Simulation Modelling Practice and Theory 19 (2011) 801–816

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

A real-time multigrid finite hexahedra method for elasticity simulation using CUDA

Christian Dick [*], Joachim Georgii, Rüdiger Westermann

Computer Graphics and Visualization Group, Technische Universität München, Germany

ARTICLE INFO

Article history:
Received 2 August 2010
Received in revised form 8 November 2010
Accepted 9 November 2010
Available online 21 November 2010

Keywords:
Elasticity simulation
Deformable objects
Finite element methods
Multigrid
GPU
CUDA

ABSTRACT

We present a multigrid approach for simulating elastic deformable objects in real time on recent NVIDIA GPU architectures. To accurately simulate large deformations we consider the co-rotated strain formulation. Our method is based on a finite element discretization of the deformable object using hexahedra. It draws upon recent work on multigrid schemes for the efficient numerical solution of partial differential equations on such discretizations. Due to the regular shape of the numerical stencil induced by the hexahedral regime, and since we use matrix-free formulations of all multigrid steps, computations and data layout can be restructured to avoid execution divergence of parallel running threads and to enable coalescing of memory accesses into single memory transactions. This enables to effectively exploit the GPU's parallel processing units and high memory bandwidth via the CUDA parallel programming API. We demonstrate performance gains of up to a factor of 27 and 4 compared to a highly optimized CPU implementation on a single CPU core and 8 CPU cores, respectively. For hexahedral models consisting of as many as 269,000 elements our approach achieves physics-based simulation at 11 time steps per second.

1. Introduction

Over the last years, graphics processing units (GPUs) have shown a substantial performance increase on intrinsically parallel computations. Key to this evolution is the GPU's design for massively parallel tasks, with the emphasis on maximizing total throughput of all parallel units. The ability to simultaneously use many processing units and to exploit thread level parallelism to hide latency have led to impressive performance increases in a number of scientific applications.

One prominent example is NVIDIA's Fermi GPU [1], on which we have based our current developments. It consists of 15 multiprocessors, on each of which several hundreds of co-resident threads can execute integer as well as single and double precision floating point operations. Double precision operations are running at 1/2 of the speed of single precision operations. Each multiprocessor is equipped with a register file that is partitioned among the threads residing on the multiprocessor, as well as a small low-latency on-chip memory block which can be randomly accessed by these threads. Threads are further provided with direct read/write access to global off-chip video memory. These accesses are cached using a two level cache hierarchy.

The threads on each multiprocessor are executed in groups of 32 called warps, and all threads within one warp run in lock-step. Due to this reason the GPU works most efficiently if all threads within one warp follow the same execution path. Automatic hardware multithreading is used to schedule warps in such a way as to hide latency caused by memory access operations. Switching between warps is virtually at no cost, since threads are permanently resident on a multiprocessor

[*] Corresponding author.
E-mail addresses: dick@tum.de (C. Dick), georgii@tum.de (J. Georgii), westermann@tum.de (R. Westermann).

1569-190X/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2010.11.005

(independently of whether they are running, blocked, or waiting). As a consequence, however, the registers are partitioned among *all* threads residing on a multiprocessor, which significantly reduces the number of registers available to each thread.

The Fermi GPU executes global memory accesses at a fix granularity of 128 bytes, i.e., the GPU reads or writes contiguous blocks of 128 bytes that are aligned at 128-byte boundaries. The hardware coalesces parallel accesses of the threads of a warp that lie in the same 128-byte segment into a single memory transaction. To effectively exploit the GPU's memory bandwidth, parallel accesses of the threads of a warp should therefore lie closely packed in memory to reduce the number of memory transactions and to avoid transferring of unnecessary data. Specifically, if the *i*th thread of a warp (half warp) accesses the *i*th 32-bit (64-bit) word of a 128-byte segment, these accesses are combined into a single memory transaction and the GPU's memory bandwidth is optimally used.

1.1. Contribution

We present a novel geometric multigrid finite element method on the GPU, and we show the potential of this method for simulating elastic material in real time on desktop PCs. To the best of our knowledge, this is the first multigrid finite element approach for solving linear elasticity problems that is realized entirely on the GPU. Since we use the co-rotational formulation of strain, even large deformations can be simulated at high physical accuracy. The CUDA API [2] is used because in contrast to graphics APIs like OpenGL or Direct3D it gives the programmer direct control over all available computing and memory resources.

To effectively exploit the GPU's massively parallel multithreading architecture, an algorithm must be restructured to expose a sufficient amount of fine-grained parallelism down or beyond one thread per data element. These threads should follow one common execution path and exhibit memory access patterns which enable coalescing of memory accesses to effectively exploit the massive memory bandwidth available on the GPU.

The particular restructuring we propose is based on a regular hexahedral discretization of the simulation domain, which provides a number of advantages for GPU-based deformable object simulation: First, a hexahedral discretization of a given object boundary surface can be generated at very high speed, including a multi-resolution representation that is required in a geometric multigrid approach. Second, the regular topology of the hexahedral grid leads to a numerical stencil of the same regular shape at each simulation vertex. This enables parallel processing of vertices using the same execution path and allows for memory layouts that support coalescing of memory access operations. Third, since all hexahedral elements have the same shape, only a single pre-computed element stiffness matrix is needed, which greatly reduces memory requirements. The stiffness matrix of a specific finite element is obtained from this matrix by scaling with the element's elastic modulus and by applying the current element rotation according to the co-rotated strain formulation.

Due to these advantages, we achieve performance gains of up to a factor of 27 compared to an optimized parallel CPU implementation running on a single CPU core. Even compared to the CPU implementation running on 8 CPU cores, our GPU implementation is a factor of up to 4 faster. This speed-up results from both the arithmetic and memory throughput on the Fermi GPU. Our CUDA implementation of the multigrid method achieves update rates of 120 time steps per second for models consisting of 12,000 hexahedral elements. For large models consisting of 269,000 elements, 11 time steps per second can still be achieved. Each time step includes the re-assembly of the system of equations, which is necessary due to the co-rotated strain formulation, as well as two multigrid V-cycles for solving this system. In combination with a high-resolution render surface, which is bound to the simulation model via pre-computed interpolation weights, a visually continuous rendering of the deformable body is achieved (see Fig. 1).

2. Related work

Over the last years, considerable effort has been spent on the efficient realization of general techniques of numerical computing on programmable GPUs [3,4]. Recent work in this field has increasingly focused on the use of the CUDA API [2],

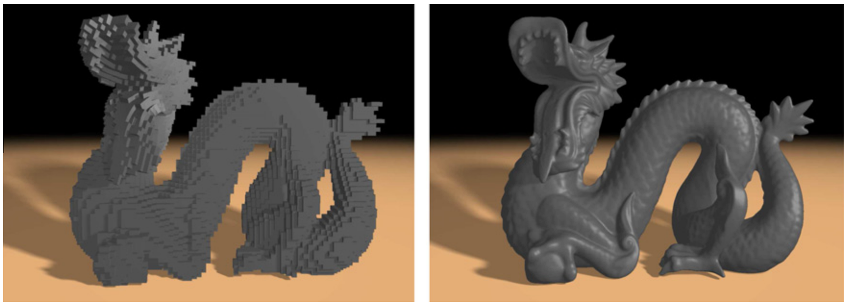

Fig. 1. Left: A deformed hexahedral object consisting of 30,000 elements is shown. Right: By using a high-resolution render surface that is bound to the deformed representation a visually continuous appearance is achieved.

addressing a multitude of different applications ranging from image processing and scientific visualization to fluid simulation and protein folding. CUDA provides a programming model and software environment for high-performance parallel computing on NVIDIA GPU architectures, allowing the programmer to flexibly adapt the parallel workload to the underlying hardware architecture. There is a vast body of literature related to this field and a comprehensive review is beyond the scope of this paper. However, [5,6] discuss the basic principles underlying the CUDA API and provide many practical details on the effective exploitation of the GPU's capacities via CUDA.

Over the last decades, extensive research has been pursued on the use of three-dimensional finite element (FE) methods to predict the mechanical response of deformable materials to applied forces (see, for example, [7] for a thorough overview). FE methods are attractive because they can realistically simulate the dynamic behavior of elastic materials, including the simulation of internal stresses due to exerted forces. Algorithmic improvements of FE methods, steering towards real-time simulation for computer animation and virtual surgery simulation have been addressed in [8–11].

Among the fastest numerical solution methods for solving the systems of linear equations arising in deformable model simulation are multigrid methods [12–14]. In a number of previous works, geometric multigrid schemes for solving the partial differential equations describing elastic deformations have been developed [15–17]. Interactive multigrid approaches for simulating linear elastic materials on tetrahedral and hexahedral grids have been proposed in [18–20], respectively.

In real-time applications, most commonly the linearized strain tensor, i.e. the Cauchy strain tensor, is used. However, since the Cauchy strain tensor is not invariant under rotations, computed element displacements tend to diverge from the correct solution in case of large deformations. The co-rotational formulation of finite elements [21] accounts explicitly for the per-element rotations in the strain computation and thus can handle non-linear relations in the elastic quantities. The efficient integration of the co-rotational formulation into real-time approaches has been demonstrated in [10,22,23].

FE-based deformable body simulation on the GPU has been addressed in a number of publications. The exploitation of a GPU-based conjugate gradient solver for accelerating the numerical simulation of the FE model has been reported in [24,25]. [26] presented a GPU-based FE surface method for cloth simulation. An overview of early GPU-accelerated techniques for surgical simulation is given by Sørensen and Mosegaard [27]. These approaches are mainly based on mass-spring systems [28]. Non-linear finite element solvers for elasticity simulation using graphics APIs and CUDA were presented by Taylor et al. [29], and Comas et al. [30], respectively. Both approaches build upon Lagrangian explicit dynamics [31] to avoid locking effects. While [29] employed a tetrahedral domain discretization, a discretization using hexahedral finite elements was used by Comas et al. [30]. [32] demonstrated clear performance gains for a multigrid Poisson solver on the GPU.

3. GPU-aware elasticity simulation

In the following we describe the physical model underlying our approach for real-time elasticity simulation, and we outline the algorithms that are used to enable fast and stable numerical simulation of this model. Special emphasis is put on the restructuring of these algorithms to support an efficient mapping to the GPU, involving matrix-free formulations of all computational steps.

3.1. Co-rotated linear elasticity

Underlying our simulation is a linear elasticity model combined with a co-rotational formulation of strain. In this model, we describe deformations as a mapping from the object's reference configuration $\Omega \subset \mathbb{R}^3$ to its deformed configuration $\{x + u(x) | x \in \Omega\}$ using a displacement function $\Omega \to \mathbb{R}^3$. Using a finite element discretization, the dynamic behavior of an object is governed by the Lagrangian equation of motion [7]

$$M\ddot{u} + C\dot{u} + Ku = f, \tag{1}$$

where M, C, and K denote the mass, damping and stiffness matrix, respectively. u is a vector built from the displacement vectors of all vertices and f is analogously built from the per-vertex force vectors. The stiffness matrix K is constructed by assembling the element stiffness matrices K^e, which are obtained by applying the principle of virtual work to each specific element.

Linear elasticity has the drawback that it is accurate only for relatively small deformations. It is based on a linear approximation of the strain tensor—meaning that there is a linear relationship between strains and displacements—and therefore can result in a significant volume increase in case of large deformations. To overcome this limitation we use the co-rotated strain formulation in our approach, which, in principle, rotates the element from the deformed to the reference configuration before the linear (Cauchy) strain is computed. This co-rotation is carried out on the finite element discretization by rotating the element stiffness matrices K^e accordingly.

3.2. Model construction

Our approach is based on a hexahedral discretization of the deformable object. The discretization is built from a voxelization of the object into a Cartesian grid structure, i.e., each grid cell is classified as inside or outside of the object boundary. The simulation model is then obtained by creating a hexahedral finite element for each interior cell. The regular hexahedral structure gives rise to a very efficient construction of a nested grid hierarchy that is essential for exploiting geometric multigrid schemes at their full potential. Due to the regular structure of the hexahedral discretization, computations can be parallelized effectively on SIMD architectures like GPUs.

From a given hexahedral model an octree hierarchy is built in a bottom-up process by successively considering grids of double cell size, i.e., the domain of a cell on the next coarser level coincides with the domain of a block of 2^3 cells on the current level. The respective next coarser level is constructed by creating a hexahedral element for exactly those cells which cover *at least one* hexahedral element on the current level. This process is repeated until the number of elements on the coarsest level is below a given threshold. On each hierarchy level a shared vertex representation is computed for the set of elements. Note that this construction process does not impose any restrictions on the size of the initial hexahedral model on the finest level. In particular, an element on the next coarser level is allowed to be only partially 'filled' with elements on the current level, for instance, at the object boundary.

In the numerical simulation, tri-linear shape functions are assigned to the finite hexahedral elements. Since all elements have the same shape, the same stiffness matrix

$$K^e = \int_{\Omega^e} B^{\mathrm{T}} D B \, \mathrm{d}x \tag{2}$$

can be used for all of them (up to scaling according to the respective element's elastic modulus). Here, B is the strain matrix and D is the material law. Requiring only a single element stiffness matrix significantly accelerates the setup phase for the simulation and greatly reduces memory demands. Note that even if the object geometry deforms and hexahedra become different shapes, no further calculations are required, since the discretization of the underlying partial differential equation always refers to the undeformed model state. Thus, the element stiffness matrices do not change.

3.3. Multigrid solver

Iterative methods such as Gauss–Seidel-type relaxation can be used in principle to solve the linear system of equations as it arises in the current application, because such methods can effectively exploit the system's sparsity. Such methods, on the other hand, require a large number of iterations until convergence of the solution. However, looking at the frequency spectrum of the error reveals that high frequencies are damped out very quickly by the relaxation, which yields the idea to solve the residual equation at a coarser grid, where the remaining low frequencies appear more oscillatory. This principle of coupling multiple scales to achieve improved convergence is underlying the basic multigrid idea. Specifically it can be shown that a linear time complexity of the solver in the number of unknowns can be achieved by applying this idea recursively on a hierarchy of successively coarser grids, yielding the so-called multigrid V-cycle scheme.

Numerical multigrid solvers are known to be among the most efficient solvers for elliptic partial differential equations of the form described above, and their potential has been exploited for simulating deformations using tetrahedral and hexahedral model discretizations. Our geometric multigrid solver builds upon these approaches, and it extends previous work by introducing a method to perform the computations for every element or vertex in lock-step using only coordinated memory accesses. Due to this property, the solver can effectively be mapped to the GPU via the CUDA API.

Before we discuss the GPU implementation in detail, we first derive the equations for the finite element simulation and the multigrid solver. Here, we put special emphasis on a matrix-free formulation that can be directly mapped to CUDA compute kernels.

3.3.1. Simulation level equations

In a hexahedral setting using the co-rotational strain formulation, the static elasticity problem for a single finite element is described by the following set of equations:

$$\sum_{j=1}^{8} R K_{ij} \left(R^{\mathrm{T}} (p_j^0 + u_j) - p_j^0 \right) = f_i, \quad i = 1, \ldots, 8. \tag{3}$$

Here, K_{ij} denotes a 3×3 block of the element stiffness matrix K^e, R is the rotation matrix determined for the element, u_j are the displacements vectors at the element vertices, p_j^0 are the positions of the element vertices in the undeformed state, and f_i are the forces acting on the element at its vertices. Solving for the unknown displacements u_j is performed by rearranging terms as

$$\sum_{j=1}^{8} \underbrace{R K_{ij} R^{\mathrm{T}}}_{\widehat{A}_{ij}} u_j = \underbrace{f_i - \textstyle\sum_{j=1}^{8} R K_{ij} \left(R^{\mathrm{T}} p_j^0 - p_j^0 \right)}_{\hat{b}_i}, \quad i = 1, \ldots, 8. \tag{4}$$

For the simulation of the dynamic behavior of the deformable object, the Newmark time integration scheme

$$\dot{u}_i = \frac{2}{dt} (u_i - u_i^{\mathrm{old}}) - \dot{u}_i^{\mathrm{old}} \tag{5}$$

$$\ddot{u}_i = \frac{4}{dt^2} (u_i - u_i^{\mathrm{old}} - \dot{u}_i^{\mathrm{old}} dt) - \ddot{u}_i^{\mathrm{old}} \tag{6}$$

is applied to the Lagrangian equation of motion

$$\sum_{j=1}^{8}\left(M_{ij}\ddot{u}_j+C_{ij}\dot{u}_j+\widehat{A}_{ij}u_j\right)=\hat{b}_i,\quad i=1,\dots,8,\tag{7}$$

leading to equations

$$\sum_{j=1}^{8}\underbrace{\left(\frac{4}{dt^2}M_{ij}+\frac{2}{dt}C_{ij}+\widehat{A}_{ij}\right)}_{A_{ij}}u_j=\underbrace{\hat{b}_i+\sum\nolimits_{j=1}^{8}\left(M_{ij}\left(\frac{4}{dt^2}\left(u_j^{\text{old}}+\dot{u}_j^{\text{old}}dt\right)+\ddot{u}_j^{\text{old}}\right)+C_{ij}\left(\frac{2}{dt}u_j^{\text{old}}+\dot{u}_j^{\text{old}}\right)\right)}_{b_i},\quad i=1,\dots,8.\tag{8}$$

Here, u_j^{old}, $\dot{u}_j^{\text{old}}$, and $\ddot{u}_j^{\text{old}}$ are the displacement vectors and their derivatives of the previous time step, and dt denotes the length of the time step. The 3×3 matrix coefficients A_{ij} and right-hand side vectors b_i are introduced to simplify the upcoming discussion. In our implementation, we use mass proportional damping ($C=\alpha M$ with $\alpha\in\mathbb{R}$) and mass lumping ($M_{ii}=m_iI_3$ with vertex masses m_i, and $M_{ij}=0$ for $i\neq j$).

The global system of equations can then be derived by accumulating the single equations of all hexahedral elements, thereby taking into account that elements share vertices, i.e., that there is one common u_i and f_i at a shared vertex. More precisely, an equation is built for every vertex $\mathbf{x}=(x_1,x_2,x_3)$ of the mesh by gathering the corresponding equations from the 8 incident hexahedra. $\mathbf{x}$ denotes integer coordinates of the vertex with respect to the underlying hexahedral grid. This results in per-vertex equations that reside on a 3^3 stencil of 27 adjacent vertices:

$$\sum_{\mathbf{i}=-\mathbf{1}}^{\mathbf{1}}A_{\mathbf{i}}^{\mathbf{x}}\,u_{\mathbf{x}+\mathbf{i}}=b_{\mathbf{x}}.\tag{9}$$

Here, $A_{\mathbf{i}}^{\mathbf{x}}$ are the accumulated 3×3 matrix coefficients associated with the adjacent vertex $\mathbf{x}+\mathbf{i}$, where $\mathbf{i}=(i_1,i_2,i_3)$ is the relative position of the adjacent vertex with respect to vertex $\mathbf{x}$. The notation $\mathbf{i}=-\mathbf{1},\dots,\mathbf{1}$ means iterating over all 27 3-tupels of the set $\{-1,0,1\}^3$, i.e. $(-1,-1,-1),(0,-1,-1),(1,-1,-1),\dots,(1,1,1)$. $u_{\mathbf{x}+\mathbf{i}}$ denotes the displacement vector at vertex $\mathbf{x}+\mathbf{i}$, and $b_{\mathbf{x}}$ is the accumulated right-hand side vector at vertex $\mathbf{x}$. Dirichlet boundary conditions are implemented by replacing the per-vertex equations of fixed vertices with dummy equations $I_3u_{\mathbf{x}}=0$.

3.3.2. Coarse grid equations

The geometric multigrid solver operates on a hexahedral grid hierarchy which is constructed in the setup phase. In the following, the respective current level and the next coarser level are indicated by sub- and superscripts h and $2h$, referring to the levels' grid spacings. The multigrid solver requires coarse grid versions of the system matrix A as well as restriction and interpolation operators to transfer quantities between subsequent grid levels. We use tri-linear interpolation for the multigrid interpolation operator I_{2h}^{h}, and the multigrid restriction operator R_h^{2h} is chosen to be the transpose of the interpolation operator, i.e., $R_h^{2h}=(I_{2h}^{h})^{\mathrm{T}}$. Furthermore, we use Galerkin-based coarsening, i.e., the coarse grid versions of the system matrix are successively built from fine to coarse grids via $A^{2h}=R_h^{2h}A^hI_{2h}^{h}$. In a matrix-free formulation, the coarse grid equations are built by distributing the equations at the fine grid vertices to coarse grid vertices and by simultaneously interpolating the displacement vectors at the fine grid vertices from the coarse grid vertices, using the weights illustrated in Fig. 2.

To construct the equations on the coarse grids, we propose a two-step approach as illustrated in Fig. 3. First, equations at fine grid vertices are distributed to coarse grid vertices (restriction, corresponding to computing linear combinations of the rows of A^h), yielding a 5^3 stencil on the fine grid with associated coefficients B. Second, these coefficients are distributed to the coarse grid vertices (interpolation, corresponding to computing linear combinations of the columns of A^h), thereby reducing the stencil to a 3^3 domain on the coarse grid. Note that the coarse grid vertex $\mathbf{x}$ corresponds to the fine grid vertex $2\mathbf{x}$ due to the different grid spacings. The construction is described by the following equations:

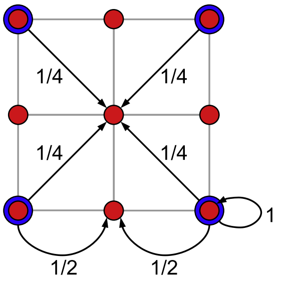

Fig. 2. Weights used to transfer quantities from a coarse grid (blue vertices) to the next finer grid (red vertices) of the multigrid hierarchy (interpolation). For the restriction from the coarser grid to the finer grid, the same weights are used. For simplicity, the weights are shown only for selected vertices in 2D. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

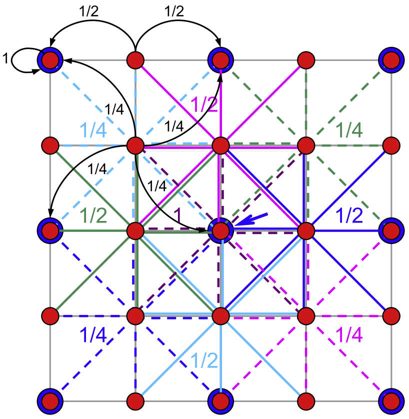

Fig. 3. Illustration of the construction of the coarse grid equation for a specific vertex (center vertex marked with a blue arrow). In the first step, a linear combination of the per-vertex equations (their stencils and the used weights are shown in different colors and line styles) in the 3^3 fine grid (red vertices) neighborhood of the considered vertex is computed. The resulting equation resides on a 5^3 stencil on the fine grid. In the second step, this equations is restricted to a 3^3 stencil on the coarse grid (blue vertices) by substituting the displacement vectors at the fine grid vertices by interpolation from the coarse grid vertices, corresponding to a distribution of the respective coefficients from the fine grid to the coarse grid vertices (black arrows and interpolation weights). Note that the weights shown in the figure correspond to the 2D case; the weights for the 3D case are given in the text. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

$${}^{h}B^{\mathbf{x}}_{\mathbf{i}} = \sum_{\substack{\mathbf{k}=-\mathbf{1}\\ |i_j-k_j|\leqslant 1,\ j=1,2,3}}^{\mathbf{1}} w_{\mathbf{k}}\ {}^{h}A^{2\mathbf{x}+\mathbf{k}}_{\mathbf{i}-\mathbf{k}}, \quad \mathbf{i}=-\mathbf{2},\ldots,\mathbf{2}, \tag{10}$$

$${}^{2h}A^{\mathbf{x}}_{\mathbf{i}} = \sum_{\substack{\mathbf{k}=-\mathbf{1}\\ |2i_j+k_j|\leqslant 2,\ j=1,2,3}}^{\mathbf{1}} w_{\mathbf{k}}\ {}^{h}B^{\mathbf{x}}_{2\mathbf{i}+\mathbf{k}}, \quad \mathbf{i}=-\mathbf{1},\ldots,\mathbf{1}. \tag{11}$$

In these equations, $w_{\mathbf{k}} = (2-|k_1|)(2-|k_2|)(2-|k_3|)/8$ are the weights used for restriction and interpolation. The additional conditions for the summation index variables ensure that no coefficients are fetched outside the valid ranges ($-1, \ldots, 1$ for coefficients A and $-2, \ldots, 2$ for coefficients B).

3.3.3. Multigrid V-cycle

The linear system of equations is solved by performing multigrid V-cycles, each consisting of the following steps:

1. Gauss–Seidel relaxation of the per-vertex equations (Eq. (9)) (n_1 steps).
2. Computation of the residual $r^h_{\mathbf{x}}$:

$$r^h_{\mathbf{x}} = b^h_{\mathbf{x}} - \sum_{\mathbf{k}=-\mathbf{1}}^{\mathbf{1}} {}^{h}A^{\mathbf{x}}_{\mathbf{k}}\ u^h_{\mathbf{x}+\mathbf{k}}. \tag{12}$$

3. Restriction of the residual to the next coarser grid:

$$r^{2h}_{\mathbf{x}} = \sum_{\mathbf{k}=-\mathbf{1}}^{\mathbf{1}} w_{\mathbf{k}}\ r^h_{2\mathbf{x}+\mathbf{k}}. \tag{13}$$

4. On the coarser grid, the error $e^{2h}_{\mathbf{x}}$ corresponding to the residual is determined by the residual equation

$$\sum_{\mathbf{k}=-\mathbf{1}}^{\mathbf{1}} {}^{2h}A^{\mathbf{x}}_{\mathbf{k}}\ e^{2h}_{\mathbf{x}+\mathbf{k}} = r^{2h}_{\mathbf{x}}. \tag{14}$$

   The residual equation is solved by applying this scheme recursively ($e^{2h}_{\mathbf{x}} \equiv u^{2h}_{\mathbf{x}},\ r^{2h}_{\mathbf{x}} \equiv b^{2h}_{\mathbf{x}}$) or solved directly if the coarsest level is reached.
5. Interpolation of the error back to the finer grid:

$$e^h_{\mathbf{x}} = \sum_{\substack{\mathbf{k}=-\mathbf{1}\\ \mathbf{x}+\mathbf{k}\equiv\mathbf{0}\pmod 2}}^{\mathbf{1}} w_{\mathbf{k}}\ e^{2h}_{(\mathbf{x}+\mathbf{k})/2}. \tag{15}$$

   The condition $\mathbf{x}+\mathbf{k}\equiv\mathbf{0} \pmod 2$ ensures that only coarse grid locations are considered.
6. Coarse grid correction: $u^h_{\mathbf{x}} += e^h_{\mathbf{x}}$.
7. Gauss–Seidel relaxation of the per-vertex equations (Eq. (9)) (n_2 steps).

In our implementation, we use $n_1 = 2$ pre-smoothing and $n_2 = 1$ post-smoothing Gauss–Seidel steps per V-cycle. On the coarsest level, a conjugate gradient solver is employed.

4. CUDA implementation

The CUDA implementation of the multigrid finite hexahedra method consists of two parts: a pre-process for creating the finite element model and the real-time simulation of the deformable model. In the pre-process, the finite element model is constructed and packed into several index arrays that are stored in GPU memory. Then, the finite element model is used for the real-time simulation of the object's deformations due to applied forces.

In the following, we first explain the data structures used to represent the finite element model in GPU memory (Section 4.1). We then show how the computations are parallelized and mapped onto the CUDA threading model (Section 4.2). The description of our CUDA implementation is completed by presenting a memory layout that enables coalesced memory accesses and thus facilitates using the full memory bandwidth available on the GPU (Section 4.3).

4.1. Data structures

The finite element model, including the multigrid hierarchy, is stored on the GPU using an indexed representation, i.e., finite elements and vertices[1] are addressed via indices. These indices are determined by enumerating the finite elements and the vertices in a specific order that will be explained in Section 4.3. The indices are counted from 0 and represented as 32-bit integer values. When referencing neighbors, parents, etc., a special index value of -1 is used to specify that an element or vertex is not existing.

For each finite element, we store its incident vertices, yielding an array with eight indices per element. *For each vertex in the multigrid hierarchy,* we store its neighbor vertices, i.e., the vertices in the 3^3 domain of the numerical stencil (array with 27 indices per vertex), the vertices on the next finer level which restrict to the considered vertex (array with 27 indices per vertex), as well as the vertices on the next coarser level which the considered vertex interpolates from (array with eight indices per vertex). Note that only up to 8 of the potential 27 indices in Eq. (15) are required due to the condition that vertices have to lie on the coarse grid. If less vertices are required, we store -1 to mark invalid indices. *For each vertex on the simulation level,* we additionally store its incident elements (array with eight indices per vertex), as well as its initial position in the undeformed state (array with three scalars per vertex). Note that these arrays are read-only, i.e., do not change during runtime.

Furthermore, *for each finite element,* we store the elastic modulus and density (two arrays, each with one scalar per element), and for *each vertex on the simulation level,* we store the external force vector acting at that vertex (array with three scalars per vertex) and whether the vertex is fixed or not (array with one bool per vertex). These three arrays can be written during runtime, for example to interactively change the applied forces or to adapt the stiffness of the finite elements.

The numerical simulation requires further arrays: *For each finite element,* we store a rotation matrix according to the co-rotational strain formulation (array with nine scalars per element). *For each vertex in the multigrid hierarchy,* we allocate memory for the per-vertex equations, i.e., the 3×3 matrix coefficients ${}^{h}A_{\mathbf{i}}^{\mathbf{x}}$ (array with $27 \cdot 9$ scalars per vertex), the right-hand side vectors $b_{\mathbf{x}}^{h}$ (array with three scalars per vertex), the displacement vectors $u_{\mathbf{x}}^{h}$ (array with three scalars per vertex), and the residual vectors $r_{\mathbf{x}}^{h}$ (array with three scalars per vertex). *For each vertex on the simulation level,* we furthermore store the displacement vectors $u_{\mathbf{x}}^{\text{old}}$ and their first and second derivatives $\dot{u}_{\mathbf{x}}^{\text{old}}$, $\ddot{u}_{\mathbf{x}}^{\text{old}}$ of the previous time step for Newmark time integration (three arrays, each with three scalars per vertex).

It is worth noting that the index-based representation of the finite element model—in contrast to an index-free representation based on a rectangular domain with implicit neighborhood relationships—has the advantage of requiring significantly less memory. This is due to the fact that the memory overhead induced by the index structures is small compared to the memory which would have to be allocated for the per-vertex equations for void regions outside of the object. Another advantage of the index-based representation is that it yields compact lists of elements and vertices (no void ranges), which greatly simplifies an efficient mapping of the computation onto the CUDA threading model, as shown in the next section. Despite of its slightly more irregular nature, we will show in Section 4.3 that the index-based representation nevertheless allows for CUDA-friendly memory layouts and can thus exploit the full bandwidth provided by the GPU.

4.2. Parallelization

In the following we discuss how the substeps of one simulation time step are parallelized and mapped onto the CUDA threading model. Note that we are using the co-rotated strain formulation, which requires to update the underlying system of equations as well as the multigrid hierarchy in every time step to consider the current element rotations.

[1] Note that the term 'finite element' only refers to the elements on the *finest* level of the multigrid hierarchy, but 'vertices' refers to the vertices at *all* levels of the multigrid hierarchy, unless noted otherwise.

4.2.1. Computation of the element rotations

The element rotations are computed by polar decomposition [33] of the elements' average deformation gradients. To parallelize these computations, we assign one CUDA thread to each finite element. Each thread fetches the current displacement vectors at the element's vertices from global GPU memory, determines the average deformation gradient, and iteratively computes its polar decomposition using five iteration steps. The resulting rotation matrix is stored in global memory.

4.2.2. Assembly of simulation level equations

For the assembly of the per-vertex equations on the simulation level, we assign one CUDA thread to each vertex. Each thread fetches the indices of the incident elements, and then loads the elements' density and elastic modulus values as well as the elements' current rotations. Furthermore, the thread fetches the external force applied to its vertex, as well as the original position, the displacement vector and its first and second derivatives of the previous time step, and the fixation status of the vertex. By using the single, pre-computed element stiffness matrix which is stored in cached constant memory, the thread assembles the per-vertex equation consisting of the 27 3×3 matrix coefficients as well as the right-hand side vector. Trading using an excessive number of registers per thread for an increase of memory traffic, the coefficients are assembled in global memory using read-modify-write operations.

4.2.3. Assembly of coarse grid equations

The levels of the multigrid hierarchy are assembled successively with one kernel call per level. We assign 9 CUDA threads to each vertex of the current level. The 9 threads first load the vertex indices of the corresponding 3^3 neighborhood on the previous finer level into shared memory. They then assemble the 27 3×3 matrix coefficients of the per-vertex equation. Each thread computes one of the 9 scalar components of every coefficient. To avoid costly read-modify-write operations to global memory, each thread uses 27 registers to store the intermediate values. To ensure lock-step execution, we map groups of 32 vertices to 9 warps (9×32 threads) such that the *i*th thread of each warp is assigned to the *i*th vertex.

4.2.4. Gauss–Seidel relaxation

The sequential version of the Gauss–Seidel algorithm traverses the vertices at a particular level and successively relaxes each per-vertex equation. For the relaxation of an equation, the *updated* displacement vectors at the previously visited vertices are used. To parallelize the Gauss–Seidel algorithm, these dependencies have to be considered. We employ the so-called multi-color Gauss–Seidel algorithm, which partitions the set of vertices into multiple subsets such that the vertices within each subset can be relaxed in parallel. The subsets, however, have to be processed sequentially. For the numerical stencil in our application, 8 subsets are required. They are defined by $\{\mathbf{x}|x_1 \bmod 2 = i_1, x_2 \bmod 2 = i_2, x_3 \bmod 2 = i_3\}$, $\mathbf{i} \in \{0,1\}^3$, i.e., when dividing the domain into blocks of 2^3 vertices, the *k*th vertex ($k = 1,\ldots,8$) of each block belongs to the *k*th subset.

To process the subsets sequentially, we issue one CUDA kernel call per subset. To compensate the reduced parallelism, which is especially important for medium-resolution finite element models having only a moderate number of vertices, we assign 13 CUDA threads to each vertex. Each thread computes two summands of the sum in Eq. (9). For each summand, the thread first fetches the index of the respective neighbor vertex, and then fetches the corresponding displacement vector. The respective 3×3 matrix coefficients are also loaded from global memory. The sum is then computed by a logarithmic reduction operation. The first thread finally computes the new displacement vector and writes it back into global memory. To ensure lock-step execution, we map groups of 32 vertices to 13 warps (13×32 threads) such that the *i*th thread of each warp is assigned to the *i*th vertex.

4.2.5. Computation of residual

For the computation of the residual, we assign one CUDA thread to each vertex. The computation is similar to the computation performed in the Gauss–Seidel relaxation step. In contrast, however, the residual computation does not exhibit any data dependencies, thus all vertices can be processed in parallel.

4.2.6. Transfer operators

For the transfer operators, we again assign one CUDA thread to each vertex. For the restriction operator, each thread iterates over the corresponding neighborhood on the next finer level to compute a weighted average of the residual vectors. For each neighbor, the thread first fetches the respective vertex index, and then loads the corresponding residual vector. The weighted average is finally written back into global memory, constituting the right-hand side vector of the per-vertex equation of the thread's vertex. Additionally, the thread initializes the displacement vector of its vertex with 0.

The interpolation operator is implemented in a similar way. Each thread iterates over the corresponding neighborhood on the next coarser level and computes a weighted average of the coarse grid correction vectors. This vector is added to the displacement vector of the thread's vertex.

Note that the transfer operators have to be implemented as gathering operations. Scattering would require atomic read-modify-write accesses to global GPU memory, since multiple threads might scatter to the same memory location. Furthermore, read-modify-write operations would increase memory traffic.

4.2.7. Conjugate gradient solver on the coarsest level

Considering that the number of vertices on the coarsest level is too small to fully exploit the parallelism offered by the GPU, and that global synchronization via multiple kernel calls is rather expensive, we run the conjugate gradient solver for the coarsest level on a single multiprocessor. We assign one CUDA thread to each vertex. The maximum number of vertices on the coarsest level is thus limited by the maximum number of threads per thread block (1024 on the Fermi GPU). To obtain an efficient multigrid V-cycle we use as many multigrid levels as are required to reduce the number of vertices on the coarsest level to less than 512.

In the CUDA threading model, threads are organized in larger groups called thread blocks. All threads in a thread block are scheduled on the same multiprocessor and can cooperate via its on-chip shared memory. For our implementation, experiments have shown that using a thread block of minimum size (32 elements/vertices per thread block) yields the best performance.

4.3. CUDA-friendly memory layout

The simulation of deformable objects comes with high memory requirements. Due to the underlying finite element discretization, the numerical stencil of a single vertex consists of 27 coefficients, with each coefficient being a 3×3 matrix. Moreover, the stencil is not constant for all vertices, but it varies due to different material parameters associated with each element and due to the co-rotated strain formulation. The numerical stencil leads to memory requirements of about 1 KB per vertex using 32-bit single floating point precision (2 KB per vertex using double precision).

A primary goal of our CUDA implementation thus is to effectively exploit the high memory bandwidth available on the GPU. In contrast to CPUs, maintaining data locality is not the main criterion for optimizing memory throughput on the GPU. Due to the specific hardware architecture of CUDA-enabled GPUs, it is mandatory to thoroughly coordinate memory access operations of parallel running threads in such a way that multiple memory accesses can be coalesced into single memory transactions. The Fermi GPU performs memory accesses at a fix granularity of 128 bytes, i.e., the GPU reads and writes entire blocks of 128 bytes which are aligned at 128-byte boundaries. Parallel memory accesses of the threads of a warp (consisting of 32 threads running in lock-step) that lie in the same 128-byte segment are coalesced into one single transaction. To maximize memory throughput, data thus should be organized in such a way that the ith thread of a warp (half warp) accesses the ith 32-bit (64-bit) word of a 128-byte segment. It is worth noting that without coalesced memory accesses, the effective memory throughput can decrease down to 1/32 of the GPU's memory bandwidth (in the case of requiring just a single 32-bit word of a 128-byte segment). Furthermore, since all threads in a warp are blocked until all memory access operations of the warp are finished, considerably higher latencies are introduced.

In the following, we describe how the data used in our application are stored in memory to allow for coalesced memory access operations. The main principle is to store each array of vectors or matrices such that their scalar components are grouped into separate memory blocks, i.e., the jth components of all vectors/matrices of the array are sequentially stored in the jth block. For the assignment of indices to the finite elements and vertices, we enumerate the elements as well as the vertices of each subset (for the multi-color Gauss–Seidel algorithm) in lexicographical order according to their 3D integer position (z first, y second, x third), with the vertices being enumerated continuously over all subsets and multigrid levels.

To align the memory accesses of warps at multiples of 128 bytes, the number of vertices per subset are rounded up to multiples of 32, i.e., the index of the first vertex of each subset is a multiple of 32. (The additional dummy vertices are marked as invalid by storing -1 in all index structures corresponding to these vertices.) In Fig. 4, we illustrate this memory layout for a generic array consisting of n elements with m scalar components per element. For each kernel call, we map each contiguous block of 32 indices to a warp (or several warps) of 32 threads. If the ith thread accesses the ith element of the array, the memory accesses are maximally coalesced and yield optimal memory throughput.

In our application, this setting is always met when a thread assigned to a specific element or vertex accesses data that are specific to that element or thread. However, when a thread accesses data belonging to a neighboring vertex (for example, when accessing the displacement vectors u in the Gauss–Seidel relaxation step), the situation is slightly different. In this case, the threads belonging to a warp still read a contiguous block of memory (except at the object's boundary), as illustrated in Fig. 5. However, since this block in general is not aligned at a 128-byte boundary, the hardware can only coalesce these memory accesses into two instead of one memory transaction.

5. Rendering

Even though the proposed CUDA implementation of elasticity simulation allows using model discretizations at reasonable resolution, a high-resolution render surface is required to achieve a visually continuous representation. To maintain and render such a representation efficiently on the GPU, we use CUDA and the graphics API OpenGL in combination.

We use the triangle surface mesh which is initially used to build the finite element model by voxelization. This surface mesh is stored in GPU memory, represented as an OpenGL index array that contains for every triangle references into a shared vertex array with associated per-vertex attributes. The shared vertex array is stored on the GPU as an OpenGL buffer object. Notably CUDA can directly write into OpenGL resources, thereby avoiding any copying operations.

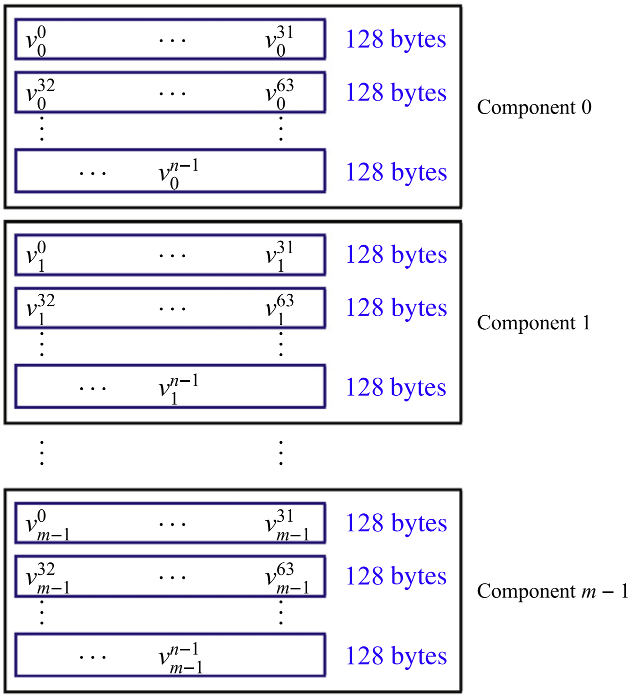

Fig. 4. Memory layout for a generic array v with n elements, each consisting of m 32-bit scalar components. v_j^i denotes the jth component of the ith element of the array. The jth components of all elements are stored sequentially in a separate memory block. If the ith thread accesses the ith element, the memory accesses can be optimally coalesced into 128-byte memory transactions (blue). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

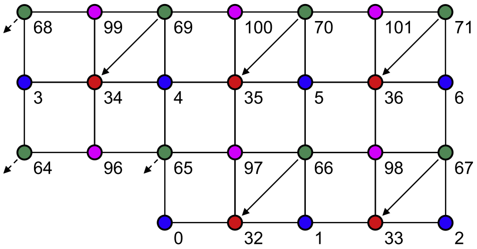

Fig. 5. Illustration of the efficiency for accessing data at the neighboring vertices. The colors of the vertices correspond to the subsets used for Gauss–Seidel relaxation. The numbers denote the indices of the vertices and correspond to the relative location of the per-vertex data in memory. In the example, the green vertices access data stored at their lower-left neighbors, which are all in the same subset (red vertices). Note that if the threads are sequentially assigned to the green vertices, the threads access their neighbors' data in contiguous memory blocks (except at the object's boundary). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The vertices of the render surface are bound to the vertices of the simulation grid via interpolation weights as illustrated in Fig. 6. For every render vertex, we determine the simulation element closest to this vertex—by using the distance between the vertex and the element center—and compute the tri-linear interpolation/extrapolation weights of the element vertices. These weights, together with respective references to the simulation vertices, are computed in the pre-process and stored in GPU memory.

At run-time, the displacement vectors of the simulation vertices are computed via CUDA as proposed in the previous Section, and the render surface vertices are updated according to these displacements using the pre-computed weights. Note that also the last step is performed via a CUDA compute kernel, which directly updates the OpenGL vertex array. Finally, the render surface is displayed using triangle rasterization.

6. Results

We analyze the performance of our CUDA-based multigrid finite element approach for simulating deformable objects using the Stanford bunny model at different resolutions, ranging from 12,000 to 269,000 hexahedral finite elements (see Fig. 12). All of

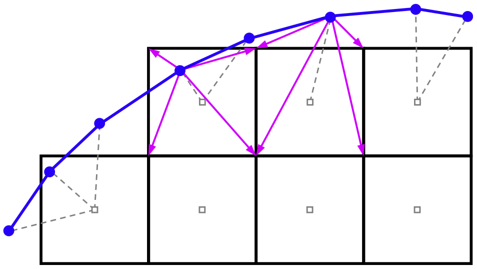

Fig. 6. Binding of a high-resolution render surface (blue) to the hexahedral simulation grid. Each render surface vertex is bound to the closest hexahedron with respect to the center of the elements (gray dashed lines). Magenta arrows indicate the element vertices used for tri-linear interpolation/extrapolation (for simplicity shown only for two selected vertices). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

our experiments were run on a high-end workstation, equipped with two quad core Intel Xeon X5560 Nehalem processors running at 3.2 GHz, 48 GB of DDR3 1333 MHz RAM, and an NVIDIA GeForce GTX 480 graphics card with 1.5 GB of video memory. Respectively half of the RAM is attached to each CPU's internal memory controller via triple channel, yielding an overall theoretical memory bandwidth of 29.7 GB/s per CPU. The two CPUs and the northbridge communicate via point-to-point quick path interconnect (QPI), which provides a theoretical bandwidth of 11.9 GB/s in each direction. Note that our system has a non-uniform memory access (NUMA) architecture, since accesses to a CPU's respective local memory are faster than accesses to its respective remote memory, which have to be performed via quick path interconnect.

The Fermi GPU on the graphics card has 480 scalar CUDA cores, running at 1401 MHz. Theoretically, each of these cores is capable of performing one fused multiply–add (FMA) operation per clock at single floating point precision, or one FMA operation every two clocks at double precision. This corresponds to a theoretical arithmetic throughput of 1.34 TFLOPS for single and 672 GFLOPS for double precision. The theoretical memory bandwidth between the GPU and the video memory is 165 GB/s.

Tables 1 and 2 show the performance of our implementation using single and double floating point precision, respectively. The construction of the simulation model is performed in a pre-process on the CPU. Each time step includes the computation of the element rotations, the assembly of the per-vertex equations on the simulation level and on the coarse grids of the multigrid hierarchy, as well as two multigrid V-cycles, each with two pre-smoothing and one post-smoothing Gauss–Seidel relaxation steps. The sustained rate of floating point operations performed per second in GFLOPS as well as the sustained memory throughput in GB/s are obtained by manually counting the number of floating point operations performed by each kernel, as well as the number of bytes read and written by each kernel (see Table 3).

The statistics thus report the *effective* memory throughput, i.e., the transfer of unnecessary data due to the fix memory transaction size of 128 bytes and cache hits are not considered. Note, however, that due to our optimized memory layout which facilitates coalescing of memory accesses and due to the fact that almost all data—the coefficients of the per-vertex

Table 1
Simulation performance on the GPU and CPU for different finite element models using *single* floating point precision. For each resolution, we first specify the number of hexahedral elements and the number of vertices (on the simulation level). We then specify the simulation time steps per second, the sustained rate of floating point operations per second in GFLOPS, and the sustained effective memory throughput in GB/s achieved on the GPU and on the CPU using 1, 4, and 8 cores, respectively.

Model		GPU			CPU, 1 core			CPU, 4 cores			CPU, 8 cores		
#Hex.	#Vert.	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	120	33.3	45.6	7.92	2.21	3.02	26.0	7.24	9.89	41.1	11.5	15.7
33,300	38,700	61.9	44.6	60.5	2.99	2.15	2.92	10.2	7.36	9.99	17.2	12.4	16.8
94,300	105,000	27.5	52.7	71.3	1.09	2.09	2.83	3.78	7.24	9.80	6.67	12.8	17.3
269,000	291,000	10.8	56.2	75.7	0.396	2.05	2.79	1.39	7.24	9.74	2.43	12.7	17.1

Table 2
Simulation performance using *double* floating point precision. The columns are analogous to Table 1.

Model		GPU			CPU, 1 core			CPU, 4 cores			CPU, 8 cores		
#Hex.	#Vert.	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s	Steps/s	GFLOPS	GB/s
11,900	14,600	91.9	25.6	67.7	6.93	1.93	5.11	20.6	5.73	15.1	31.6	8.81	23.3
33,300	38,700	41.4	29.8	78.3	2.58	1.86	4.88	7.70	5.54	14.5	12.9	9.32	24.5
94,300	105,000	17.1	32.7	85.4	0.951	1.82	4.76	2.85	5.47	14.3	4.75	9.09	23.8
269,000	291,000	6.54	34.2	88.8	0.345	1.80	4.68	1.02	5.34	13.9	1.69	8.81	22.9

Table 3
Detailed analysis of the costs per finite element per time step (total costs per time step divided by the number of finite elements) for each individual CUDA kernel. The analysis is based on the bunny model with 269,000 elements. From left to right, the columns contain the kernel, the number of FLOPs, the respective percentage of the total number of FLOPs, the number of bytes read and written for single and double floating point precision, the respective percentage of the total number of bytes read and written, and finally the measured percentage of GPU time spent for each individual kernel using single and double precision, respectively. The factors 6× and 2× correspond to performing 2 V-cycles per time step, each with 2 pre- and 1 post-smoothing Gauss–Seidel steps.

CUDA kernel	FLOPs	%	Bytes R/W		%	% GPU time	
			Single	Double		Single	Double
Computation of element rotations	470	2	160	300	1	0	1
Assembly of simulation level equations	10,000	51	6600	13,000	23	33	30
Assembly of coarse grid equations	3200	17	5100	10,000	18	24	21
Gauss–Seidel relaxation	6 × 660	20	6 × 1800	6 × 3500	39	27	33
Computation of residual	2 × 620	6	2 × 1800	2 × 3500	13	11	12
Restriction of residual	2 × 210	2	2 × 580	2 × 1000	4	1	1
Interpolation of error and coarse grid corr.	2 × 39	0	2 × 200	2 × 350	1	2	1
CG solver on coarsest level	2 × 3	0	2 × 1	2 × 1	0	1	1
Total (per finite element per time step)	19,000		28,000	54,000			

equations—are not accessed repeatedly in a kernel, the *effective* memory throughput should be close to the *physical* memory throughput. On the GPU using single floating point precision, we achieve 120 time steps per second for the 12,000 element model and 11 time steps per second for the 269,000 element model. For double precision, the update rates are 92 and 6.5 time steps per second, respectively.

Updating and rendering a high-resolution render surface (70,000 triangles) that was bound to the simulation grid took less than 6.5 ms in all examples. This time is not included in the timings in Tables 1 and 2.

In order to analyze the performance gain that is achieved by our GPU-based implementation, we compare it to an optimized CPU-based implementation which has been parallelized using OpenMP. The CPU implementation is very similar to the GPU implementation in that we use the same data structures and algorithms. The differences are as follows: On the CPU, we use one thread per core (with a fix assignment of threads to cores), and each thread processes a block of elements or vertices—in contrast to the GPU, where we use one or several threads per element or vertex. Corresponding to the different mapping of the work to threads, we use different memory layouts on the CPU and GPU. On the CPU, we store the scalar components of each element of an array directly one after another, whereas on the GPU the elements are stored interleaved by grouping corresponding scalar components of the array elements into separate memory blocks, as described in Section 4.3.

To achieve optimal performance on our NUMA CPU target architecture operated under Windows 7, we store the thread-local per-element and per-vertex data in the respective CPU's *local* memory. The operating system's default strategy for assigning a memory page to a memory frame is to choose a page from the local memory of the CPU that performs the first read/write access to the frame ('first touch' strategy). Thus, by initializing the respective memory addresses *from within the respective thread* directly after virtual memory allocation, the intended memory assignment is obtained.

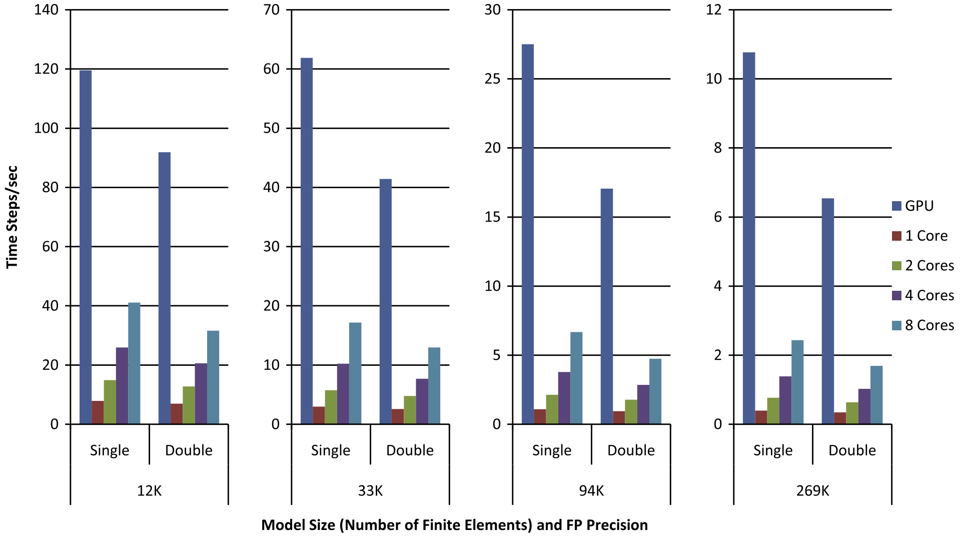

Fig. 7. Simulation time steps per second achieved for different model sizes as well as single and double floating point precision. Each group of five columns shows the performance obtained on the GPU and on the CPU using 1, 2, 4, and 8 cores. Each time step includes the re-assembly of the system of equations as well as two multigrid V-cycles.

We compare the GPU-based implementation to the CPU-based implementation running on 1, 2, 4, and 8 CPU cores, respectively. When using 1, 2, and 4 cores, we use cores belonging to the same CPU. For 1, 4 and 8 cores, the time steps per second, GFLOPS, and memory throughputs are listed in Tables 1 and 2. Fig. 7 shows the simulation time steps per second on the GPU and on the CPU for 1, 2, 4, and 8 cores. The respective speed-up factors, measured with respect to a single CPU core, are given in Fig. 8.

The diagram shows that the speed-up on the GPU increases with the model resolution due to a better utilization of the GPU's massive parallel architecture with increasing model size. For the largest model consisting of 269,000 finite elements, with respect to a single CPU core we achieve a speed-up of 27 for single and of 19 for double floating point precision. Even with respect to 8 CPU cores, the GPU is still a factor of about 4 faster for single and double precision. Figs. 9 and 10 show the floating point performance in GFLOPS and the memory throughput (in GB/s), respectively.

For the largest model we achieve 56 GFLOPS (single precision) and 34 GFLOPS (double precision) on the GPU. Since these values are clearly below the theoretical arithmetic throughput of the GPU, we assume that our GPU implementation is memory-bound. This is confirmed by the statistics on memory throughput, which report sustained rates of 76 GB/s for single and 89 GB/s for double precision, which is about half of the theoretical memory bandwidth. The decrease of performance when switching from single to double precision thus results from the doubled memory size of double precision values

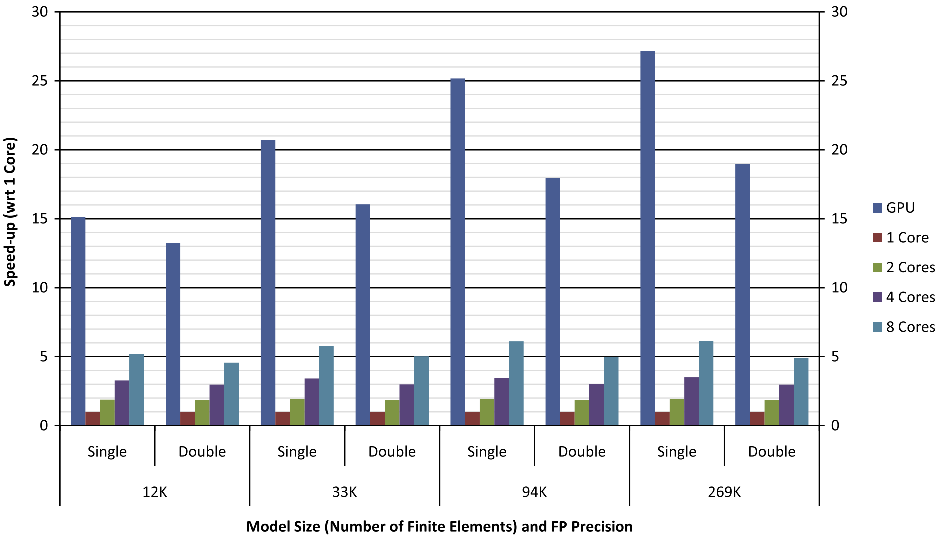

Fig. 8. Speed-ups achieved by parallelization on the GPU and on the CPU. All speed-ups are calculated with respect to a single CPU core.

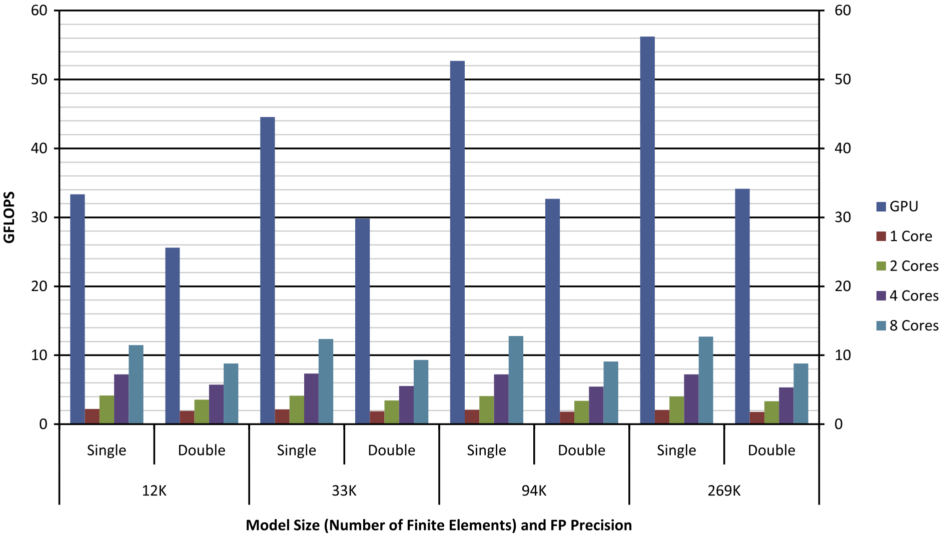

Fig. 9. Floating point performance (in GFLOPS) achieved on the GPU and on the CPU.

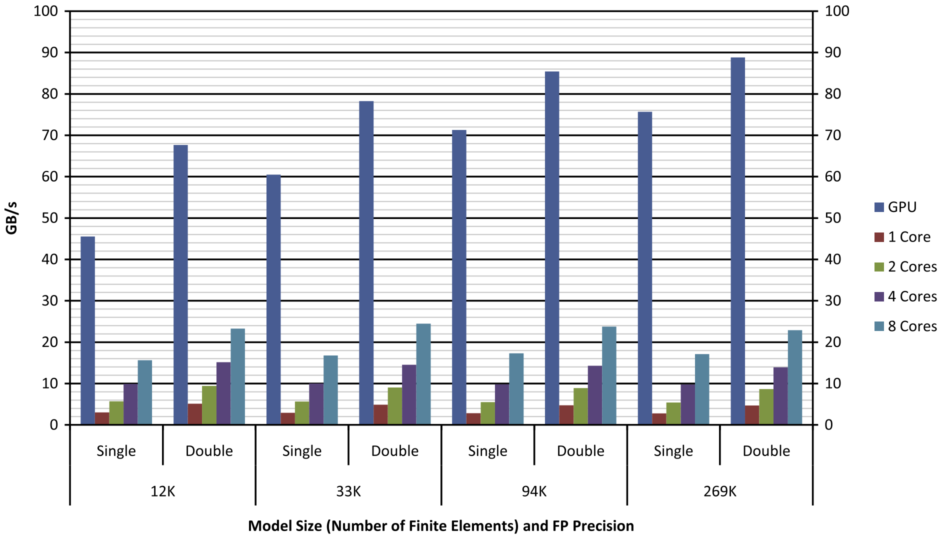

Fig. 10. Memory throughput (in GB/s) achieved on the GPU and on the CPU.

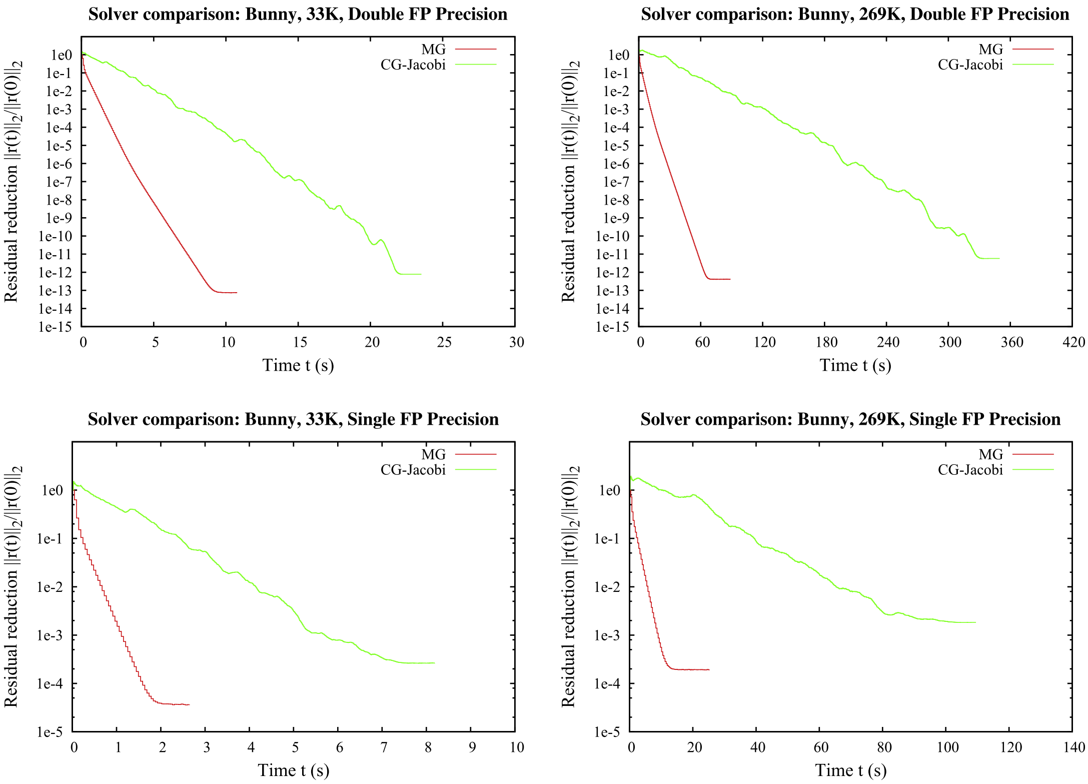

Fig. 11. Convergence of our multigrid solver (red) and a conjugate gradient solver with Jacobi pre-conditioner (green) with respect to computing time on a single CPU core, using two different model sizes as well as single and double floating point precision. The ordinate shows the reduction of the norm of the residual with respect to the start of the solver. ($E = 10^6$ Pa, $\nu = 0.3$, $\rho = 10^5$ kg/m^3, $dt = 50$ ms, edge length of hexahedral elements is 2.8 mm and 1.4 mm, respectively). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

compared to single precision values. On the CPU using 1 core, we achieve about 2 GFLOPS for both single and double precision. Two cores almost yield twice the performance of 1 core. Since the two cores are on the same CPU and thus use the same memory connection, this indicates that our CPU implementation is compute-bound on a single core. The statistics further show a better scalability in the number of cores for single precision than for double precision, and further report an

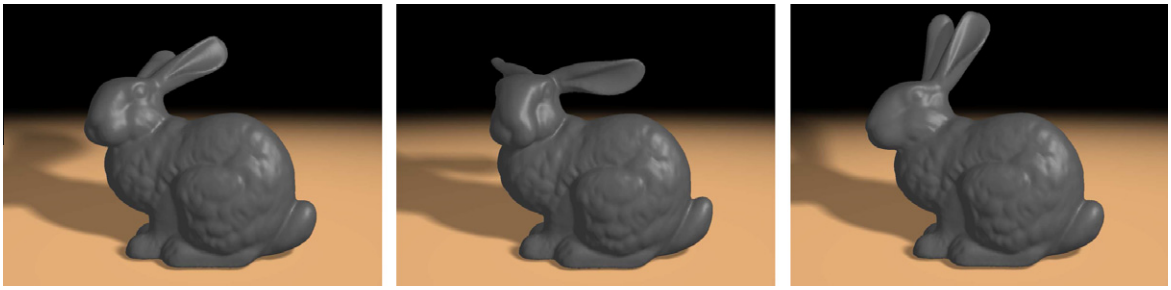

Fig. 12. Deformations of the Stanford bunny model (94,000 elements). The left image shows the model in the undeformed state. Simulation runs at 28 (17) time steps per second using single (double) floating point precision.

increasing impact on performance when switching from single to double precision with an increasing number of cores. This indicates that for double precision the CPU implementation is becoming memory-bound as more and more CPU cores are used. For the 269,000 element model, the CPU implementation achieves speed-ups of 1.76 and 1.65 using single and double precision, respectively, when going from 1 to 2 CPUs (corresponding to 4 and 8 cores).

In the future, we will also investigate the parallelization of our implementation on multiple GPUs. Since on current architectures GPU-to-GPU communication has to be initiated by the CPU and performed via PCI Express, we expect the scalability to be limited by the high latencies that are introduced.

In summary, the speed-ups achieved by the GPU compared to the CPU result from *both* the higher floating point performance and memory bandwidth on the GPU. Since for our application the limiting factor on the GPU is memory throughput, we expect the performance on future GPU architectures to be strongly related to the available memory bandwidth.

Finally, we also analyze the convergence behavior of our multigrid solver to demonstrate its suitability for the simulation on complicated domains. Fig. 11 shows the reduction of the norm of the residual achieved by the multigrid solver (red curves) with respect to computing time. The convergence behavior of a conjugate gradient (CG) solver with Jacobi preconditioner (green curves) is shown for comparison. Note that the timings were performed on the CPU (using 1 core) to allow for a precise measurement of the individual solver cycles. As can be seen, the multigrid solver exhibits a constant rate of residual reduction over time, which is characteristic for this solver type. Furthermore, the multigrid solver is significantly more efficient than a CG solver, in that in a given time budget it reduces the residual by a much higher factor than the CG solver. Note that we simulate the dynamics of deformable objects, thus the solution from the previous time step provides a very good initial guess to start with in the current time step. Due to this, we achieve high simulation accuracy using only 2 V-cycles per time step.

7. Conclusion

In this work we have presented a real-time method for physics-based elasticity simulation using CUDA on NVIDIA's Fermi GPU. The method employs the power of numerical multigrid schemes for the efficient solution of the governing differential equation. Underlying our approach is a hexahedral discretization of the simulation domain, giving rise to efficient algorithms for model construction and parallel simulation. By introducing a discretization-specific restructuring on the algorithmic level, the multigrid simulation scheme can efficiently be mapped to the GPU via the CUDA parallel programming API. In this way, significant performance gains can be achieved compared to our optimized parallel CPU implementation.

Our method also opens a number of areas for future research. Since our application is memory-bound, a mixed floating point precision approach as proposed in [34] might alleviate memory bandwidth limitations and further increase simulation performance. Another interesting question is how to parallelize the method on GPU clusters. Parallelization strategies similar to the one proposed in [17] will be considered, with the focus on minimizing inter-GPU communication. An additional challenge is to integrate GPU-based collision detection and handling in real-time deformable body simulations. Image-based techniques as proposed in [35] will be investigated for this purpose.

Acknowledgement

The first author is funded by the International Graduate School of Science and Engineering (IGSSE) of the Technische Universität München.

References

[1] NVIDIA, NVIDIA's next generation CUDA compute architecture: Fermi, 2009, <http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf>.
[2] NVIDIA, NVIDIA CUDA C programming guide version 3.1, 2010, <http://www.nvidia.com/cuda>.
[3] M. Houston, N. Govindaraju, GPGPU: general purpose computation on graphics hardware, in: ACM SIGGRAPH Courses, 2007.

[4] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips, GPU computing, Proceedings of the IEEE 96 (5) (2008) 879–899.

[5] C. Zeller, Tutorial CUDA, 2008, <http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf>.

[6] D.P. Luebke, I.A. Buck, J.M. Cohen, J.D. Owens, P. Micikevicius, J.E. Stone, S.A. Morton, M.A. Clark, High performance computing with CUDA, in: ACM/IEEE Supercomputing Tutorials, 2009.

[7] K.-J. Bathe, Finite Element Procedures, Prentice Hall, 2002.

[8] M. Bro-Nielsen, S. Cotin, Real-time volumetric deformable models for surgery simulation using finite elements and condensation, Computer Graphics Forum 15 (3) (1996) 57–66.

[9] S. Cotin, H. Delingette, N. Ayache, Real-time elastic deformations of soft tissues for surgery simulation, IEEE TVCG 5 (1) (1999) 62–73.

[10] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, B. Cutler, Stable real-time deformations, in: Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation, 2002, pp. 49–54.

[11] O. Etzmuß, M. Keckeisen, W. Straßer, A fast finite element solution for cloth modelling, in: Proc. Pacific Graphics, 2003, pp. 244–251.

[12] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Mathematics of Computation 31 (138) (1977) 333–390.

[13] W. Hackbusch, Multi-Grid Methods and Applications, Springer Series in Computational Mathematics, Springer, 1985.

[14] W.L. Briggs, V.E. Henson, S.F. McCormick, A Multigrid Tutorial, second ed., SIAM, 2000.

[15] I.D. Parsons, J.F. Hall, The multigrid method in solid mechanics: Part I – algorithm description and behaviour, International Journal for Numerical Methods in Engineering 29 (4) (1990) 719–737.

[16] M. Adams, J.W. Demmel, Parallel multigrid solver for 3D unstructured finite element problems, in: Proc. ACM/IEEE Supercomputing, 1999.

[17] R.S. Sampath, G. Biros, A parallel geometric multigrid method for finite elements on octree meshes, SIAM Journal on Scientific Computing 32 (3) (2010) 1361–1392.

[18] X. Wu, F. Tendick, Multigrid integration for interactive deformable body simulation, in: Proc. International Symposium on Medical Simulation, Lecture Notes in Computer Science, vol. 3078, 2004, pp. 92–104.

[19] J. Georgii, R. Westermann, A multigrid framework for real-time simulation of deformable bodies, Computer & Graphics 30 (3) (2006) 408–415.

[20] C. Dick, J. Georgii, R. Burgkart, R. Westermann, Computational steering for patient-specific implant planning in orthopedics, in: Proc. Eurographics Workshop on Visual Computing for Biomedicine, 2008, pp. 83–92.

[21] C.C. Rankin, F.A. Brogan, An element independent corotational procedure for the treatment of large rotations, ASME Journal of Pressure Vessel Technology 108 (2) (1986) 165–174.

[22] M. Hauth, W. Strasser, Corotational simulation of deformable solids, Journal of WSCG 12 (1) (2004) 137–144.

[23] J. Georgii, R. Westermann, Corotated finite elements made fast and stable, in: Proc. Workshop in Virtual Reality Interactions and Physical Simulation, 2008, pp. 11–19.

[24] W. Wu, P.A. Heng, A hybrid condensed finite element model with GPU acceleration for interactive 3D soft tissue cutting, Computer Animation and Virtual Worlds 15 (3–4) (2004) 219–227.

[25] Y. Liu, S. Jiao, W. Wu, S. De, GPU accelerated fast FEM deformation simulation, in: Proc. IEEE Asia Pacific Conference on Circuits and Systems, 2008, pp. 606–609.

[26] J. Rodriguez-Navarro, A. Susin, Non structured meshes for cloth GPU simulation using FEM, in: Proc. Workshop in Virtual Reality Interactions and Physical Simulation, 2006, pp. 1–7.

[27] T.S. Sørensen, J. Mosegaard, An introduction to GPU accelerated surgical simulation, in: Proc. International Symposium on Biomedical Simulation, Lecture Notes in Computer Science, vol. 4072, 2006, pp. 93–104.

[28] J. Mosegaard, P. Herborg, T.S. Sørensen, A GPU accelerated spring mass system for surgical simulation, Studies in Health Technology and Informatics 111 (2005) 342–348.

[29] Z.A. Taylor, M. Cheng, S. Ourselin, High-speed nonlinear finite element analysis for surgical simulation using graphics processing units, IEEE Transactions on Medical Imaging 27 (5) (2008) 650–663.

[30] O. Comas, Z.A. Taylor, J. Allard, S. Ourselin, S. Cotin, J. Passenger, Efficient nonlinear FEM for soft tissue modelling and its GPU implementation within the open source framework SOFA, in: Proc. International Symposium on Biomedical Simulation, Lecture Notes in Computer Science, vol. 5104, 2008, pp. 28–39.

[31] K. Miller, G. Joldes, D. Lance, A. Wittek, Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation, Communications in Numerical Methods in Engineering 23 (2) (2007) 121–134.

[32] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, S. Turek, Using GPUs to improve multigrid solver performance on a cluster, International Journal of Computational Science and Engineering 4 (1) (2008) 36–55.

[33] N.J. Higham, Computing the polar decomposition—with applications, SIAM Journal on Scientific and Statistical Computing 7 (4) (1986) 1160–1174.

[34] M.A. Clark, R. Babich, K. Barros, R.C. Brower, C. Rebbi, Solving lattice QCD systems of equations using mixed precision solvers on GPUs, Computer Physics Communications 181 (9) (2010) 1517–1528.

[35] J. Georgii, J. Krüger, R. Westermann, Interactive collision detection for deformable and GPU objects, IADIS International Journal on Computer Science and Information Systems 2 (2) (2007) 162–180.