible and nearly incompressible dynamic explicit applications, Commun. Numer. Methods Engrg. 14 (1998) 437–449.
[28] J. Bonet, H. Marriott, O. Hassan, An averaged nodal deformation gradient linear tetrahedral element for large strain explicit dynamic applications, Commun. Numer. Methods Engrg. 17 (2001) 551–561.
[29] O.C. Zienkiewicz, J. Rojek, R.L. Taylor, M. Pastor, Triangles and tetrahedra in explicit dynamic codes for solids, Int. J. Numer. Methods Engrg. 43 (1998) 565–583.
[30] C.R. Dohrmann, M.W. Heinstein, J. Jung, S.W. Key, W.R. Witkowski, Node-based uniform strain elements for three-node triangular and four-node tetrahedral meshes, Int. J. Numer. Methods Engrg. 47 (2000) 1549–1568.
[31] T.J.R. Hughes, The Finite Element Method: Linear Static and Dynamic Finite Element Analysis, Dover Publications, Mineola, 2000. p. 682.
[32] J.O. Hallquist, LS-DYNA Theory Manual, Livermore Software Technology Corporation, Livermore, California, 2005.
[33] R.G. Sauve, G.D. Morandin, Simulation of contact in finite deformation problems – algorithm and modelling issues, Int. J. Mech. Mater. Design 1 (2004) 287–316.
[34] P. Underwood, Dynamic relaxation, in: T. Belytschko, T.J.R. Hughes (Eds.), Computational Methods for Transient Analysis, New-Holland, Amsterdam, 1983, pp. 245–265.
[35] K. Miller, K. Chinzei, Constitutive modelling of brain tissue: experiment and theory, J. Biomech. 30 (11/12) (1997) 1115–1121.
[36] K. Miller, K. Chinzei, G. Orssengo, P. Bednarz, Mechanical properties of brain tissue in-vivo: experiment and computer simulation, J. Biomech. 33 (2000) 1369–1376.
[37] K. Miller, K. Chinzei, Mechanical properties of brain tissue in tension, J. Biomech. 35 (2002) 483–490.

[38] M.I. Miga, T.K. Sinha, D.M. Cash, R.I. Galloway, R.J. Weil, Cortical surface registration for image-guided neurosurgery using laser-range scanning, IEEE Trans. Med. Imag. 22 (8) (2003) 973–985.
[39] H. Sun, H. Farid, K. Rick, A. Hartov, D.W. Roberts, K.D. Paulsen, Estimating cortical surface motion using stereopsis for brain deformation models, in: Medical Image Computing and Computer-Assisted Intervention – MICCAI 2003, Springer, Berlin, Heidelberg, 2003, pp. 794–801.
[40] A. Wittek, R. Kikinis, S.K. Warfield, K. Miller, Brain shift computation using a fully nonlinear biomechanical model, in: Eighth International Conference on Medical Image Computing and Computer Assisted Surgery, MICCAI 2005, Palm Springs, California, USA, 2005.
[41] A. Wittek, T. Hawkins, K. Miller, On the unimportance of constitutive models in computing brain deformation for image-guided surgery, Biomech. Model. Mechanobiol. 8 (1) (2009) 77–84.
[42] A. Horton, A. Wittek, G.R. Joldes, K. Miller, A meshless total Lagrangian explicit dynamics algorithm for surgical simulation, Int. J. Numer. Methods Biomed. Eng. (2010), doi:10.1002/cnm.1374.