# Elastically Deformable Models based on the Finite Element Method Accelerated on Graphics Hardware using CUDA

Mickeal Verschoor
Eindhoven University of Technology,
The Netherlands
m.verschoor@tue.nl

Andrei C. Jalba
Eindhoven University of
Technology, The Netherlands
a.c.jalba@tue.nl

**Abstract**

Elastically deformable models have found applications in various areas ranging from mechanical sciences and engineering to computer graphics. The method of Finite Elements has been the tool of choice for solving the underlying PDE, when accuracy and stability of the computations are more important than, e.g., computation time. In this paper we show that the computations involved can be performed very efficiently on modern programmable GPUs, regarded as massively parallel co-processors through Nvidia's CUDA compute paradigm. The resulting global linear system is solved using a highly optimized Conjugate Gradient method. Since the structure of the global sparse matrix does not change during the simulation, its values are updated at each step using the efficient update method proposed in this paper. This allows our fully-fledged FEM-based simulator for elastically deformable models to run at interactive rates. Due to the efficient sparse-matrix update and Conjugate Gradient method, we show that its performance is on par with other state-of-the-art methods, based on e.g. multigrid methods.

**Keywords:** Elastically deformable models, Finite Elements, sparse-matrix update, GPU.

## 1 INTRODUCTION

Mathematical and physical modeling of elastically deformable models has been investigated for many years, especially within the fields of material and mechanical sciences, and engineering. In recent years, physically-based modeling has also emerged as an important approach to computer animation and graphics modeling. As nowadays graphics applications demand a growing degree of realism, this poses a number of challenges for the underlying real-time modeling and simulation algorithms. Whereas in engineering applications modeling of deformable objects should be as accurate as possible compared to their physical counterparts, in graphics applications computational efficiency and stability of the simulation have most often the highest priority.

The Finite Element Method (FEM) constitutes one of the most popular approaches in engineering applications which need to solve Partial Differential Equations (PDEs) at high accuracies on irregular grids [PH05]. Accordingly, the (elastically) deformable object is viewed as a continuous connected volume, and the laws of continuum mechanics provide the governing PDE, which is solved using FEM. Other popular methods are the Finite Difference Method (FDM) [TPBF87], the Finite Volume Method (FVM) [TBHF03] and the Boundary Element Method (BEM) [JP99] (see [NMK*06, GM97]). FDM is the easiest to implement, but as it needs regular spatial grids, it is difficult to approximate the boundary of an arbitrary object by a regular mesh. FVM [TBHF03] relies on a geometrical framework, making it more intuitive than FEM. However, it uses heuristics to define the strain tensor and to calculate the force emerging at each node. BEM performs the computations on the surface of the model, thus achieving substantial speedups as the size of the problem is proportional to the area of the model's boundary as opposed to its volume. However, this approach only works for objects whose interior is made of homogeneous material. Furthermore, topological changes are more difficult to handle than in FEM methods [NMK*06].

In this paper we present a fully-fledged FEM-based simulator for elastically-deformable models, running solely on GPU hardware. We show that the involved computations can be performed efficiently on modern programmable GPUs, regarded as massively parallel co-processors through Nvidia's CUDA compute paradigm. Our approach relies on the fast GPU Conjugate Gradient (CG) method of [VJ12] to solve the resulting linear system. Since the topology of the deformed mesh does not change during the simulation, the structure of the sparse-matrix describing the linear system is reused throughout the simulation. However, during the simulation, the matrix values have to be updated efficiently. To achieve this, we propose a method that updates the sparse-matrix entries respecting the ordering of the data, as required by the CG method of [VJ12], see Sect. 5.4. Thanks to the optimized CG method and the efficient sparse-matrix update procedure, we ob-

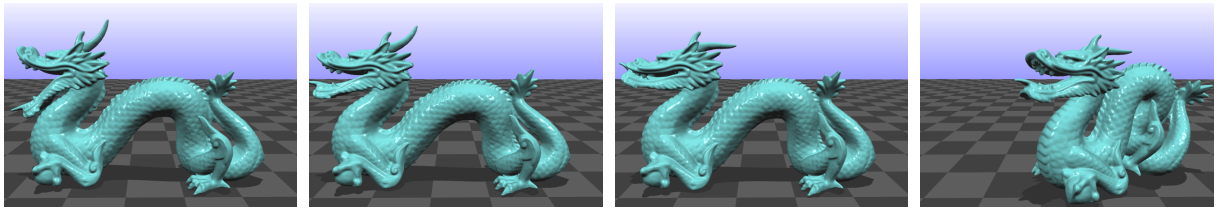


Figure 1: Effect of external (stretching) forces on an 'elastic' dragon.

tain results similar to state-of-the-art multigrid methods [DGW11].

The paper is organized as follows. Sections 3 and 4 describe the involved discretizations using FEM. Next, Section 5 presents the non-trivial parts of our GPU mapping, i.e., computing the local matrices, updating the global sparse matrix and solving the linear system. Finally, in Section 6 results are presented and analyzed.

## 2 PREVIOUS AND RELATED WORK

Bolz *et al.* [BFGS03], and Krüger and Westermann [KW03] were among the first to implement CG solvers on graphics hardware, using GPU programming based on (fragment) shaders. These authors had to deal with important limitations, *e.g.*, the lack of scatter operations, limited floating-point precision and slow texture switching based on pixel buffers, as exposed by the 'rendering-based' GPU-programming paradigm. One of the first GPU implementations of FEM is due to Rumpf and Strzodka [RS01], in the context of solving linear and anisotropic diffusion equations. Related work on GPU-accelerated FEM simulations also include the papers by Göddeke and collaborators [GST05, GST07, GSMY*07]. However, the emphasis is on improving the accuracy of *scientific* FEM-based simulations. Prior related work with respect to elastically deformable models, discretized using FEM, can be found in [HS04, MG04, ITF06]. They proposed methods which compensate for the rotation of the elements. Liu *et al.* [LJWD08] also present a FEM-based GPU implementation. Their results show that the involved CG method dominates the total computation time.

Since FEM often involves a CG solver, considerable research was done on efficiently mapping the CG method and Sparse Matrix-Vector Multiplications (SPMV) on modern GPUs using CUDA, see [BG08, BCL07, VJ12] and the references therein. Other approaches for solving the resulting PDE use multigrid methods, see e.g. [GW06]. An efficient GPU implementation of a multigrid method, used for deformable models, was recently presented in [DGW11]. Although multigrid methods typically converge faster than CG methods, implementing them efficiently on a GPU is a much more elaborate process. For example, invoking an iterative solver such as CG, constitutes only one of the steps of a multigrid method, the others being smoothing, interpolation and restriction.

## 3 ELASTICITY THROUGH THE METHOD OF FINITE ELEMENTS

As common in computer graphics applications (see [MG04] and the references therein), we employ a linearized model based on *linear* elasticity theory [PH05]. Further, to solve the underlying PDE we use the Method of Finite Elements with *linear tetrahedral elements*.

### 3.1 Continuum elasticity

In continuum elasticity, the deformation of a body, *i.e.*, a continuous connected subset $M$ of $\mathbb{R}^3$, is given by the *displacement* vector field $\boldsymbol{u}(\boldsymbol{x}) = [u(\boldsymbol{x}), v(\boldsymbol{x}), w(\boldsymbol{x})]^T$, where $\boldsymbol{x} = [x, y, z]^T$ is some point of the body at rest. Thus, every point $\boldsymbol{x}$ of the undeformed body corresponds to a point $\boldsymbol{x} + \boldsymbol{u}(\boldsymbol{x})$ of the deformed one.

The equilibrium equation of the deformation is usually written in terms of the *stress tensor*, $\boldsymbol{\sigma}$. However, since it cannot be measured directly, one uses Cauchy's *linear strain tensor*, $\boldsymbol{\varepsilon}$, and some material parameters to approximate the stress inside the body. Similar to Hooke's law for a 1D spring, in 3D one has

$$\boldsymbol{\sigma} = \boldsymbol{D} \cdot \boldsymbol{\varepsilon}, \tag{1}$$

for each point of the body, where $\boldsymbol{D} \in \mathbb{R}^{6\times 6}$ is the so-called *material stiffness matrix* representing material parameters. The elastic force $\boldsymbol{f}_e$ acting at a point of the body is given by

$$\boldsymbol{f}_e = \boldsymbol{K} \cdot \boldsymbol{u} = \left(\boldsymbol{P}^T \boldsymbol{D} \boldsymbol{P}\right) \cdot \boldsymbol{u}, \tag{2}$$

with $\boldsymbol{K} \in \mathbb{R}^{3\times 3}$, $\boldsymbol{f}_e$ and $\boldsymbol{u} \in \mathbb{R}^{3\times 1}$. $\boldsymbol{K}$ represents the local *stiffness matrix* and $\boldsymbol{P} \in \mathbb{R}^{6\times 3}$ is a matrix of partial derivative operators.

### 3.2 System dynamics

Having defined the elastical forces acting in a body, we now derive the equations of motion required to simulate the dynamic behaviour of the object. The coordinate vectors $\boldsymbol{x}$ are now functions of time, *i.e.* $\boldsymbol{x}(t)$, such that the equation of motion becomes

$$m\ddot{\boldsymbol{x}} + c\dot{\boldsymbol{x}} + \boldsymbol{f}_e = \boldsymbol{F}_{ext}, \tag{3}$$

where $m$ is the mass of a body particle at position $\boldsymbol{x}$, $c$ the damping coefficient, $\boldsymbol{f}_e$ the elastic force and $\boldsymbol{F}_{ext}$ the vector of external forces, i.e., the gravitational force. We approximate Eq. (3) using a *semi-implicit* method, *i.e.*,

$$m\frac{\left(\boldsymbol{v}^{i+1}-\boldsymbol{v}^{i}\right)}{\Delta t}+c\boldsymbol{v}^{i+1}+\boldsymbol{K}\cdot\boldsymbol{u}^{i+i}=\boldsymbol{F}^{i}_{ext}. \quad (4)$$

$$\boldsymbol{x}^{i+1}=\boldsymbol{x}^{i}+\Delta t\,\boldsymbol{v}^{i+1}, \quad (5)$$

with $\boldsymbol{u}^{i+1}=\Delta t\boldsymbol{v}^{i+1}+\boldsymbol{x}^{i}-\boldsymbol{x}^{0}$, which can be rearranged as

$$\left(m+\Delta tc+\Delta t^{2}\boldsymbol{K}\right)\cdot\boldsymbol{v}^{i+1}= m\boldsymbol{v}^{i}-\Delta t\left(\boldsymbol{K}\cdot\boldsymbol{x}^{i}-\boldsymbol{K}\cdot\boldsymbol{x}^{0}-\boldsymbol{F}^{i}_{ext}\right). \quad (6)$$

### 3.3 Discretization using FEM

Within FEM, the continuous displacement field $\boldsymbol{u}$ is replaced by a discrete set of displacement vectors $\tilde{\boldsymbol{u}}$ defined only at the nodes of the elements. Within each element $e$ the displacement field is approximated by

$$\boldsymbol{u}=\boldsymbol{\Phi}_e\cdot\tilde{\boldsymbol{u}}, \quad (7)$$

where $\boldsymbol{\Phi}_e\in\mathbb{R}^{3\times 12}$ is the matrix containing the element *shape functions* and $\tilde{\boldsymbol{u}}=[u_1,v_1,w_1,\ldots,u_4,v_4,w_4]^T$ is the vector of the nodal displacement approximations. Next, Galerkin's method of weighted residuals is applied over the whole volume $V$, in which the *weighting* functions are equal to the shape functions. Each term in Eq. (6) is weighted and approximated as in Eq. (7), which results in

$$\int_V\boldsymbol{\Phi}^T\left(m+\Delta tc+\Delta t^{2}\boldsymbol{K}\right)\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{v}}^{i+1}dV= \int_V m\boldsymbol{\Phi}^T\boldsymbol{\Phi}\tilde{\boldsymbol{v}}^{i}dV- \Delta t\int_V\boldsymbol{\Phi}^T\left(\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^{i}-\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^{0}-\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{F}}^{i}_{ext}\right)dV, \quad (8)$$

with $\boldsymbol{\Phi}^T$ the weighting functions. The equation above is defined for each individual element and generates one matrix consisting of the local mass ($\boldsymbol{M}_e$), damping ($\boldsymbol{C}_e$) and element stiffness ($\boldsymbol{K}_e$) matrices. Additionally, a local force matrix ($\boldsymbol{F}_e$) is generated, representing the net external force applied to the object. These local matrices are given by

$$\begin{aligned}\boldsymbol{M}_e&=\rho_e\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\,dV\\ \boldsymbol{C}_e&=c\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\,dV\\ \boldsymbol{K}_e&=\textstyle\int_V\boldsymbol{\Phi}^T\boldsymbol{P}^T\boldsymbol{D}\boldsymbol{P}\boldsymbol{\Phi}\,dV\\ \boldsymbol{F}_e&=\textstyle\int_V\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e\cdot\tilde{\boldsymbol{F}}_{ext}\,dV,\end{aligned} \quad (9)$$

with $\rho_e$ the density of element $e$. See [PH05] for more details on computing these matrices.

Finally, the global matrix $\boldsymbol{K}\in\mathbb{R}^{3n\times 3n}$ (with $n$ the number of mesh vertices) is 'assembled' from individual element matrices. This resulting system is then solved using the *Conjugate Gradient* method for the unknown velocity $\boldsymbol{v}^{i+1}$, which is then used to update the positions of the nodes, see Eq. (5). Eq. (5) shows a first order method for updating the positions which can be replaced by higher order methods as described in [ITF06].

Unfortunately, the above equations for simulating elastic deformation only work fine as long as the model does not undergo *large rotations*. This is because linearized elastic forces are used, which are only 'valid' close to the initial configuration. Therefore we use the so-called *Element-based Stiffness Warping* or *Corotational Strain* method [MG04, HS04] to compensate for the rotation of the elements. To extract the rotation part of the deformation, we use the polar decomposition method proposed in [Hig86]. The rotation-free element stiffness matrix $\boldsymbol{K}_{re}$ then becomes $\boldsymbol{K}_{re}=\boldsymbol{R}_e\boldsymbol{K}_e\boldsymbol{R}_e^{-1}$, with $\boldsymbol{R}_e\in\mathbb{R}^{12\times 12}$ the rotation matrix for element $e$. Note that this gives rise to an initial elastic force $\boldsymbol{f}_{e0}=\boldsymbol{R}_e\boldsymbol{K}_e\cdot\boldsymbol{x}_0$, which replaces the term $\boldsymbol{K}\boldsymbol{\Phi}\cdot\tilde{\boldsymbol{x}}^{0}$ in the right-hand-side of Eq. (8).

## 4 OVERVIEW OF THE ALGORITHM

Algorithm 1 gives an overview of the simulation of elastically deformable models as described in Section 3. First, a tetrahedralization of the polygonal mesh representing the surface of the object is computed, see Section 5.5. Each tetrahedron is considered as an element in FEM. Then, the initial stiffness-matrices of the elements are computed (line 3); these matrices do not change during the simulation and thus are precomputed. Additionally, as the shape functions are constant during the simulation, we can pre-calculate most matrices from Eq. (9), using $\boldsymbol{N}_1=\boldsymbol{\Phi}_e^T\boldsymbol{\Phi}_e$. This matrix is identical for all elements and is therefore only computed once.

**Algorithm 1** Simulation algorithm.

| | | |
|---|---|---|
| 1: | Compute $\boldsymbol{N}_1$ | see Eq. (9) |
| 2: | **foreach** element $e$ | |
| 3: | Compute $\boldsymbol{K}_e$ | see Eq. (9) |
| 4: | **loop of the simulation** | |
| 5: | **foreach** element $e$ | |
| 6: | Compute volume $v_e$ | |
| 7: | Compute $\boldsymbol{R}_e$ | see Section 3.3 |
| 8: | Compute $\boldsymbol{K}_{re}=\boldsymbol{R}_e\boldsymbol{K}_e\boldsymbol{R}_e^{-1}v_e$ | |
| 9: | Compute $\boldsymbol{M}_e=\rho_e\boldsymbol{N}_1v_e$ | |
| 10: | Compute $\boldsymbol{C}_e=c\boldsymbol{N}_1v_e$ | |
| 11: | Compute $\boldsymbol{f}_{e0}=\boldsymbol{R}_e\boldsymbol{K}_e\cdot\boldsymbol{x}_0v_e$ | |
| 12: | Compute $\boldsymbol{F}_e=\boldsymbol{N}_1\cdot\tilde{\boldsymbol{F}}_{ext}v_e$ | see Eq. (9) |
| 13: | Compute $\boldsymbol{F}_{te}=\boldsymbol{M}_e\cdot\boldsymbol{v}^i-\Delta t\left(\boldsymbol{f}_{e0}-\boldsymbol{K}_{re}\cdot\boldsymbol{x}^i-\boldsymbol{F}_e\right)$ | |
| 14: | Compute $\boldsymbol{K}_{te}=\boldsymbol{M}_e+\Delta t\boldsymbol{C}_e+\Delta t^2\boldsymbol{K}_{re}$ | |
| 15: | Assemble global $\boldsymbol{K}$ and $\boldsymbol{F}$ using $\boldsymbol{K}_{te}$ and $\boldsymbol{F}_{te}$ of elements | |
| 16: | Solve $\boldsymbol{K}\cdot\boldsymbol{v}^{i+1}=\boldsymbol{F}$ for $\boldsymbol{v}^{i+1}$ | |
| 17: | Update $\boldsymbol{x}^{i+1}=\boldsymbol{x}^i+\Delta t\boldsymbol{v}^{i+1}$ | see Section 3.2 |

After all local matrices have been computed and stored (line 14), the global matrix is assembled

(line 15). The resulting linear system of equations is solved for velocities (line 16), which are then used to advance the position vectors (line 17).

# 5 GPU MAPPING USING CUDA

In this section we describe our GPU mapping of the simulation on NVida GeForce GPUs using CUDA [NVI]. First we shall give details about implementing the rotation extraction through polar decomposition. Then, we describe the computation of the local stiffness matrices which are used to assemble the global (sparse) stiffness matrix (matrix $\boldsymbol{K}$ in Algorithm 1). The resulting system of linear equations is solved using a Jacobi-Preconditioned CG Method.

## 5.1 Rotation extraction

As mentioned in subsection 3.3 we have to estimate the rotation of each element in order to calculate displacements properly. Finding the rotational part of the deformation matrix is done using a Polar Decomposition as described in [MG04, HS04, Hig86]. Although a large number of matrix inversions is involved, this can be done efficiently because small $4 \times 4$ matrices are used. Since each matrix contains 16 elements, we chose to map the computations of 16 such matrices to a single CUDA thread-block with 256 threads.

For computing the inverse of a $4 \times 4$ matrix we perform a *co-factor expansion*. Each thread within a thread-block computes one co-factor of the assigned matrix. Since the computation of a co-factor requires almost all values of the matrix, memory accesses have to be optimized. In order to prevent for possible bank-conflicts during the computation of the co-factors, each matrix is stored in one memory bank of shared memory. Accordingly, the shared-memory segment (of size $16 \times 16$ locations) is regarded as a matrix stored in row-major order, where each column represents a $4 \times 4$ local matrix. Therefore, each column (local matrix) maps exactly to one memory-bank. Since a large number of rotation matrices are computed in parallel, a large performance boost is obtained.

## 5.2 Local stiffness matrices

Solving a specific problem using FEM starts with describing the problem locally per element. Since a typical problem consists of a large number of elements, the computations involved per element can be easily parallelized. Further, since the matrices used to construct $\boldsymbol{K}_e$ are in $\mathbb{R}^{12\times12}$, we map the computation of each individual local element stiffness matrix to a thread-block containing $12 \times 12$ threads. The inner loop in Algorithm 1 is implemented using one or two CUDA kernels, depending on the architecture version. Instead of creating kernels for each individual matrix operation, we combine a number of them into one larger kernel. Since data from global memory can be reused multiple times, less global memory transactions are required, which improves the overall performance.

## 5.3 Solving the linear system

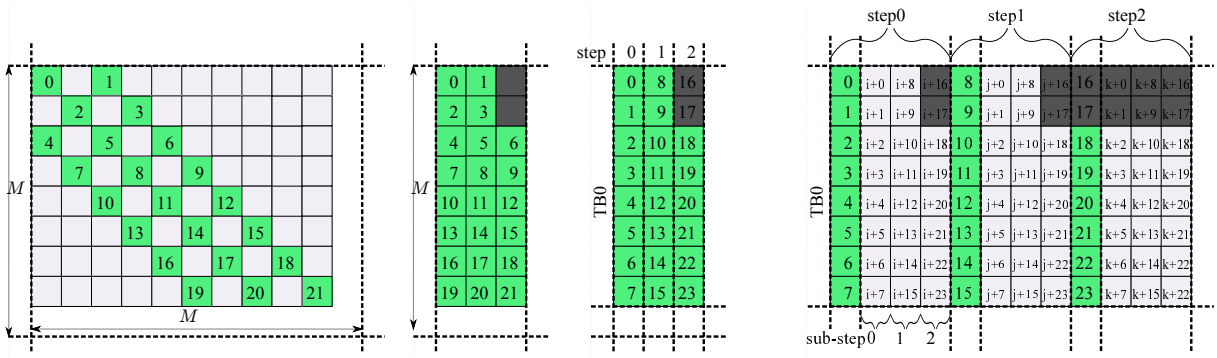
Given the local element matrices and load vectors, the global stiffness matrix of the system is assembled. Next, the system has to be solved for the unknown velocity $\boldsymbol{v}_{i+i}$. The (Jacobi-Preconditioned) CG method performs a large number of sparse matrix-vector multiplications and other vector-vector operations. Therefore, solving a large linear system efficiently, requires a fast and efficient implementation of sparse matrix-vector multiplications, which is highly-dependent on the layout used for storing the sparse matrix. Since three unknown values (components of the velocity vector) are associated to each mesh vertex, a block with $3 \times 3$ elements in the global matrix corresponds to each edge of the tetrahedral mesh. Therefore, a *Block-Compressed Sparse Row* (BCSR) format is very well suited for storing the global matrix, and thus improving the speed of the CG method.

Furthermore, since the vertex degree of internal nodes is constant in a regular tetrahedralization (see sect 5.5), the variation of the number of elements per row in the global matrix is minimal. Therefore, we use the optimized BCSR format from [VJ12]. This method efficiently stores a large sparse-matrix in BCSR format and reorders the blocks in memory to improve the efficiency of the memory transactions. This fact is very important since the main bottleneck of the CG method is the memory throughput. In [VJ12], through extensive experiments, it is shown that their optimized BCSR layout outperforms other storage formats for efficient matrix-vector multiplication on the GPU.

## 5.4 Global matrix update

Each local matrix represents a set of equations for each individual tetrahedron. To obtain the global system of equations, each component of each local matrix is added to the corresponding location of the global matrix. The location is obtained using the indices of the vertices for that specific element. Since the structure of the underlying mesh does not change during the simulation, also the structure of the global matrix remains unchanged. Therefore we assemble the global matrix only once and updates its values every time step. In this section, we propose an extension of [VJ12] which allows us to efficiently update a sparse matrix stored in the BCSR format.

For updating the global matrix, two approaches are possible. Within the first approach (*scatter*), all values of a local matrix are added to their corresponding values in the global matrix. When the local matrices are processed on the GPU, many of them are processed in parallel. Therefore, multiple threads *can* update the same




(a) Block layout of a sparse-matrix: Each green block stores $N \times N$ values and its position within the block-row. Numbers represent memory locations. Each gray block contains zero-values and is not explicitly stored. $M$ represents the dimension of the matrix.

(b) BCSR layout: Each block-row is compressed; an additional array with indices to the first block in a block row is necessary (not shown here).

(c) Blocks of consecutive block rows are mapped to a thread block ($TB0$). Blocks mapped to the same thread block are reordered so that blocks processed in the *same step* are continuous in memory. Padding is required (gray blocks).

(d) Updating matrix blocks (green), requires the associated local values. The indices of these values are stored in *index blocks* (gray), in the same order as the matrix blocks. Within each sub-step, a set of continuous index-blocks are loaded and used to fetch the corresponding values from the local matrices. The dark-gray blocks are used for padding and contain $-1$'s. $i, j, k$ represent (starting) offsets in memory.

Figure 2: Updating the sparse matrix: the initial sparse matrix is created, stored and processed, (a), (b) and (c). Updating the matrix is done by collecting the corresponding values from the local matrices, (d).

value in the global matrix at the same time, which results in *race conditions*. In order to prevent race conditions from appearing, access to the values of the global matrix would have to be serialized.

The second approach is to *gather* per element in the global matrix, the corresponding values from the local matrices. To do so, the indices of all associated local values are stored per element in the global matrix. Each index represents the position of the local value in an array $A$, which stores the values of all local matrices. Given these indices per global element value, the local values are looked-up and used to update the corresponding value in the global matrix.

Within the optimized BCSR implementation of [VJ12], the global sparse-matrix is divided in $N \times N$-sized blocks, Fig. 2(a). Next, block rows are compressed and sorted by length, Fig. 2(b). Finally, a number of consecutive block rows are grouped and mapped to a CUDA thread block. Within each group of block rows, the blocks are reordered in memory, such that accessing these blocks is performed as optimal as possible. Accessing the blocks (for e.g. a multiplication) is done as follows. First, all threads of a thread-block ($TB0$) are used to access the blocks mapped to it in the first step (step 0), see Fig. 2(c). Each thread computes an index pointing to these blocks. Next, blocks $0-7$ are loaded from the global memory. Note that these are the same blocks appearing in the first column of Fig. 2(b). For the next step, each thread increases its current index, such that the next set of blocks $(8-15)$ can be loaded (step 1). Note that all block rows must have the same length, and therefore, empty blocks must be padded (blocks 16 and 17).

To actually update the data blocks of the global matrix, we use a *gather* approach. Accordingly, $N \times N$-sized *index blocks* are used for each matrix block, see Fig. 2(d). Since the matrix blocks have a specific ordering, the same ordering is used for the index-blocks. For each step, a number of *sub-steps* is performed. Within each sub-step a set of index-blocks is loaded from memory, given a start offset ($i$, $j$ or $k$ in Fig. 2(d)). Then, for each index-block, its $N \times N$ values (indices) are used to fetch the corresponding $N \times N$ data values from local matrices, stored in global memory. Please note that the $N \times N$ data values fetched using one $N \times N$ index-block, do not come, in general, from the same local matrices. To accumulate the local contributions, an array (stored in shared memory) is used. If an index has value $-1$, no update is performed. For the next sub-step, the indices pointing to the index blocks are increased. Therefore, per step, the number of index blocks for each processed matrix block must be equal, which requires padding with '$-1$' index blocks. The advantage of this approach is that loading the indices and writing the updated values always result in an optimal throughput. Loading the actual local-element values is in general not optimal.

## 5.5 Tetrahedralization and rendering

The quality of the tetrahedral mesh is essential for efficiently simulating a deforming elastic object represented by a polygonal mesh. We have experimented with tetrahedralizations in which the surface mesh forms the outer boundary of the tetrahedral mesh. Since the tri-

angles of the surface mesh can have a high variety in size, the generated tetrahedralization also contains tetrahedral elements with a high variation in size and configuration. This can have a negative effect on the quality of the tetrahedralization. Therefore, we chose to create a tetrahedral mesh, using equi-sized elements, which however, may result in a rather rough approximation of the original surface mesh. We tackle this problem by coupling the input polygonal mesh to the (deforming) tetrahedral mesh, as follows.

First, a regular 3D grid of $N^3$ voxels is created, in which each voxel containing a part of the surface is marked as important; typical values for $N$ are $32, 64$ or 128. Next, a regular tetrahedralization of the grid is created using equi-sized tetrahedral elements, and each element containing at least one important vertex of the grid, is stored. Further, the inner volume of the object is tetrahedralized using the same equi-sized tetrahedral elements. Next, in order to reduce the amount of elements, those elements belonging to the inner volume are merged together into fewer larger ones. This reduces the total amount of elements and thus the total computation time. Note however that this approach is most useful with models which have large internal volumes, similar to the bunny in Figure 5. Finally, the original surface mesh is coupled with the tetrahedral one similar to [MG04]: each vertex in the original surface mesh is mapped to exactly one tetrahedron, and its barycentric coordinates in that tetrahedron are stored along with the vertex coordinates.

When the new positions of the tetrahedra are computed, the surface mesh is also updated. To compute new positions of the deformed surface mesh, for each vertex of the input mesh, the positions of the four vertices of the corresponding tetrahedron are looked-up and interpolated using the barycentric coordinates of the original vertex.

## 6 RESULTS

All experiments have been performed on a machine equipped with an Intel Q6600 quad-core processor and a GeForce GTX 570 with 1.2 Gb of memory.

Figure 3 shows the performances obtained for computing the local element matrices (*Matrix*), the rotation matrices (*Rotation*), solving the resulting linear system (*CG*), performing a single SpMV (*SpMV*), and the total performance (*Total*) as a function of the number of elements. *Steps/sec* is the corresponding number of simulation steps performed per second. Similarly, Figure 4 shows the computation time per simulation time-step. For each model, we have used the following material parameters: Young's modulus of elasticity, $E = 5 \times 10^5 N/m^2$; Poisson's ratio, $\mu = 0.2$; density, $\rho = 1000 KG/m^3$. Furthermore, the time step of the simulation $\Delta t = 0.001$ and the volume of each initial element $v_e = 1.65 \times 10^{-6} m^3$. Each model used in this paper is scaled such that each dimension is at most 66 *cm* and is tetrahedralized as described in Section 5.5. With these settings, the CG solver found a solution for each model in 5 to 18 iterations. In order to obtain a generic performance picture, we have fixed the number of iterations to 18, which resulted in the performances from Fig. 3.

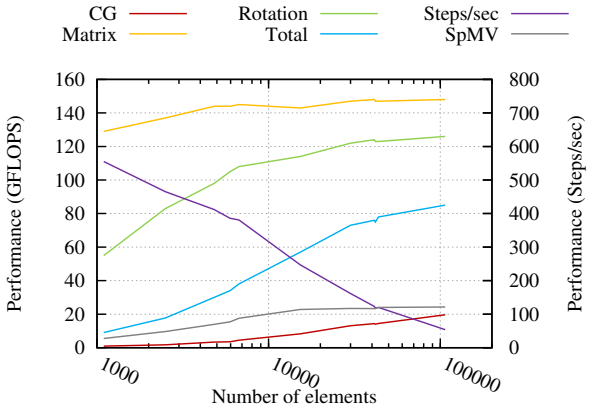



Figure 3: Performance results with different numbers of elements. *CG* represents the performance of the CG solver, *Matrix* – the performance for computing the local element matrices, *Rotation* – the performance of the rotation extraction procedure, *SpMV* – the performance of the SpMV operation; *Total* represents the overall performance. *Steps/sec* represents the number of simulation steps per second. The global-matrix update was performed with an effective throughput of 50 GB/sec.

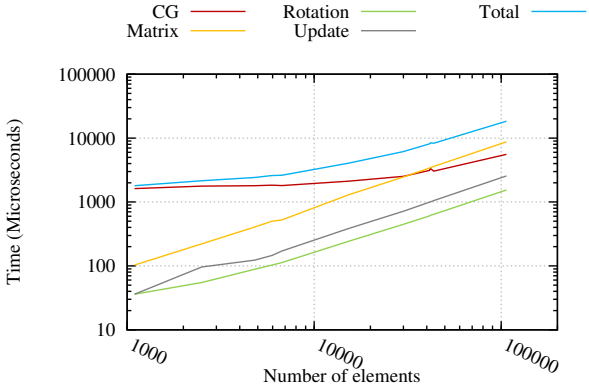



Figure 4: Timing results with different numbers of elements, per time step. *CG* represents the time of the CG solver, *Matrix* – the time for computing the local element matrices, *Rotation* – the time of the rotation extraction procedure; *Total* represents the total elapsed time per time-step.

Within Figure 3 a number of interesting patterns can be seen. First, the performance for computing the local element matrices reaches its maximum very soon. Since each matrix is mapped to exactly one thread-

block, a large amount of thread-blocks is created, resulting in a 'constant' performance. Second, the performance figures for computing the rotation matrices show a larger variation. Since 16 rotation matrices are processed by one thread-block, a significantly smaller amount of thread-blocks is used. Finally, the performance of the CG method seems to be low compared to the other operations. The CG method operates on a global sparse-matrix and performs a large number of sparse-matrix vector multiplications (SPMVs) and vector-vector operations, for which the performances are mainly bound by the memory throughput. However, the CG performances from Figure 3 agree with those from [VJ12], given the dimensions of the problem.

The measured, effective throughput for updating the global matrix was about 50 GB/sec, in all cases with more than $5k$ elements. Since this operation transfers a large amount of data, the memory bus is saturated very soon, resulting in a good throughput. However, since not all transactions can be coalesced, the maximum throughput is not reached. This operation is very similar to an SPMV with $1 \times 1$ blocks, but now for a matrix containing $d\times$ more elements, with $d$ the degree of internal nodes in the model. This observation shows that the measured throughput is close to the expected one, according to the results in [VJ12].

As expected, the total performance increases with the number of elements. This shows that the computational resources are used efficiently for larger models. The number of elements, for which the maximum performance is reached, depends on the actual GPU mapping of the computations. For example, the CG solver does not reach its maximum performance for $100k$ elements, while the computation of the local element matrices reaches its peak at $5k$ elements. Due to this, one can expect better performances for the CG method when larger models are used. Furthermore, for models having less than $30k$ elements, the total computation is dominated by the time spent by the CG solver. For larger models, more time is spent on computing the local matrices, see Figure 4.

The measured overall performance is based on the total time needed per simulation step, which includes all operations performed, except the rendering of the model. Figure 3 also shows the number of simulation steps performed per second, given the number of elements; these numbers are based on the total computation time. Accordingly, even for large models, interactive frame rates can be reached. A rough comparison of the obtained performance and frame rate with other state-of-the-art multigrid GPU implementations [DGW11] shows that, even if in theory the CG method converges slower than multigrid, comparable results *can* be obtained for similar models. We assume that memory transactions in our method are more efficient, despite of transferring more data. However, more research is required to get a full understanding of the differences between both methods performed on modern GPUs, with respect to performance figures. Finally, Figures 1, 5, 6, 7, 8 and 9 show example results from our simulations.

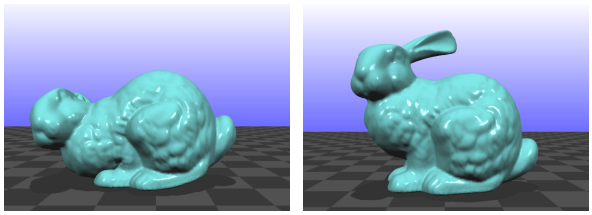


Figure 5: Material properties and collision handling. *Left*: flexible material ($E = 5 \times 10^4$). *Right*: stiffer material ($E = 5 \times 10^5$). Simulation rate: 120 frames per second.

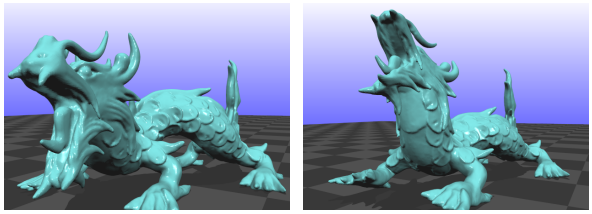


Figure 6: Left: stretching and deforming a model using external forces. Right: deformation after releasing external forces. Simulation rate: 118 frames per second.

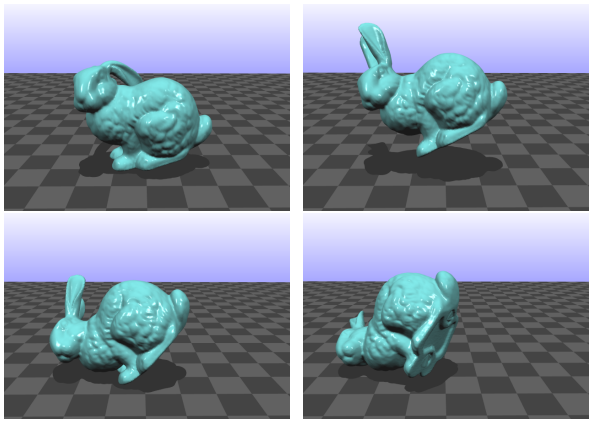


Figure 7: Bunny bouncing on the floor. Simulation rate: 120 frames per second.

## 7 CONCLUSIONS

We have presented an efficient method for simulating elastically deformable models for graphics applications, accelerated on modern GPUs using CUDA. Our method relies on a fast Conjugate Gradient solver and an efficient mapping of the SPMV operation on modern GPUs [VJ12]. Since the topology of the underlying grid

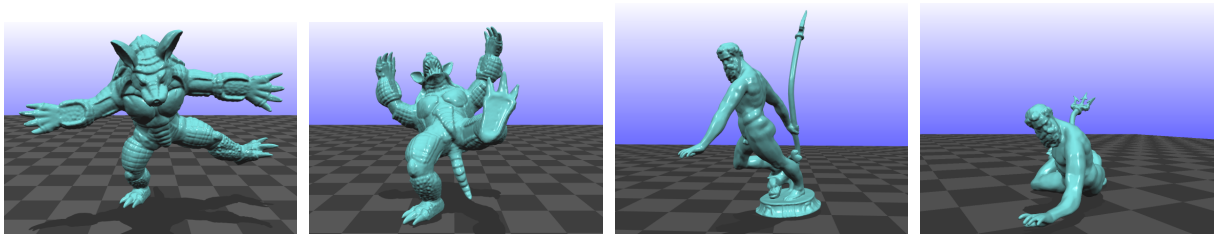


Figure 9: Other simulation results. Simulation rate: 160 frames per second.

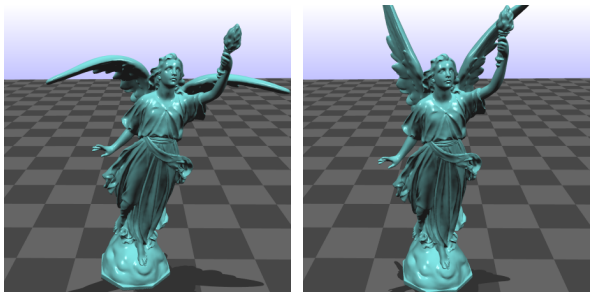


Figure 8: Left: applying external forces on the wings. Right: after releasing the external forces. Simulation rate: 116 frames per second.

does not change during the simulation, data structures are reused for higher efficiency. To further improve the performance, we proposed a scheme which allows to efficiently update the sparse matrix, during the simulation.

In future work we will investigate the performance of this method when multiple GPUs are used. Furthermore, we will investigate the performance difference between traditional CG methods and multigrid methods performed on modern GPUs. Also, we plan to enhance the simulation to allow for plastic behaviour as well as brittle and fracture of stiff materials.

## REFERENCES


[BCL07] BUATOIS L., CAUMON G., LÉVY B.: Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Perf. Comp. Conf. (HPCC)* (2007). 2

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *Proc. SIGGRAPH'03* (2003). 2

[BG08] BELL N., GARLAND M.: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. Tech. Rep. NVR-2008-004, NVidia, 2008. 2

[DGW11] DICK C., GEORGII J., WESTERMANN R.: A real-time multigrid finite hexahedra method for elasticity simulation using CUDA. *Simulation Modelling Practice and Theory 19*, 2 (2011), 801–816. 2, 7

[GM97] GIBSON S., MIRTICH B.: *A Survey of Deformable Modeling in Computer Graphics*. Tech. Rep. TR-97-19, MERL, Cambridge, MA, 1997. 1

[GSMY*07] GÖDDEKE D., STRZODKA R., MOHD-YUSOF J., MCCORMICK P., BUIJSSEN S. H., GRAJEWSKI M., TUREK S.: Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing 33*, 10–11 (2007), 685–699. 2

[GST05] GÖDDEKE D., STRZODKA R., TUREK S.: Accelerating double precision FEM simulations with GPUs. In *Proc. ASIM 2005 - 18th Symp. on Simul. Technique* (2005). 2

[GST07] GÖDDEKE D., STRZODKA R., TUREK S.: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *Int. Journal of Parallel, Emergent and Distributed Systems 22*, 4 (2007), 221–256. 2

[GW06] GEORGII J., WESTERMANN R.: A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics 30*, 3 (2006), 408–415. 2

[Hig86] HIGHAM N. J.: Computing the polar decomposition – with applications. *SIAM Journal of Scientific and Statistical Computing 7* (1986), 1160–1174. 3, 4

[HS04] HAUTH M., STRASSER W.: Corotational simulation of deformable solids. In *WSCG* (2004), pp. 137–144. 2, 3, 4

[ITF06] IRVING G., TERAN J., FEDKIW R.: Tetrahedral and hexahedral invertible finite elements. *Graph. Models 68*, 2 (2006), 66–89. 2, 3

[JP99] JAMES D. L., PAI D. K.: ArtDefo: accurate real time deformable objects. In *Proc. SIGGRAPH'99* (1999), pp. 65–72.

1

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for gpu implementation of numerical algorithms. In *Proc. SIGGRAPH'03* (2003), pp. 908–916. 2

[LJWD08] LIU Y., JIAO S., WU W., DE S.: Gpu accelerated fast fem deformation simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on* (30 2008-dec. 3 2008), pp. 606 –609. 2

[MG04] MÜLLER M., GROSS M.: Interactive virtual materials. In *Proc. Graphics Interface 2004* (2004), pp. 239–246. 2, 3, 4, 6

[NMK*06] NEALEN A., MÜLLER M., KEISER R., BOXERMANN E., CARLSON M.: Physically based deformable models in computer graphics. *Computer Graphics Forum 25* (2006), 809–836. 1

[NVI] NVIDIA CORPORATION: *Compute Unified Device Architecture programming guide*. Available at `http://developer.nvidia.com/cuda`. 4

[PH05] PEPPER D. W., HEINRICH J. C.: *The Finite Element Method: Basic Concepts and Applications*. Taylor and Francis, 2005. 1, 2, 3

[RS01] RUMPF M., STRZODKA R.: Using graphics cards for quantized FEM computations. In *Proc. IASTED Vis., Imaging and Image Proc.* (2001), pp. 193–202. 2

[TBHF03] TERAN J., BLEMKER S., HING V. N. T., FEDKIW R.: Finite volume methods for the simulation of skeletal muscle. In *In SIGGRAPH/Eurographics symposium on Computer Animation* (2003), pp. 68–74. 1

[TPBF87] TERZOPOULOS D., PLATT J., BARR A., FLEISCHER K.: Elastically deformable models. In *Proc. SIGGRAPH'87* (1987), pp. 205–214. 1

[VJ12] VERSCHOOR M., JALBA A. C.: Analysis and performance estimation of the conjugate gradient method on multiple gpus. *Parallel Computing* (2012). (in press). 1, 2, 4, 5, 7