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

Abstract

In this paper we present a GPU-based multigrid approach for simulating elastic deformable objects in real time. 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 and to support memory access patterns which enable the hardware to coalesce multiple 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 12 compared to a highly optimized CPU implementation. By using our approach, physics-based simulation at an object resolution of 64^3 is achieved at interactive rates.

Keywords: Elasticity simulation, deformable objects, finite element methods, multigrid, GPU, CUDA

1. Introduction

Over the last years, graphics processing units (GPUs) have shown a tremendous increase in performance on intrinsically parallel computations. Key to this evolution is the GPU's design for massively parallel workloads, 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 operations with high latency have led to substantially higher performance in applications where parallelism is abundant.

Modern GPUs consist of up to 30 multiprocessors, on each of which several hundreds of co-resident threads can be executed. Each multiprocessor contains a number of scalar processor cores which execute integer as well as single and double precision floating point operations. On current GPUs like NVIDIA's Fermi architecture (NVIDIA, 2009), double precision operations are running at only 1/2 of the speed of single precision operations. Threads are provided with direct read/write access to global off-chip DRAM and to a small low-latency on-chip memory segment per multiprocessor. In addition, an on-chip texture cache can substantially improve the performance of read-only memory accesses by reading linear global memory through textures.

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 multi-threading is used to schedule warps in such a way as to hide latency caused by memory access operations. To effectively increase memory throughput, the hardware coalesces global memory accesses of parallel threads into larger memory transactions if certain access patterns are met. Specifically, if the memory accesses by the threads of a half warp (16 threads) lie in a 128-byte segment (for example, when the *i*-th thread accesses the *i*-th word in the segment), these accesses are coordinated into one single transaction.

Contribution. In the light of the GPU's architectural design, we consider the parallelization of an important computational kernel in this paper, with the focus on memory bandwidth issues. We present a novel geometric multigrid finite element method on the GPU, and we show the use of this method for simulating elastic material in real time on desktop PCs. To fully exploit the GPU's massively parallel multi-threading architecture, the CUDA parallel programming abstraction (NVIDIA, 2008) is used. To the best of our knowledge, this is the first time that a multigrid finite element approach for solving the governing equations underlying elasticity is realized entirely on the GPU. Since we use the corotational formulation of strain, even large deformations can be simulated at high physical accuracy.

The CUDA API is used because in contrast to graphics APIs like OpenGL or Direct3D it gives the programmer direct control over all available memory resources. This enables application-specific restructuring on an algorithmic level to expose a sufficient amount of fine-grained parallelism and coherent memory accesses.

The particular restructuring we propose is based on a regular

*Corresponding author

Email addresses: dick@tum.de (Christian Dick), georgii@tum.de (Joachim Georgii), westermann@tum.de (Rüdiger Westermann)

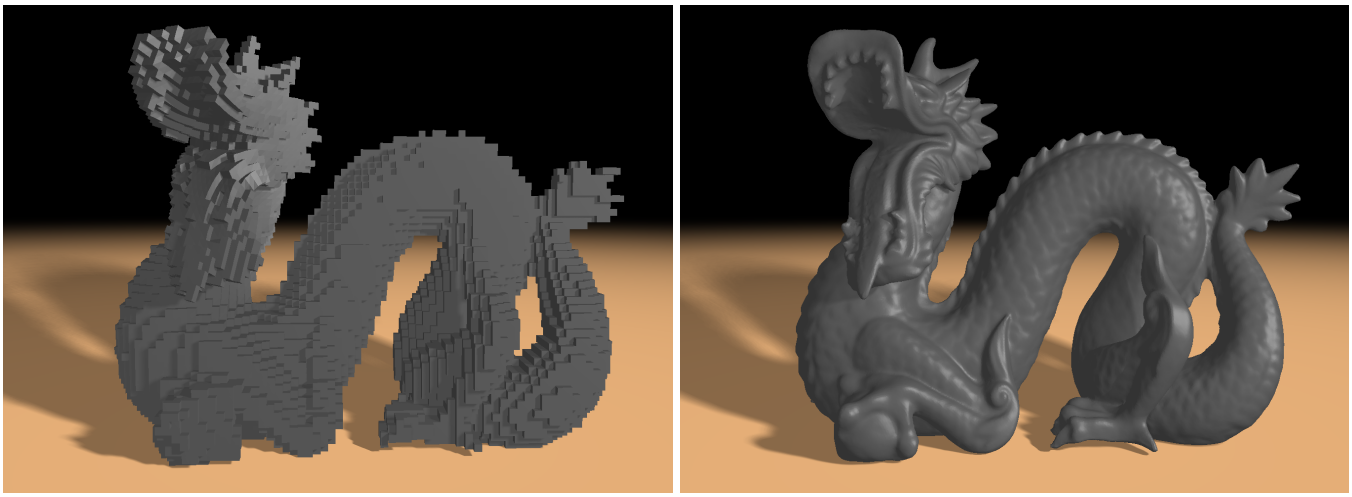

Figure 1: Left: A deformed hexahedral object consisting of 30,000 elements is shown. Right: By using a high-resolution render surface that is bound to the deformed representation a visually continuous appearance is achieved.

hexahedral discretization of the simulation domain. Hexahedral schemes provide 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 on the GPU, including a multi-resolution representation that is required in a geometric multigrid approach. Second, due to the regular topology of the hexahedral grid, a numerical stencil of the same regular shape is used for every simulation vertex, enabling parallel processing of vertices using the same execution path. Third, the regular stencil facilitates a memory layout which enables coalescing for all memory access operations, thereby exploiting the maximal memory bandwidth available on the GPU. Fourth, since all hexahedral elements have the same shape, only a single pre-computed element stiffness matrix is needed. 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 corotated strain formulation. The single pre-computed element stiffness matrix is stored in cached constant memory which greatly reduces global memory accesses during update of the simulation equations.

Due to these advantages, the restructuring we propose achieves performance gains of up to a factor of up to 12 compared to a cache-optimized CPU multigrid method using the same finite element discretization. This speed-up mainly results from the high memory bandwidth provided by the GPU. In combination with a high-resolution render surface, which is bound to the simulation model via pre-computed interpolation weights, physics-based, yet visually continuous deformable body simulation is achieved at high resolution. Figure 1 demonstrates the use of our approach. The proposed CUDA-based multigrid simulation achieves update rates of 110 ms per simulation step for models consisting of roughly 120,000 hexahedral elements, including two multigrid V-cycles and the update of the render surface.

2. Related Work

In the last years, considerable effort has been spent on the efficient realization of general techniques of numerical computing on programmable GPUs (Owens et al., 2005; Houston and Govindaraju, 2007; Owens et al., 2008). Recent work in this field has increasingly focused on the use of the CUDA API (NVIDIA, 2008), 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 multi-core 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, Zeller (2008) and Luebke et al. (2009) 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 (Martin and Carey, 1973; Bathe, 2002). 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 (Terzopoulos and Fleischer, 1988; Bro-Nielsen and Cotin, 1996; Cotin et al., 1999; Müller et al., 2002; Etzmuß et al., 2003).

Among the fastest numerical solution methods for solving the systems of linear equations arising in deformable model simulation are multigrid methods (Brandt, 1977; Hackbusch, 1985; Briggs et al., 2000). In a number of previous works, efficient multigrid schemes for solving the elliptic partial dif-

ferential equations describing elastic deformations have been proposed ((Parson and Hall, 1990; Adams and Demmel, 1999; Griebel et al., 2003; Sampath and Biros, 2009)). Especially for the use of multigrid approaches in medical applications, real-time approaches based on tetrahedral (Wu and Tendick, 2004; Georgii and Westermann, 2006) and hexahedral (Dick et al., 2008) discretizations have been considered.

In real-time applications, most commonly the linearized strain tensor, i.e. the Cauchy strain tensor, is used. However, since the Cauchy tensor is not invariant under rotations, computed element displacements tend to diverge from the correct solution in case of large deformations. The corotational formulation of finite elements (Belytschko and Hsieh, 1979; Rankin and Brogan, 1986; Felippa and Haugen, 2005) accounts explicitly for the per-element rotations in the strain computation, and, thus, it can handle non-linear relations in the elastic quantities. The efficient integration of the corotational formulation into real-time approaches has been demonstrated in (Müller et al., 2002; Hauth and Straßer, 2004; Georgii and Westermann, 2008). Non-linear FE methods for deformable body simulation were considered in (Zhuang and Canny, 1999; Wu et al., 2001; Picinbono et al., 2001; Debunne et al., 2001; Mendoza and Laugier, 2003; Picinbono et al., 2003; Zhong et al., 2005) to accurately handle geometric as well as material non-linearities.

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 (Wu and Heng, 2004; Liu et al., 2008). Rodriguez-Navarro and Susin (2006) 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 (2006). These approaches are mainly based on mass-spring systems (Mosegaard et al., 2005). Non-linear finite element solvers for elasticity simulation using graphics APIs and CUDA were presented by Taylor et al. (2008) and Comas et al. (2008), respectively. Both approaches build upon Lagrangian explicit dynamics (Miller et al., 2007) to avoid locking effects. While Taylor et al. (2008) employed a tetrahedral domain discretization, a discretization using hexahedral finite elements was used by Comas et al. (2008). Göddeke et al. (2008) demonstrated clear performance gains for a multigrid Poisson solver on the GPU.

3. GPU-Aware Elasticity Simulation

Physics-based elasticity simulation in real time places strong requirements on the used algorithms and technologies. Simulation tools supporting this kind of operability must be able to accurately predict the movement of the deformable material due to internal and external loads. The efficient and high-quality rendering of an accurate boundary surface of the moving body is crucial for using such tools in virtual environments such as surgery simulation, laparoscopy training, and needle insertion planning.

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. Corotated Linear Finite Element Method

Underlying our simulation is a linear elasticity model, meaning that there is a linear relationship between stresses and displacements. In this model, we describe deformations as a mapping from the object's reference configuration Ω to its deformed configuration $\{x + u(x) \mid x \in \Omega\}$ using a displacement function $u : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. The dynamic behavior of an object with linear elastic response is governed by the Lagrangian equation of motion on a finite element discretization

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

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 one 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, and therefore can result in a significant volume increase in case large deformations are applied. To overcome this limitation we use the corotated Cauchy strain formulation in our approach, which, in principle, rotates the element from the deformed to the reference configuration before the linear strain approximation is computed. This corotation 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. The advantage of this strategy is that the meshing process is greatly simplified, especially compared to a discretization using tetrahedral elements. Even though building a discretization is usually performed in a preprocess, applications like implant planning (Dick et al., 2008) require this discretization to be updated permanently and, thus, demand for procedures capable of dealing with real-time constraints.

The regular hexahedral structure also gives rise to a very efficient construction of a nested grid hierarchy that is essential for exploiting geometric multigrid schemes at their full potential. Such a scheme is used in our approach to substantially accelerate the numerical simulation of deformations at high object resolution. In addition, due to the regular structure of the hexahedral discretization, computations can be parallelized effectively on SIMD architectures like GPUs.

To build a hexahedral simulation model, different options exist. First, the model can be obtained directly from a given volume scan by classifying voxels into exterior and interior parts

based on any meaningful assignment procedure. The simulation model then consists of all interior voxels. Second, if a polygonal boundary surface representation of the object is available, which, for instance, can be constructed efficiently from a volume scan using surface fitting techniques like the marching cubes algorithm (Lorensen and Cline, 1987), a voxelization can be computed at very high speed on the GPU. GPU-based voxelization of a closed surface mesh leverages the rasterization hardware exposed via graphics APIs like OpenGL to efficiently determine the voxels in the interior of the mesh (Eisemann and Décoret, 2008). It essentially renders the mesh orthographically along a particular direction and determines along each ray of sight the intervals the ray is passing through the object. For instance, to generate a 256^3 hexahedral discretization of the boundary surface shown in Figure 1 it took roughly 5 ms on our target GPU architecture.

From a given hexahedral model representation an octree hierarchy is built in a bottom-up process by successively grouping 2^3 cells into one coarser cell. Starting on the finest (voxel) level, if a grid cell on the coarser level contains at least one interior voxel the cell is constructed. On all subsequent levels a cell is constructed if it covers at least one cell on the respective finer level. The process is repeated until the number of hexahedral elements on the coarsest level is below a given threshold. On each hierarchy level a shared vertex representation for the set of cells is computed.

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

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

can be used for all of them. Here, B is the strain matrix and D is the material law. Due to this property, the setup phase for the simulation is significantly accelerated. Furthermore, even if the object geometry deforms and hexahedra become different shapes no further calculations are required, since the discretization of the partial differential equation underlying the linear elastic model always refers to the undeformed model state, and 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 systems of linear equations as they arise in the current application, because such methods can effectively exploit the systems' 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 residual reveals that high frequencies in the solution are damped out very quickly by the relaxation, which yields the idea to solve the residual equation at a coarser grid. This principle of coupling multiple scales to achieve improved convergence is underlying the basic multigrid idea (Brandt, 1977; Hackbusch, 1985; Briggs et al., 2000). Specifically it can be shown that a linear time complexity of the solver can be achieved by applying this idea recursively, yielding the so-called multigrid V-cycle.

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 multigrid computations for every simulation element in lock-step using only coordinated data accesses across sets of elements. Due to this property, the solver can effectively be mapped to the GPU via the CUDA API. Before we are going to discuss the GPU implementation in detail, let us first derive the multigrid equations to be used on the finite hexahedra hierarchy. Here, we put special emphasis on a matrix-free formulation of the equations that can be directly mapped to CUDA compute kernels.

Per-Vertex Equations. In a hexahedral setting that respects the corotational strain formulation, the following set of equations is derived for every finite element (for simplicity reasons, we refer to the static elasticity problem here):

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

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 given 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 first rearranging terms as

$$\sum_{j=1}^{8} \underbrace{R\, K_{ij}\, R^{\mathrm{T}}}_{A_{ij}}\, u_j = \underbrace{f_i + \sum_{j=1}^{8} R\, K_{ij} \left(p_j^0 - R^{\mathrm{T}}\, p_j^0\right)}_{b_i} . \tag{2}$$

By applying the Newmark time integration scheme to the Lagrangian equation of motion (1), dynamics is included in the simulation (Bathe, 2002). The coefficients A_{ij} and right-hand sides b_i are introduced to simplify the upcoming discussion.

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. 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 regular grid. This results in per-vertex equations that depend on a regular 3^3 stencil of 27 adjacent vertices:

$$\sum_{\mathbf{i}=\mathbf{-1}}^{\mathbf{1}} A_{\mathbf{i}}^{\mathbf{x}}\, u_{\mathbf{x}+\mathbf{i}} = b_{\mathbf{x}}. \tag{3}$$

Here, $A_{\mathbf{i}}^{\mathbf{x}}$ are the accumulated 3×3 coefficient matrices associated with the adjacent vertex $\mathbf{i}$, 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 $\mathbf{i} = \mathbf{-1}, \ldots, \mathbf{1}$ means iterating over all 27 3-tupels of the set $\{-1, 0, 1\}^3$, i.e.

$(-1,-1,-1),(-1,-1,0),(-1,-1,1),\ldots,(1,1,1)$. $u_{\mathbf{x}+\mathbf{i}}$ denotes the displacement vector at vertex $\mathbf{x}+\mathbf{i}$, and $b_{\mathbf{x}}$ is the accumulated right-hand side at vertex $\mathbf{x}$.

Coarse Grid Equations. The geometric multigrid solver operates on a hierarchy of hexahedral grids which are constructed in the setup phase. In the following, the currently considered level and the next coarser level are indicated by superscripts h and $2h$ denoting the levels' grid spacings. To solve for the residual equations on the coarse grid levels, the equations at the coarse grid vertices have to be determined. We use tri-linear interpolation for the multigrid interpolation operator $I^{2h\to h}$, and use Galerkin-based coarsening to construct the coarse grid operators, i.e., $A^{2h} = R^{h\to 2h}\, A^h\, I^{2h\to h}$. The multigrid restriction operator $R^{h\to 2h}$ is chosen to be the transpose of the interpolation operator $I^{2h\to h}$, i.e., $R^{h\to 2h} = \left(I^{2h\to h}\right)^{\mathrm{T}}$. Transferring these equations to 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 substituting the displacement vectors at the fine grid vertices via interpolation from the coarse grid vertices, using the weights illustrated in Figure 2.

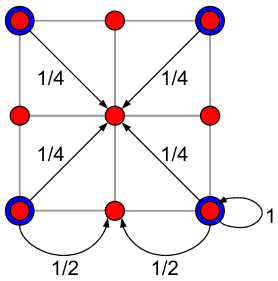

Figure 2: Weights used to transfer quantities from a coarse grid (blue vertices) to the next finer grid (red vertices) of the multigrid hierarchy (interpolation). For the restriction from the coarser grid to the finer grid, the same weights are used. For simplicity, the weights are shown only for selected vertices in 2D.

To construct the equations on the coarse grids, we propose a two-step approach as illustrated in Figure 3. First, equations at fine grid vertices are distributed to coarse grid vertices (multigrid restriction), yielding a 5^3 neighborhood of dependent fine grid vertices with associated coefficients B. Second, these coefficients are distributed to the coarse grid vertices (multigrid interpolation), thereby reducing the neighborhood to a 3^3 stencil. Note that the coarse grid vertex $\mathbf{x}$ corresponds to the fine grid vertex $2\mathbf{x}$ due to the different grid spacings. The following equations describe the distribution process between adjacent levels:

$$ {}^{h}B^{\mathbf{x}}_{\mathbf{i}} \;=\; \sum_{\substack{\mathbf{k}=\mathbf{-1}\\ |i_j-k_j|\le 1,\ j=1,2,3}}^{\mathbf{1}} w_{\mathbf{k}}\; {}^{h}A^{2\mathbf{x}+\mathbf{k}}_{\mathbf{i}-\mathbf{k}}\,, \quad \mathbf{i} = \mathbf{-2},\ldots,\mathbf{2}, \tag{4} $$

$$ {}^{2h}A^{\mathbf{x}}_{\mathbf{i}} \;=\; \sum_{\substack{\mathbf{k}=\mathbf{-1}\\ |2i_j+k_j|\le 2,\ j=1,2,3}}^{\mathbf{1}} w_{\mathbf{k}}\; {}^{h}B^{\mathbf{x}}_{2\mathbf{i}+\mathbf{k}}\,, \quad \mathbf{i} = \mathbf{-1},\ldots,\mathbf{1}. \tag{5} $$

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

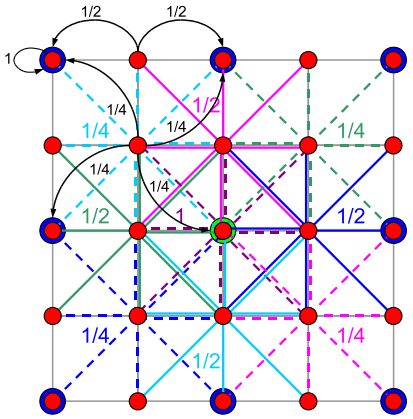

Figure 3: Illustration of the construction of the coarse grid equation for a specific vertex (green, center). In the first step, a weighted average of the per-vertex equations (their stencils and weights are shown in different colors and line styles) in the 3^3 fine grid (red vertices) neighborhood of 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 and green vertices) by substituting the displacement vectors at the fine grid vertices via 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.

Multigrid V-cycle. In the multigrid V-cycle, quantities are transferred between adjacent grid levels using the restriction and interpolation operators. Going down a V-cycle requires the residual to be restricted to the next coarser grid via

$$ b^{2h}_{\mathbf{x}} = \sum_{\mathbf{k}=\mathbf{-1}}^{\mathbf{1}} w_{\mathbf{k}}\; r^{h}_{2\mathbf{x}+\mathbf{k}}. \tag{6} $$

The residual r^h is computed as

$$ r^{h}_{\mathbf{x}} = b^{h}_{\mathbf{x}} - \sum_{\mathbf{k}=\mathbf{-1}}^{\mathbf{1}} {}^{h}A^{\mathbf{x}}_{\mathbf{k}}\; u^{h}_{\mathbf{x}+\mathbf{k}}. \tag{7} $$

Going up a V-cycle requires the coarse grid corrections e^{2h} to be interpolated to the next finer grid:

$$ e^{h}_{\mathbf{x}} = \sum_{\substack{\mathbf{k}=\mathbf{-1}\\ \mathbf{x}+\mathbf{k}\,\equiv\,\mathbf{0}\ (\mathrm{mod}\ 2)}}^{\mathbf{1}} w_{\mathbf{k}}\; e^{2h}_{(\mathbf{x}+\mathbf{k})/2}\,. \tag{8} $$

The condition $\mathbf{x}+\mathbf{k} \equiv \mathbf{0} \pmod 2$ ensures that only coarse grid vertices are considered. The resulting corrections are then added the current solution, yielding a complete multigrid V-cycle as follows:

1. Gauss-Seidel relaxation of Equation 3.
2. Compute residual $r^h_{\mathbf{x}}$ (Equation 7).
3. Restrict residual yielding $b^{2h}_{\mathbf{x}}$ (Equation 6).

4. Solve coarse grid residual equation (recursively):

$$\sum_{\mathbf{k}=-\mathbf{1}}^{\mathbf{1}} {}^{2h}A^{\mathbf{x}}_{\mathbf{k}}\, e^{2h}_{\mathbf{x}+\mathbf{k}} = b^{2h}_{\mathbf{x}}.$$

5. Interpolate correction $e^{2h}_{\mathbf{x}}$ yielding $e^{h}_{\mathbf{x}}$ (Equation 8).
6. Apply coarse grid correction: $u^{h}_{\mathbf{x}} \mathrel{+}= e^{h}_{\mathbf{x}}$.
7. Gauss-Seidel relaxation of Equation 3.

4. CUDA Implementation

The CUDA implementation of the multigrid finite hexahedra method consists of two parts: a preprocess for creating the finite element model and the real-time simulation of the deformable model. In the preprocess, 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 internal and external 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 memory layouts that enable coalesced memory accesses and thus facilitate 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, ..., 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), its corresponding vertices on the next finer level for restriction (array with 27 indices per vertex), as well as its corresponding vertices on the next coarser level for interpolation (array with eight indices per vertex). Note that only up to 8 of the potential 27 indices in Equation 8 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 finest 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 finest level,* we store the external force vector acting on 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 corotational strain formulation (array with nine scalars per vertex). *For each vertex in the multigrid hierarchy,* we allocate memory for the per-vertex equations, i.e., the 3×3 matrix coefficients ${}^{h}A^{\mathbf{x}}_{\mathbf{i}}$ (array with $27 \cdot 9$ scalars per vertex), the right-hand side vectors $b^{h}_{\mathbf{x}}$ (array with three scalars per vertex), the displacement vectors $u^{h}_{\mathbf{x}}$ (array with three scalars per vertex), and the residuum vectors $r^{h}_{\mathbf{x}}$ (array with three scalars per vertex). *For each vertex on the finest level,* we furthermore store the displacement vectors $u^{\text{old}}_{\mathbf{x}}$ and its first and second derivatives $\dot{u}^{\text{old}}_{\mathbf{x}}$, $\ddot{u}^{\text{old}}_{\mathbf{x}}$ 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 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 sub-steps of one simulation time step are parallelized and mapped onto the CUDA threading model. Note that we are using the corotated 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.

Computation of the Element Rotations. The element rotations are computed by polar decomposition (Higham, 1986) of the element's average deformation gradient. 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.

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

Assembly of the Per-Vertex Equations. For the assembly of the per-vertex equations on the finest level, we assign one CUDA thread to each vertex. Each thread first 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 deformation vector of the previous time step and its first and second derivatives for Newmark time integration, and the fixation status of that vertex. By using the single, pre-computed element stiffness matrix stored in cached constant memory, the thread then assembles the per-vertex equation consisting of the 27 3×3 matrix coefficients as well as the right-hand side. Note that we construct the coefficients strictly one after another to keep the number of registers used for temporary data as low as possible. Whenever a coefficient is constructed, it is immediately stored in global memory.

Assembly of the Multigrid Hierarchy. The levels of the multigrid hierarchy are assembled successively with one kernel call per level. We again assign one CUDA thread to each vertex of the current level. Each thread first loads the vertex indices of the corresponding neighborhood on the next finer level. It then computes the 3×3 matrix coefficients of the per-vertex equation. To reduce the number of registers used for temporary data while at the same time avoiding redundant global memory accesses to the respective equations on the next finer level, we compute the k-th entries ($k = 1 \dots 9$) of all 27 matrices simultaneously and write these entries into global memory before proceeding with the $(k + 1)$-st entries.

Gauss-Seidel Relaxation Step. The sequential version of the Gauss-Seidel algorithm traverses the vertices at a particular level and successively relaxes each per-vertex equation. In each relaxation step the *updated* displacement vectors from the previous relaxation steps 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 so 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 $\{\mathbf{x} \mid x_1 \mod 2 = i_1, x_2 \mod 2 = i_2, x_3 \mod 2 = i_3\}$, $\mathbf{i} \in \{0, 1\}^3$, i.e., when dividing the domain into blocks of 2^3 vertices, the k-th vertex ($k = 1 \dots 8$) 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 exhibiting only a moderate number of vertices, we assign two CUDA threads to each vertex. Each thread computes one half of the sum in Equation 3. 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. Upon completion, the first thread hands over his sum to the other thread by synchronization via on-chip shared memory. The second 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 two warps (2×32 threads), so that for each vertex the first and second half of the sum is computed by threads of the first and second warp, respectively.

Computation of the Residuum. For the computation of the residuum, 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 residuum computation does not exhibit any data dependencies, thus allowing all vertices to be processed in parallel.

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 residuum vectors. For each neighbor, the thread first fetches the respective vertex index, and then loads the corresponding residuum vector. The weighted average is finally written back into global memory, constituting the right-hand side 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 in 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, scattering would lead to a significantly higher memory traffic due to the absence of a cache for global GPU memory. This implies that for each scatter operation the current value must first be fetched from global memory before it can be modified and written back.

Conjugate Gradient Solver for 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. To store all temporary data in on-chip shared memory and to assign one CUDA thread to each scalar unknown, we use as many multigrid levels as are required to reduce the number of vertices on the coarsest level to less than 200.

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 showed that using a thread block size of 64 threads (2 warps) yields the best performance, i.e., the highest memory throughput. However, our implementation is rather insensitive

to the choice of the thread block size in that it exhibits a constantly high performance for a wide range of 32-512 threads per block.

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 as well as due to the corotated strain formulation. Therefore, we have an overall memory consumption of about 2 KB per vertex using 64-bit double floating point precision (1 KB for single precision).

The ultimate goal of our CUDA implementation is to effectively exploit the high memory bandwidth available on the GPU. In contrast to CPUs—which are equipped with cache hierarchies of several MB—current GPUs do not provide any caching of global memory accesses (with the exception of a very small texture cache for reading global memory via textures). Thus, 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. In particular, if the memory accesses of the threads of a half-warp (consisting of 16 threads running in lock-step) all fit into a 128-byte segment aligned at a 128-byte boundary, these memory accesses can be coalesced into one single transaction. To maximize memory throughput, data should be organized in such a way that the i-th thread of a half-warp accesses the i-th word of the segment. It is worth noting that without coalesced memory accesses, only up to 1/4 (when reading 64-bit double precision floating point values) of the maximal memory bandwidth can be exploited, since 32 byte is the smallest memory transaction size. Furthermore, since all threads in a half-warp are blocked until all memory access operations of that half-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 j-th components of all vectors/matrices of the array are sequentially stored in the j-th 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 half warps at multiples of 128 byte, the number of vertices per subset are rounded up to multiples of 16, i.e, the index of the first vertex of each subset is a multiple of 16. (The additional dummy vertices are marked as invalid by storing -1 in all index structures corresponding to these vertices.) In Figure 4, we illustrate this memory layout for a generic array consisting of n elements with m components per element. For each kernel call, we map each contiguous block of 32 indices to a warp of 32 threads. If the i-th thread accesses the i-th element of the array, the memory accesses are maximally coalesced and yield optimal memory throughput.

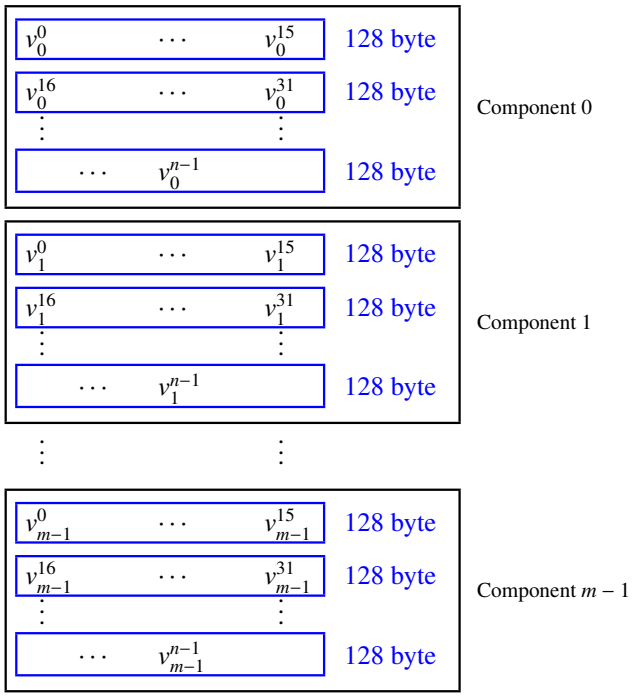

Figure 4: Memory layout for a generic array v with n elements, with each element consisting of m scalar components. v_j^i denotes the j-th component of the i-th element of the array. The j-th components of all elements are stored sequentially in a separate memory block. If the i-th thread accesses the i-th element, the memory accesses can be coalesced into single 128 byte memory transactions (blue).

In our application, this setting is always met when a thread assigned to a specific element or vertex accesses data that is 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 half-warp still read a contiguous block of memory (except at the object's boundary), as illustrated in Figure 5. Since this block is in general not aligned at a 128-byte boundary (or 64-byte boundary for 32-bit elements), the hardware can only coalesce these memory accesses in two instead of one memory transaction. Note, however, that in our application, all memory accesses to neighboring vertices are read accesses. This enables performing these read accesses through textures, resulting in almost maximal memory throughput due to the small on-chip texture cache.

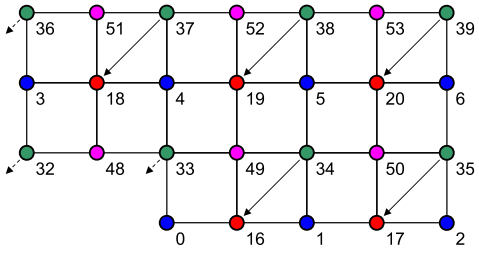

Figure 5: Illustration of the efficiency for accessing data at the neighboring vertices. The colors of the vertices correspond to the subsets used for Gauss-Seidel relaxation. The numbers denote the indices of the vertices, which correspond to the relative location of the per-vertex data in memory. In 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).

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.

In a preprocess, a triangular object boundary surface is constructed, for instance by using a super-resolution hexahedral discretization and reconstructing a triangular iso-surface from this discretization. The surface 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 as well as the array containing the vertices of the simulation grid are stored on the GPU as an OpenGL buffer object. Notably CUDA can directly write into these OpenGL resources, thereby avoiding any copying operations.

Vertices of the render surface are bound to vertices of the simulation grid via interpolation weights as illustrated in Figure 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 preprocess and stored in GPU memory.

At run-time, the displacement vectors of the simulation vertices are computed via CUDA as proposed in the previous Chapter, 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 arrays. Finally, the render surface is displayed using triangle rasterization.

6. Results

We analyze the performance of the CUDA-based multigrid approach for simulating deformable objects for a number of models with different resolutions (see Figures 7 to 9). All of our experiments were run on a desktop PC, equipped with an Intel Xeon X5482 3.2 GHz processor, 8 GB of RAM, and an NVIDIA Quadro FX 5800 graphics card with 4 GB of video memory. Updating and rendering a high-resolution render surface that was bound to the simulation grid took less than 3 ms in all examples.

Since all models were initially given as triangle surfaces, in the third column of Table 1 we first analyze the time which was required to construct a finite hexahedral representation from these models. As can be seen, the preprocessing times are always below one second, allowing instant construction and updates of the simulation grid.

For the models used in this paper, the second column in Table 1 provides the number of finite hexahedra used to represent these models as well as the number of degrees of freedom in the resulting systems of equations. The times given in columns 4 and 5 refer to one update step, including the computation of element rotations, the assembly of per-vertex equations, and the computation of the coarse grid equations, and one solve step, including two multigrid V-cycles using two pre-smoothing and one post-smoothing Gauss-Seidel step. Since our approach simulates the dynamics of deformable bodies, the solution from the last 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.

In order to analyze the performance gain that is achieved by our CUDA implementation, it is compared to a memory-optimized CPU implementation of the finite element approach described in Section 3. In contrast to the GPU implementation, which stores the data interleaved (see Section 4.3), on the CPU the data is stored linearly in memory to allow for cache coherent memory access operations and efficient hardware prefetching. The simulation vertices are stored in Z-order (Morton-order) to exploit coherence in the sequential processing of the simulation elements. As can be seen in column 6 of Table 1, the single threaded CPU implementation shows an *average* memory throughput that is close to the theoretical memory bandwidth of the system (8.3 GB/s). In particular this indicates that a parallelization on multi-core desktop PCs with standard memory interfaces can only marginally increase the performance.

For the CUDA implementation the timings in Table 1 indi-

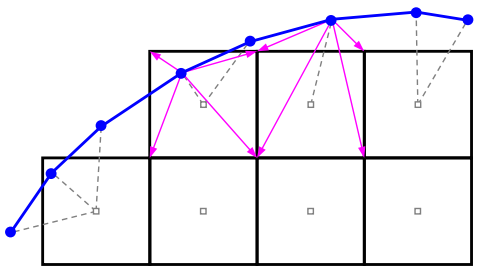

Figure 6: Binding of a high-resolution render surface (blue) to the hexahedral simulation grid. Each render surface vertex is bound to the closest hexahedron with respect to the center of the elements (gray dashed lines). Magenta arrows indicate the element vertices used for tri-linear interpolation/extrapolation (for simplicity shown only for two selected vertices).

Model	#Hexahedra	#DOFs	Time [ms] Preprocess	Time CPU [ms] Update	Time CPU [ms] Solve	Time GPU [ms] Update	Time GPU [ms] Solve	Memory throughput CPU	Memory throughput GPU
Liver	29,256	105,480	94	133	116	27	22	4.2 GB/s	21 GB/s
Horse	65,927	238,773	192	312	264	52	39	4.1 GB/s	26 GB/s
Dragon	119,698	442,428	375	581	496	59	50	4.1 GB/s	40 GB/s
Cube	262,144	823,875	821	1,191	1,030	105	83	3.7 GB/s	44 GB/s

Table 1: Simulation performance and average memory throughput on the CPU and the GPU for deformable models at different resolutions.

cate a speed-up of up to a factor of 12 compared to the CPU implementation. The speed-up is measured at double floating point precision. Switching to floating point precision on both the CPU and the GPU roughly doubles the performances, which attests once again that the simulation is memory bound. In fact, an analysis of the *average* memory throughput of the CUDA implementation (last column in Table 1) shows that we are close to the theoretical memory bandwidth of 102 GB/s of the NVIDIA Quadro FX 5800. This indicates that the application is well-optimized with respect to memory access operations, and, thus, a further speed-up on this architecture cannot be achieved.

In summary, the multigrid finite hexahedra methods we have implemented on the GPU using CUDA and on the CPU are both memory bound. As a consequence one can expect that the performance that can be achieved on future architectures is strongly related to the memory bandwidth available on these architectures. On Intel's new Core i7 processors, a theoretical memory bandwidth of 32 GB/s is possible, which yields a maximal speed-up of four on this CPU. On the other hand, NVIDIA's GTX 285 already has a theoretical memory bandwidth of 159 GB/s, and the upcoming NVIDIA Fermi architecture has an even higher bandwidth of 177.4 GB/s. Therefore, we expect the performance gap between the CUDA and the CPU implementation to remain in the future.

7. Conclusion

In this work we have presented a real-time method for physics-based elasticity simulation using CUDA. The method employs the power of numerical multigrid schemes for the efficient solution of the corresponding partial 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

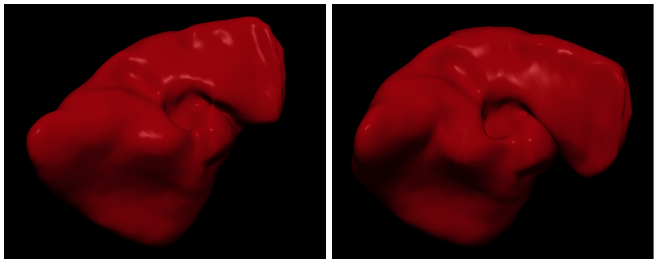

Figure 7: Real-time deformations for surgical simulation systems. The liver model consists of 29,000 elements, and the simulation runs with 20 time steps per second.

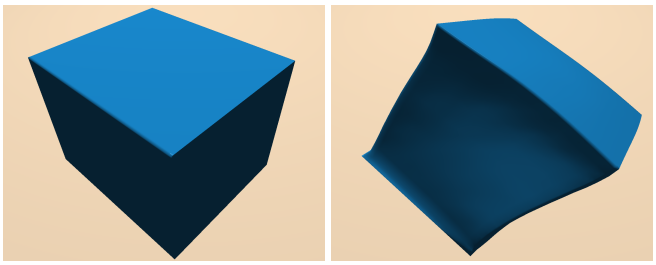

Figure 8: Simulation of the cube model (64^3 elements) runs with 5 time steps per second.

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 speed-ups can be achieved compared to an optimized CPU solution. The performance of our approach makes it amenable for model discretizations at high resolution, and, thus, it is well suited for interactive applications like virtual surgery simulation.

Our method also opens a number of areas for future research. One interesting question is how to parallelize the method on GPU clusters. Parallelization strategies similar to the one proposed in (Sampath and Biros, 2009) 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 (Georgii et al., 2007; Faure et al., 2008) will be investigated for this purpose. Finally, in order to further enhance the simulation tool towards interactive medical applications, research has to be pursued on the integration of real-time cutting algorithms.

Adams, M., Demmel, J.W., 1999. Parallel multigrid solver for 3d unstructured finite element problems, in: ACM/IEEE Supercomputing.

Bathe, K.J., 2002. Finite Element Procedures. Prentice Hall.

Belytschko, T., Hsieh, B.J., 1979. Application of higher order corotational stretch theories to nonlinear finite element analysis. Computers & Structures 10, 175–182.

Brandt, A., 1977. Multi-level adaptive solutions to boundary-value problems. Mathematics of Computation 31, 333–390.

Briggs, W.L., Henson, V.E., McCormick, S.F., 2000. A Multigrid Tutorial, Second Edition. SIAM.

Bro-Nielsen, M., Cotin, S., 1996. Real-time volumetric deformable models for surgery simulation using finite elements and condensation, in: Proceedings of Eurographics, pp. 57–66.

Comas, O., Taylor, Z., Allard, J., Ourselin, S., Cotin, S., Passenger, J., 2008. Efficient nonlinear fem for soft tissue modelling and its gpu implementation within the open source framework sofa, in: International Symposium on Computational Models for Biomedical Simulation, Springer. pp. 28–39.

Cotin, S., Delingette, H., Ayache, N., 1999. Real-time elastic deformations of soft tissues for surgery simulation, in: IEEE Transactions on Visualization

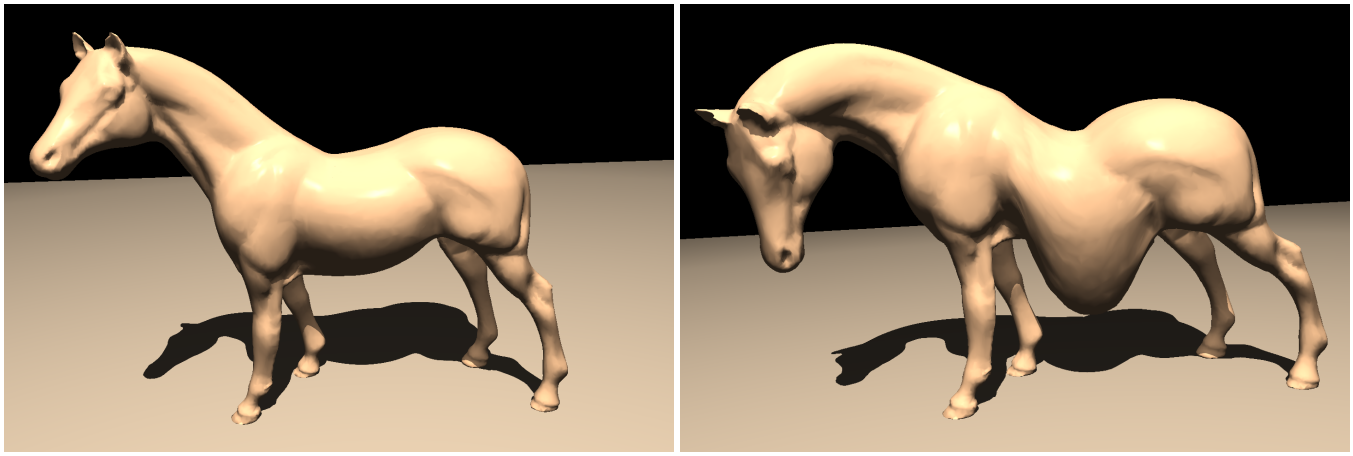

Figure 9: A finite element model (66,000 elements) composed of materials of different stiffness deforms under gravity. Simulation runs at 9 time steps per second.

and Computer Graphics, pp. 62–73.

Debunne, G., Desbrun, M., Cani, M.P., Barr, A.H., 2001. Dynamic real-time deformations using space & time adaptive sampling, in: Proceedings of SIGGRAPH, pp. 31–36.

Dick, C., Georgii, J., Burgkart, R., Westermann, R., 2008. Computational steering for patient-specific implant planning in orthopedics, in: Proceedings of Visual Computing for Biomedicine 2008, pp. 83–92.

Eisemann, E., Décoret, X., 2008. Single-pass GPU solid voxelization for real-time applications, in: Proc. Graphics Interface, pp. 73–80.

Etzmuß, O., Keckeisen, M., Straßer, W., 2003. A fast finite element solution for cloth modelling, in: Proceedings of Pacific Conference on Computer Graphics and Applications, p. 244.

Faure, F., Barbier, S., Allard, J., Falipou, F., 2008. Image-based collision detection and response between arbitrary volume objects, in: SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland. pp. 155–162.

Felippa, C., Haugen, B., 2005. A unified formulation of small-strain corotational finite elements: I. theory. Computer Methods in Applied Mechanics and Engineering 194, 2285–2335.

Georgii, J., Krüger, J., Westermann, R., 2007. Interactive collision detection for deformable and gpu objects. IADIS International Journal on Computer Science and Information Systems 2, 162–180.

Georgii, J., Westermann, R., 2006. A Multigrid Framework for Real-Time Simulation of Deformable Bodies. Computer & Graphics 30, 408–415.

Georgii, J., Westermann, R., 2008. Corotated finite elements made fast and stable, in: Proceedings of the 5th Workshop On Virtual Reality Interaction and Physical Simulation, pp. 11–19.

Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., Turek, S., 2008. Using gpus to improve multigrid solver performance on a cluster. Int. J. Comput. Sci. Eng. 4, 36–55.

Griebel, M., Oeltz, D., Schweitzer, M.A., 2003. An algebraic multigrid method for linear elasticity. SIAM Journal on Scientific Computing 25, 385–407.

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

Hauth, M., Straßer, W., 2004. Corotational simulation of deformable solids, in: Proceedings of WSCG, pp. 137–145.

Higham, N.J., 1986. Computing the polar decomposition—with applications. SIAM Journal on Scientific and Statistical Computing 7, 1160–1174.

Houston, M., Govindaraju, N., 2007. General-purpose computation on graphics hardware, in: SIGGRAPH 2007 GPGPU Course.

Liu, Y., Jiao, S., Wu, W., De, S., 2008. Gpu accelerated fast fem deformation simulation, in: Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on, pp. 606–609.

Lorensen, W.E., Cline, H.E., 1987. Marching cubes: A high resolution 3d surface construction algorithm. SIGGRAPH Comput. Graph. 21, 163–169.

Luebke, D.P., Buck, I.A., Cohen, J.M., Owens, J.D., Micikevicius, P., Stone, J.E., Morton, S.A., Clark, M.A., 2009. High performance computing with cuda, in: SuperComputing Tutorials.

Martin, H., Carey, G., 1973. Introduction to Finite Element Analysis: Theory and Application. McGraw-Hill Book Co, New York.

Mendoza, C., Laugier, C., 2003. Simulating soft tissue cutting using finite element models, in: Proceedings of IEEE International Conference on Robotics and Automation, pp. 1109–1114.

Miller, K., Joldes, G., Lance, D., Wittek, A., 2007. Total lagrangian explicit dynamics finite element algorithm for computing soft tissue deformation. Communications in Numerical Methods in Engineering 23.

Mosegaard, J., Herborg, P., Sørensen, T.S., 2005. A gpu accelerated spring mass system for surgical simulation. Studies in health technology and informatics 111, 342–348.

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

NVIDIA, 2008. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide (v. 2.2). http://www.nvidia.com/cuda.

NVIDIA, 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J., 2008. Gpu computing, in: Proceedings of the IEEE, pp. 879–899.

Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T., 2005. A survey of general-purpose computation on graphics hardware, in: Eurographics 2005.

Parson, I.D., Hall, J.F., 1990. The multigrid method in solid mechanics: Part i-algorithm description and behaviour. International Journal for Numerical Methods in Engineering 29, 719–737.

Picinbono, G., Delingette, H., Ayache, N., 2001. Non-linear and anisotropic elastic soft tissue models for medical simulation, in: Proceedings of IEEE International Conference on Robotics and Automation, pp. 1370–1375.

Picinbono, G., Delingette, H., Ayache, N., 2003. Non-linear anisotropic elasticity for real-time surgery simulation. Graph. Models 65, 305–321.

Rankin, C., Brogan, F., 1986. An element-independent co-rotational procedure for the treatment of large rotations. ASME J. Pressure Vessel Tchn. 108, 165–174.

Rodriguez-Navarro, J., Susin, A., 2006. Non structured meshes for cloth gpu simulation using fem, in: Proc. of 3rd. Workshop in Virtual Reality, Interactions, and Physical Simulations (VRIPHYS'06), pp. 1–7.

Sampath, R., Biros, G., 2009. A parallel geometric multigrid method for finite elements on octree meshes. In review, available online at http://www.cc.gatech.edu/grads/r/rahulss/.

Sørensen, T.S., Mosegaard, J., 2006. An introduction to gpu accelerated surgical simulation, in: Harders, M., Szekely, G. (Eds.), Third International Symposium, ISBMS 2006, Springer Berlin / Heidelberg. pp. 93–104.

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

Terzopoulos, D., Fleischer, K., 1988. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture, in: Proceedings of SIGGRAPH, pp. 269–

278.
Wu, W., Heng, P.A., 2004. A hybrid condensed finite element model with gpu acceleration for interactive 3d soft tissue cutting: Research articles. Comput. Animat. Virtual Worlds 15, 219–227.
Wu, X., Downes, M.S., Goktekin, T., Tendick, F., 2001. Adaptive nonlinear finite elements for deformable body simulation using dynamic progressive meshes, in: Proceedings of Eurographics, pp. 349–358.
Wu, X., Tendick, F., 2004. Multigrid integration for interactive deformable body simulation, in: Proceedings of International Symposium on Medical Simulation, pp. 92–104.
Zeller, C., 2008. Tutorial cuda. http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf.
Zhong, H., Wachowiak, M.P., Peters, T.M., 2005. A real time finite element based tissue simulation method incorporating nonlinear elastic behavior. Computer Methods Biomechan. Biomed. Eng. 8, 177–189.
Zhuang, Y., Canny, J., 1999. Real-time simulation of physically realistic global deformation, in: Proceedings of IEEE Visualization, pp. 270–273.