Volume 0 (1981), Number 0 pp. 1–12 COMPUTER GRAPHICS *forum*

Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications

Daniel Weber[1], Jan Bender[2], Markus Schnoes[3], André Stork[1,3] and Dieter Fellner[1,3]

[1]Fraunhofer IGD, Germany
[2]Graduate School CE, TU Darmstadt, Germany
[3]TU Darmstadt, Germany

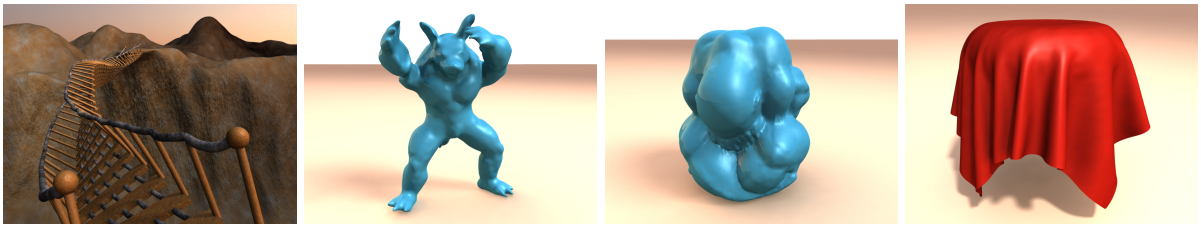

Figure 1: *Interactive simulations with our novel GPU-based solver. (From left to right) Simulations of complex articulated bodies; volumetric deformation with 10K quadratic finite elements and 47K degrees of freedom; volumetric deformation with 65K linear finite elements and 38K degrees of freedom and highly-detailed cloth models.*

Abstract

We present GPU data structures and algorithms to efficiently solve sparse linear systems which are typically required in simulations of multibody systems and deformable bodies. Thereby, we introduce an efficient sparse matrix data structure that can handle arbitrary sparsity patterns and outperforms current state-of-the-art implementations for sparse matrix vector multiplication. Moreover, an efficient method to construct global matrices on the GPU is presented where hundreds of thousands of individual element contributions are assembled in a few milliseconds.

A finite element based method for the simulation of deformable solids as well as an impulse-based method for rigid bodies are introduced in order to demonstrate the advantages of the novel data structures and algorithms. These applications share the characteristic that a major computational effort consists of building and solving systems of linear equations in every time step. Our solving method results in a speed-up factor of up to 13 in comparison to other GPU methods.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.1]: Hardware Architecture—Graphics processors; Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Animation

1. Introduction

Simulating rigid bodies and deformable models is still a challenging task in computer graphics. Several algorithms have been presented to produce realistic physically-based animations, e.g. the finite element method.

In this paper we focus on interactive simulation where the computation time for a single time step is limited. By interactivity we mean that we want to

achieve at least ten simulation steps per second. Realistic deformable models often contain stiff components which lead to stiff differential equations. For example, even low elastic coefficients in finite element simulations force a limitation of the time step size. Many approaches use an implicit time integration which is unconditionally stable in order to solve the stability problems with explicit integration methods. Articulated rigid bodies are often simulated using a Lagrange multiplier or an impulse-based method. Hence, most simulation methods for deformable models and articulated rigid bodies must solve a huge system of linear equations where the corresponding system matrices have to be updated in every time step. These linear systems are typically sparse, symmetric and positive definite, so a conjugate gradient solver can be employed. In general, building and solving linear systems requires a major computational effort in each step. Therefore, a fast solver is an essential component for interactive simulations.

Computer graphics processors (GPUs) with their high computational throughput theoretically offer a tremendous speed-up by offering the possibility to execute special GPU-programs also known as kernels. In the past years several GPU-based solvers have been presented in order to accelerate the solution of linear systems. For example many fluid dynamic applications in computer graphics can be significantly sped up by those methods, since there the matrix is constant during the simulation. However, for deformable and articulated bodies the matrix changes in each step. Hence, the matrix construction must also run fast to make an efficient simulation possible. Furthermore, it is desirable to be able to control the velocity and position change of certain elements, e.g. for collision handling. Existing GPU solvers provide neither an efficient update of the linear system nor a possibility to control velocities or positions.

In this paper we present a novel GPU-based solver which allows a fast update of sparse matrix structures and provides velocity and position control. Furthermore, we introduce a novel data structure that accelerates the sparse matrix vector product (SpMV) and therefore the solution of sparse linear systems significantly. In contrast to existing approaches, our goal is to minimize the number of kernel calls needed, since in interactive applications the kernel launch overhead can influence the performance significantly. Due to an optimized preconditioned conjugate gradient (PCG) algorithm the solver outperforms current state-of-the-art solvers considerably. The presented method permits real-time simulations of very complex deformable and articulated bodies. In order to demonstrate the performance gain in practice, we will present simulations for deformable bodies using linear and quadratic finite elements, cloth simulations and simulations of articulated bodies. Finally, we make a comparison with current state-of-the-art solvers and observe a performance gain roughly by a factor of 13 at maximum.

Our contributions:

- A novel GPU-based PCG algorithm with position and velocity control which is designed for the use in dynamics simulations. It is optimized by minimizing the number of kernel calls.
- A novel GPU data structure for arbitrary sparsity patterns that shows beyond state-of-the-art performance for matrix vector products.
- An algorithm that efficiently updates sparse matrices on the GPU.

2. Related Work

Early works by Bolz et al. [BFGS03] as well as Krüger and Westermann [KW03] used graphics hardware to speed up computations for solving systems of linear equations. Both used the graphics pipeline with programmable shaders and textures to represent operations and the data, respectively. With the introduction of NVIDIA's Compute Unified Device Architecture [NVI11a] to abstract the underlying graphics hardware, general purpose programming on graphics processing units (GPU) has caused an increasing interest. Among a lot of work for dense matrices, a few methods have been presented to handle sparse matrices. In particular, the library CUSparse [NVI11b] has been released providing GPU-accelerated BLAS operations and sparse matrix vector multiplications (SpMV). The work of Bell et al. [BG08,BG09] presents different optimized data structures for sparse matrices. There, the performance of well-known formats like diagonal (DIA), ELLPACK (ELL), compressed sparse row (CSR) and the coordinate (COO) on GPU hardware is analyzed. Furthermore, a hybrid format (HYP) is introduced which combines the COO and the ELL format for sparse matrices with a varying number of non-zero entries per row. The open source library CUSP [BG10] is based on this work and clearly exceeds the performance of the CUSparse library for matrices with highly varying row lengths. Another approach has been presented by Buatois et al. [BCL09] where the clustering of non-zero entries in the sparse matrix is identified to permit register blocking and optimized memory fetches. Baskaran et al. [BB09] focus on coalesced loading of sparse matrix and data reuse and report a further acceleration of SpMV operations with matrices in CSR-format. Vazquez et al. [VOFG10] propose an optimization of the ELL-format for GPUs called ELLPACK-R by additionally storing the row length. This approach is generalized by the Slided ELLPACK format of Monakov et

al. [MLA10] grouping rows together. But for both data structures the SpMV operation needs additional kernel calls for a subsequent reduction. Recently, Oberhuber et al. [OSV11] introduced the row-grouped CSR format, a data structure similar to our proposed approach, but they do not cluster non-zero elements. In [MCG04] the L-CSR format is introduced in order to perform an efficient sparse matrix-vector product for matrices with very short row lengths by using an unroll-and-jam approach.

Simulating deformable bodies with finite elements has been pioneered by Terzopoulus et al. [TW88]. They use the Green's strain tensor that is invariant w.r.t. rigid body transformations. Different approaches to simplify the governing equations have been presented. O'Brien et al. [OH99] used finite elements with explicit time stepping and lumped mass matrices. Müller et al. [MG04] presented a co-rotational approach using the linear Cauchy's strain tensor with implicit time stepping for unconditional stability. Georgii et al. [GW08] presented a multigrid algorithm for co-rotational mesh hierarchies. Finite elements with quadratic basis functions for deformation simulation have been introduced by Mezger et al. [MTPS08]. Recently, Weber et al. [WKS*11] presented an optimized quadratic finite element simulation with basis functions in Bernstein-Bézier form. For a stable simulation often an implicit integration method is used which requires the solution of a linear system in each step. Therefore, an efficient solver is essential for real-time simulations.

A GPU-based simulation of elastic bodies has been presented by Dick et al. [DGW11]. They use a lattice to discretize the simulated geometry which is very well suited for multigrid algorithms. Lately, Allard et al. [ACF11] presented a GPU-based approach to simulate deformable models with tetrahedral elements. They propose to emulate the sparse matrix vector multiplication by splitting the computations down to the individual element contributions. This is a reasonable approach when the time required to generate the matrices is higher than the performance loss that is induced by every matrix vector multiplication. In [CA09] a parallel Gauss-Seidel algorithm is presented to solve a dense linear complementarity problem on multi-core CPUs or GPUs. This is used for an efficient contact handling between deformable bodies.

An articulated body is a system of rigid bodies which are connected by joints. Joint simulation can be performed by using a reduced (or generalized) coordinate formulation to describe the state of an articulated body [Fea07]. The formulation in maximal coordinates, which uses all coordinates of the body, is also common in computer graphics. The Lagrange

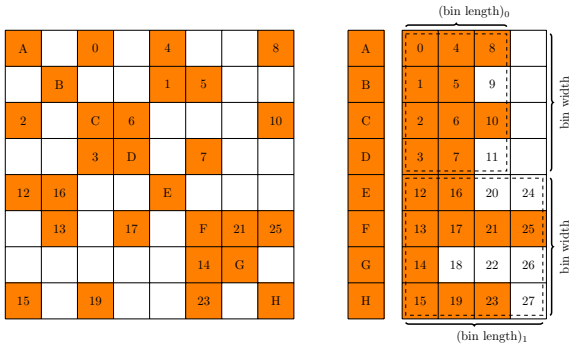

Figure 2: *(Left) Sample of a sparse matrix. Colored squares represent non-zero entries. The numbers indicate the order in memory in our data structure and letters represent the diagonal. (Right) Conceptual data structure with separated diagonal and off-diagonal part. Blank squares with numbers represent padded values. Bin borders are indicated by dashed lines.*

multiplier method is a popular example [Bar96]. However, the method has a drifting problem which must be solved by an additional stabilization [CP03]. Impulse-based methods prevent a constraint violation by adding impulses to the system. These methods avoid the stabilization problem by using a preview of the joint state for the computation of the constraint impulses [BS06, WTF06]. To compute the joint forces or impulses, a system of linear equations must be solved which is one of the most time-consuming parts in each simulation step. In this area the parallel sparse linear solver PARDISO [SG04] is used in different works.

3. BIN-CSR data structure

In this section, we present a new GPU data structure that performs an efficient matrix-vector product

$$\mathbf{y} \leftarrow \mathbf{A}\mathbf{x}.$$

Here, the matrix $\mathbf{A}$ is sparse, i.e., there are only a few numbers of non-zero entries. Furthermore, the amount of non-zero entries per row varies across the matrix. In the simulation applications presented in this paper the dimension of the vectors $\mathbf{x}$, $\mathbf{y}$ and the matrix $\mathbf{A}$ corresponds to the number of degrees of freedom.

Our data structure, the BIN-CSR format, is highly optimized for fast sparse matrix-vector products (SpMV) on GPUs. It uses a combination of a compressed sparse row storage (CSR) and a partitioning scheme grouping the matrix entries into bins. We implemented the data structure and the arithmetic operations using CUDA [NVI11a]. In general,

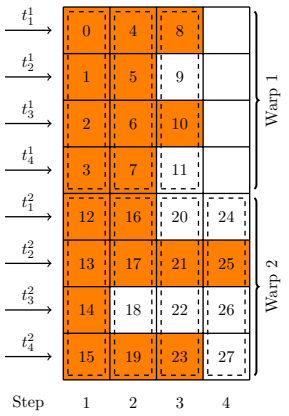

Figure 3: *The off-diagonal elements of the matrix in Fig. 2 can be loaded in one coalesced memory for each step. One memory request for the thread t_i^j $(i = 0, \ldots, 3)$ in a warp j are indicated by dashed lines.*

SpMV operations for GPUs are memory bandwidth bound [BG09]. Therefore, it is very important to provide a GPU friendly memory access pattern in order to achieve an optimal throughput and high performance. We designed the data structure in such a way that each thread processes one row, i.e., each thread computes one value in the vector $\mathbf{y}$ for a SpMV operation. Similar to Oberhuber et al. [OSV11] we store the values with a specific order, so that the threads can access the data coalesced. We therefore introduce the concept of a *bin* which is a portion of the matrix that a group of threads accesses concurrently (see Fig. 2). In order to achieve optimal performance the width of such a bin should correspond to the number of threads that run concurrently on a multiprocessor. On Fermi architectures (compute capability 2.0) this is the size of a warp (32) or the size of a half warp (16) on older hardware (compute capability 1.X). In our examples and figures we use a bin width of four for the sake of simplicity. In contrast to the approach of Oberhuber et al. [OSV11], we do not store this bin width explicitly saving memory bandwidth.

Fig. 2 (right) shows the conceptual layout of our BIN-CSR data structure. The diagonal is stored separately allowing an efficient Jacobi preconditioner. The matrix is compressed, i.e., only non-zero entries are stored with their corresponding column indices. These indices are stored in a separate array but in the same memory layout. The data structure therefore consists of four arrays: *diagonal*, *data*, *offset* and *colIndices*. In contrast to existing CSR formats, the order of storage of the non-zero values is adapted, so that entries in the same compressed column are placed next to each other. Fig. 3 shows the resulting off-diagonal matrix entries in memory. Then, loading the matrix entries in a kernel can be done very efficiently: All threads in a warp can coalesce their memory request for each matrix column with only one memory fetch. In order to construct such a layout, we need all rows in each bin to have the same length. We therefore determine the maximum row length per bin:

$$\text{bin length}_i = \max_{j \in \text{bin}_i} \text{length}(\text{row}_j).$$

For each bin i we then allocate $(\text{bin length}_i) * (\text{bin width})$ elements for the data and the column indices and pad the remaining data to meet the memory alignment. Furthermore, the offsets for each bin and for each row are computed and stored to provide access to the start of the non-zero entries for each row. The subsequent data can be obtained by incrementing the index by bin width elements. As in the CSR format, the data structure is then represented by the non-zero matrix data, the row offsets and the column indices.

A further extension of this data structure is motivated due to the characteristics of our applications. There, the global matrices exhibit the property that the non-zero entries only arise in 3×3 blocks. Then the number of memory reads for the column indices can be reduced to further speed-up the operations. Note that in rigid body simulation this assumption cannot be used for all kind of constraints. Our extension, the BIN-BCSR format (BIN - block compressed sparse row) is conceptually similar to the work of Buatois et al. [BCL09] who also propose to group non-zero entries in blocks. They determine clusters of 2×2 or 4×4 blocks of non-zero entries in a precomputation step to permit register blocking and maximizing the memory fetch bandwidth. Then, the processing of 2 or 4 rows, respectively, needs to be merged in one thread. But this considerably reduces the number of threads that are launched and may result in a low utilization of the multiprocessors. In contrast, we use our 3×3 blocks that only consist of non-zero entries and start one thread per single row. This keeps the number of threads high resulting in a higher occupancy. Exploiting the properties of the matrices, we can reduce the number of column indices that need to be stored and loaded, as two out of three indices are implicit.

3.1. Sparse Matrix-Vector Multiplication

Using our proposed data structure results in a very fast SpMV operation $\mathbf{y} \leftarrow \mathbf{Ax}$, since the data of the matrix can be efficiently read from memory. Threads in a warp can load the non-zero values and the column indices coalesced. In contrast to other approaches (e.g., [BCL09, BB09, VOFG10, MLA10]) this coalescing is achieved without a subsequent reduction step, since

the data is ordered for one thread per row. Furthermore, as the size of each bin equals a multiple of the bin width, the offsets for each bin and for each row are naturally aligned. Since the non-zero patterns of our matrices are strongly irregular, we use texture memory to cache the vector $\mathbf{x}$ (similar to Bell et al. [BG09] and Baskaran et al. [BB09]). Writing the results to the vector $\mathbf{y}$ is also coalesced as each thread processes one row.

Algorithm 1 SpMV operation for row and thread i

```
1: y[i] ← diagonal[i] * x[i]
2: index ← offset[i];
3: endIndex ← offset[i + bin width];
4: while index < endIndex do
5:     y[i] ← y[i] + data[index] * x[colIndices[index]];
6:     index ← index + bin width
```

Algorithm 1 shows the SpMV operation for the BIN-CSR matrix in pseudo code. The major differences compared to a CSR implementation are in line 1, 5 and 6. First, the result $y[i]$ is initialized by multiplication with the diagonal matrix element and second, the index is incremented by bin width. This results in high performance due to the minimal number of memory fetches. However, the data layout for optimizing coalesced loading leads to idle threads and increased memory consumption when the row lengths differ heavily in a bin. For the BIN-BCSR matrix data structure Algorithm 1 needs to be slightly adapted: The multiplication in the while-loop must be performed three times for each column in a block. Furthermore, the column index must be read only once at the beginning of the loop and be incremented by one for the remaining two multiplications.

Our proposed sparse matrix representation can be interpreted as a combination of the CSR and the ELLPACK (ELL) format [BG08, BG09]. The ELL format pads the data per row to the maximum row length and is therefore well suited for matrices with a nearly constant row length. As the efficiency of the ELL format degrades rapidly when the number of entries per row varies, Bell et al. [BG09] propose a hybrid (HYB) format combining ELL and coordinate (COO) data structure. It stores the matrix entries' majority in an ELL structure and exceptional long rows in a COO format. However, it is complicated to determine a good criterion that adjusts the size of the ELL portion. Furthermore, the COO format suffers from the fact that its SpMV operation is based on a computationally expensive segmented reduction scheme and therefore multiple kernels are required for a single SpMV operation.

In the work of Vazquez et al. [VOFG10] an improved version of the ELL format, the ELLPACK-R format, is introduced. It uses an additional array to store the length of each row. Thus, a very fast and efficient SpMV implementation can be achieved. But, in a scenario with only a few rows that contain large numbers of non-zero elements there is very high memory overhead for ELL-based matrix formats (e.g., for matrices of quadratic finite elements, see Table 2). Monakov et al. [MLA10] propose a generalization with the Sliced ELLPACK format by grouping S adjacent rows and storing only row lengths for these groups. Thus, the memory consumption is reduced significantly. As in the approach of Oberhuber et al. [OSV11], our data structure reorders the data and allows for varying bin lengths. This provides a good trade-off between memory and performance benefits. In the worst case, the memory consumption of our format is equal to ELL-based formats but in general it is significantly lower.

3.2. Kernel call minimization for PCG solver

When implementing a PCG algorithm one normally assembles it out of a set of single operations like SpMV, dot products, and AXPY, where the latter computes $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$. In the case of a conjugate gradient algorithm three AXPY operations, two dot products and one SpMV operation are needed in the inner loop [She94]. Using a PCG algorithm there is an additional matrix vector multiplication with the preconditioner matrix. In our case this is computationally similar to a AXPY operation since we use a simple Jacobi preconditioner that only requires the multiplication with the inverted diagonal. Additionally, there are two filter operations that can be used for constraints (see Baraff et al. [BW98]). This allows for a position and velocity control, e.g. to resolve collisions or to constrain the movement of nodes. Thus, implementing the PCG algorithm for CUDA results in up to nine subsequent kernel calls in the inner loop. As invoking of such kernels naturally generates some overhead, it is desirable to minimize the number of calls. Particularly in real-time applications the kernel call overhead can be in the magnitude of simple kernels as the AXPY operation. Moreover, the AXPY operation and similar kernels are heavily memory bandwidth bound. Merging some operations into a few kernels where possible, can increase the number of arithmetic operations per memory operation. Furthermore, it is desirable to avoid memory transfers between GPU and CPU, as they trigger a significant overhead.

Algorithm 2 shows the PCG algorithm (cf. [She94, BCL09]) to solve the linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

which may result from elastic or articulated body sim-

Algorithm 2 PCG algorithm

1: $i \leftarrow 0$; $\mathbf{r} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}$; filter($\mathbf{r}$);
2: $\mathbf{d} \leftarrow \mathbf{M}^{-1}\mathbf{r}$; $\delta \leftarrow \mathbf{r}^T\mathbf{d}$; $\delta_0 \leftarrow \delta$;
3: **while** $i \leq i_{\max}$ and $\delta > \varepsilon^2\delta_0$ **do**
4: $\quad \delta_{\text{old}} \leftarrow \delta$; $\mathbf{q} \leftarrow \mathbf{A}\mathbf{d}$; filter($\mathbf{q}$); $\alpha \leftarrow \frac{\delta}{\mathbf{d}^T\mathbf{q}}$;
5: $\quad \mathbf{x} \leftarrow \mathbf{x} + \alpha\mathbf{d}$; $\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q}$; $\mathbf{s} \leftarrow \mathbf{M}^{-1}\mathbf{r}$; $\delta \leftarrow \mathbf{r}^T\mathbf{s}$;
6: $\quad \beta \leftarrow \frac{\delta}{\delta_{\text{old}}}$; $\mathbf{d} \leftarrow \mathbf{r} + \beta\mathbf{d}$; $i \leftarrow i + 1$;

ulation (see Equ. 3 or Equ. 6). Here, $\mathbf{M}$ is an appropriate preconditioner for $\mathbf{A}$ and $\mathbf{x}$ is initialized with a first guess. In our applications the result from the last time step is used as a warm start for faster convergence. The *filter* operation allows for partially constraining values in the $\mathbf{x}$ vector that may be used for velocity correction (see [BW98] for details). In our merged conjugate gradient (MCG) approach we combine the initialization operations (in lines 1 and 2) as well as the operations in each line of the while-loop each into one kernel. Merging the AXPY and SpMV-operations is simple, as our SpMV implementation can be executed per row. However, to incorporate the dot product, we need a special synchronization mechanism since we need its result (e.g., α, β and δ) globally in all threads.

Generally, our dot product computation is a reduction that consists of two phases: in the first kernel each block computes an intermediate result by partially reducing the associated data in shared memory. In the second kernel each block loads all the intermediate data and again performs a reduction (see Fig. 4). So, the second reduction is computed redundantly by each block, but its result is then available in all blocks. However, for extremely large linear systems the number of required blocks can get high so that the second reduction slows down. But in these cases the performance gain of merging the kernels for solving the linear system would be low.

In order to minimize the overall number of kernel calls, the reduction is combined with the remaining computations (e.g. AXPY). As there are two dot products and the SpMV operation in the inner loop of each CG iteration that needs synchronization, the number of kernel calls can be reduced to three.

3.3. GPU matrix construction and update

An efficient matrix construction is crucial for applications, where the linear system changes in every iteration step. In our applications, the update of every non-zero entry is computed by summation

$$A_{ij} = \sum_{k \in \Gamma_{ij}} A^e_k, \tag{2}$$

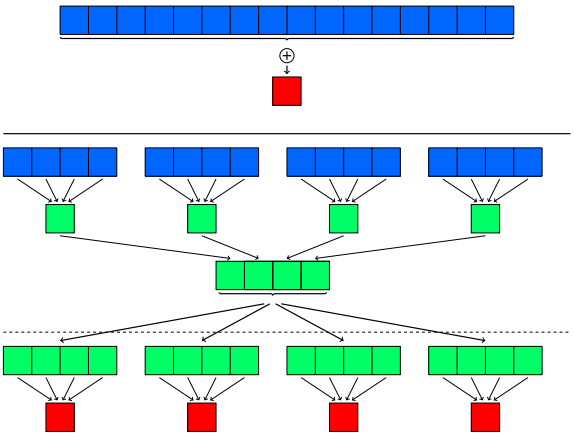

Figure 4: *Upper part: Simplified example for a reduction of a 16 element input vector in blue and its result in red. Lower part: The reduction is computed in four different blocks. The first kernel (above dashed line) partially reduces the data and stores it in global memory (green). As the final result is needed in all blocks, the second reduction is done redundantly in the second kernel (below dashed line).*

as a matrix entry is influenced by a few local element contributions A^e_k. We store the element entries A^e_k linearly in memory. In a precomputation step, for every global entry A_{ij} the indices k referring to the positions in the element array are collected in the list of arrays Γ_{ij}. We store all the indices linearly (*eIndices*) and use an array (*eOffset*) for pointing to the start of a global entry. Algorithm 3 illustrates the assembly of the global matrix for one thread i that computes the i-th row. The access for writing the global matrix is the same as in the SpMV kernel and is therefore coalesced. The access pattern for reading the element matrix entries is irregular, so we use the texture cache to speed it up.

Algorithm 3 Matrix assembly for row and thread i

1: index $\leftarrow$ offset[i];
2: endIndex $\leftarrow$ offset[i + bin width];
3: **while** index $<$ endIndex **do**
4: $\quad$ data[index] $= 0$
5: $\quad$ eStart $\leftarrow$ eOffset[index];
6: $\quad$ eEnd $\leftarrow$ eOffset[index + 1];
7: $\quad$ **for** l $=$ eStart $\rightarrow$ eEnd $- 1$ **do**
8: $\quad\quad$ k $=$ [eIndices[l]];
9: $\quad\quad$ data[index] $\leftarrow$ data[index] $+ A^e_k$;
10: $\quad$ index $\leftarrow$ index + bin width

In a typical application, where the global matrix

changes in every iteration, the algorithm works as follows. First, all element matrix entries are computed on the GPU. As the updates of these element matrices are independent from each other, a parallelization is trivial. We use one thread per element matrix entry A_k^e to recompute the necessary data. Afterwards, the global matrix is reassembled using Algorithm 3 with one thread per single row. Finally, the system of linear equations is set up and solved on the GPU using Algorithm 2.

4. Simulation applications

In this section we describe two different simulation applications that greatly benefit from our GPU data structures. In both applications there is a system of linear equations that needs to be solved in every simulation step. The corresponding matrices are huge, symmetric and positive definite and have varying row sizes. Furthermore, the entries in the matrices change in every simulation step, whereas the matrix structures stay the same.

4.1. Elasticity simulation using FEM

For a realistic simulation of elastic bodies several approaches have been presented that discretize the partial differential equations of elasticity using the finite element method. In our application we use finite elements with linear and quadratic basis functions, i.e. with 4 and 10 degrees of freedom per tetrahedron, respectively. There, a tetrahedral mesh discretization of the object is used and a co-rotational formulation is employed [WKS*11]. Applying the finite element discretization results in a set of ordinary differential equations

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}_{\text{ext}}.$$

Here, $\mathbf{M}$ and $\mathbf{K}$ are the mass and stiffness matrices, respectively. The discretized displacement field $\mathbf{u} = \mathbf{x} - \mathbf{x}_0$ needs to be computed taking external and inertial forces $\mathbf{f}_{\text{ext}}$ and $\mathbf{M}\ddot{\mathbf{u}}$, respectively, into account. Employing an implicit time integration (see Baraff et al. [BW98]) results in a system of linear equations

$$\left(\mathbf{M} + \Delta t^2 \mathbf{K}\right) \Delta \mathbf{v} = \Delta t \left(\mathbf{f} + \Delta t \mathbf{K}\mathbf{v}\right) \tag{3}$$

where the resulting force $\mathbf{f}$ is computed by

$$\mathbf{f} = \mathbf{f}_{\text{ext}} + \mathbf{f}_{\text{el}} = \mathbf{f}_{\text{ext}} + \mathbf{K}\mathbf{u}. \tag{4}$$

Note that the time integration introduces numerical viscosity, so we do not consider damping forces explicitly. Employing co-rotation requires computing the rotation $\mathbf{R}^e$ for each element e by a polar decomposition (see [MG04]) and updating the element stiffness matrices $\bar{\mathbf{K}}^e = (\mathbf{R}^e)^T \bar{\mathbf{K}}^e \mathbf{R}^e$ in every time step. Therefore, the linear system of Equ. 3 with the global stiffness matrix $\bar{\mathbf{K}}$ has to be reconstructed in every step. The system matrix $\mathbf{A} = \mathbf{M} + \Delta t^2 \bar{\mathbf{K}}$ is sparse, i.e., there are only non-zero 3×3-blocks when two degrees of freedom share a tetrahedron. Using this formulation, we additionally need to replace the elastic force $\mathbf{f}_{\text{el}}$ in Equ. 4 with the co-rotated force $\mathbf{f}^e_{\text{rot}} = \bar{\mathbf{K}}^e \mathbf{x} - (\mathbf{R}^e)^T \mathbf{x}_0$ that needs to be summed up over all elements e. By solving Equ. 3 we obtain the velocity changes $\Delta\mathbf{v}$ that are used to update the state of the tetrahedral mesh

$$\mathbf{x} = \mathbf{x} + \Delta t(\mathbf{v} + \Delta \mathbf{v}).$$

4.2. Simulation of articulated bodies

Articulated bodies introduce holonomic constraints of the form $\mathbf{C}(\mathbf{x}, t) = \mathbf{0}$ to a system of rigid bodies in order to simulate joints. We want to simulate these constraints by using an impulse-based approach similar to [BS06] and [WTF06]. Therefore, we bring all constraints in a general form $\mathbf{J}\mathbf{v} + \mathbf{c} = \mathbf{0}$ by differentiating the constraint function $\mathbf{C}$ w.r.t. time.

Analogous to the Lagrange multiplier method, we could compute the magnitudes λ of the impulses that are required to simulate a constraint by solving

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\lambda = -\mathbf{J}\mathbf{M}^{-1}\mathbf{p}_{\text{ext}} - \mathbf{c} \tag{5}$$

where the vector $\mathbf{p}_{\text{ext}}$ contains impulses and angular momenta which correspond to the external forces and torques in one time step. In this way the impulses are computed in order to compensate the influence of external forces and torques. If we use this approach, we need an additional stabilization method to prevent the joints from breaking up [CP03].

Instead of this we want to use a prediction of the constraint state in order to compute the required impulses which solves the stabilization problem [BS06]. The prediction $\mathbf{C}(\tilde{\mathbf{x}}, t + \Delta t)$ for a constraint is determined by solving the unconstrained equations of motion which gives us a prediction $\tilde{\mathbf{x}}$ of the positions. Now, impulses are required that change the velocities of the bodies at time t so that the constraint is fulfilled after a time step of size Δt. This means that $\mathbf{C}(\mathbf{x}, t + \Delta t) = \mathbf{0}$ must hold.

To get the required impulses a nonlinear equation must be solved since the relative motion of the bodies is generally nonlinear. Weinstein et al. use Newton iteration to get the solution of this equation [WTF06]. In contrast to that, we approximate the required velocity change by assuming a linear motion which results in $\Delta\mathbf{v} = 1/\Delta t \cdot \mathbf{C}(\tilde{\mathbf{x}}, t + \Delta t)$. Thanks to this linearization of the problem we can efficiently compute the impulses by solving a system of linear equations. We get the required system

$$\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T\lambda = \Delta\mathbf{v} \tag{6}$$

by exchanging the right hand side of the system of linear Equ. 5. Since $\Delta\mathbf{v}$ is just an approximation, the resulting impulses will generally not eliminate the violation exactly. Therefore, we perform the prediction and impulse computation in an iterative process until a user-defined accuracy is reached. Note that in most cases the motion of the joint connectors is almost linear. Therefore, we required an average of just one or two iterations to get an accuracy of 10^{-6} m in different test simulations with large external forces. For models with kinematic loops the matrix in equation 6 may get singular during the simulation. In this case we remove joints of the articulated body in order to break up the problematic loops and add additional impulses to mimic the effects of the kinematic loop as proposed in [BETC12].

For the simulation of a velocity constraint of the form $\mathbf{C}(\mathbf{v},t) = \mathbf{0}$, e.g. to perform a post-stabilization step [WTF06], the velocity difference $\Delta\mathbf{v}$ can be determined directly. Therefore, the system of linear Equ. 6 must be solved just once in a simulation step to obtain an exact solution for the impulses. Note that the matrix of the system is constant for time t. Hence, it must be created only once per simulation step to compute the impulses of all holonomic and velocity constraints. The required matrix multiplication $\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T$ is performed on the GPU. Furthermore, the matrix is typically sparse since a block is not zero if and only if two joints have a common body.

4.3. Collision handling

The collision handling in our applications is performed using the method presented by Bridson et al. [BFA02]. Since the treatment of collisions is not the main focus of our research, we implemented a single core CPU version of this method. However, collision detection is a major bottleneck in the simulation pipeline. Therefore, different efficient GPU-based continuous collision detection methods were developed in the last years which also can be integrated in our simulation, e.g. the work of Lauterbach et al. [LMM10] which is available as open-source project. Lauterbach et al. use a bounding volume hierarchy (BVH) of oriented bounding boxes and perform front tracking in order to improve the parallelism. Their approach is based on explicit balancing of work units. There exist also other fast GPU collision detection methods which can be used in combination with the collision handling of Bridson et al. The detection of Tang et al. [TMLT11] is based on BVHs with front tracking. They use different streams for the data and acceleration structures to parallelize the collision algorithms. Pabst et al. [PKS10] use a parallel spatial subdivision approach on the GPU instead of a BVH to perform a fast detection.

Matrix	Origin	Dim.	NNZ
Ship	CUSP	140,874	7,813,404
Bridge	RB sim.	41,550	448,668
Armadillo	Quadratic FE	46,812	3,620,142
Bunny	Quadratic FE	226,284	18,109,026
Pensatore	Linear FE	38,379	1,590,165
Cloth	Cloth sim.	76,800	3,865,140

Table 1: *Matrices used for the performance test. All matrices are quadratic and symmetric. Dimension represents the number of rows and columns, whereas NNZ specifies the number of non-zero entries. The cloth model consists of a rectangular* 160×160 *patch.*

5. Results

In this section, we analyze the efficiency of the presented data structures and compare the performance with previous works. Our tests were performed in single precision on an Intel Core 2 Quad Q6600 with 2.4 GHz and a GeForce GTX 470 unless stated otherwise. We used CUDA SDK 4.0 [NVI11a] and several matrices (see Table 1) that originate from our applications and from the tests in the work of Bell et al. [BG09]. The matrix "Cloth" is obtained from a cloth simulation with a 160×160 patch (cf. Choi et al. [CK02]). Finally, we demonstrate how different applications greatly benefit from these fast solving routines and discuss the properties of the proposed methods.

5.1. Performance Analysis

First, we focus our performance analysis on the SpMV routine that is the most time consuming part in the PCG algorithm. We applied similar performance tests that are described in the work of Bell et al. [BG09] and use their CUSP library [BG10] with texture memory cache enabled for comparison. In this library there are different kinds of matrix formats like the hybrid (HYB), coordinate (COO), ELLPACK (ELL) and the (vector) compressed sparse row (CSR) format. We omitted the diagonal- (DIA) and the scalar-CSR format as both showed a rather low performance in the tests. The performance is evaluated in terms of GFlop/s, which is the number of (single-precision) floating point operations that are executed per second. In the case of a SpMV operation this is twice the number of non-zero entries as for each entry there is a multiplication with the corresponding value in the $\mathbf{x}$ vector and an addition to the $\mathbf{y}$ vector. For the tests, we used matrices taken from the simulation applications and the matrix "FEM/Ship" from Bell et al. (see Table 1). Table 2 shows the varying row lengths in these matrices that are typical for irregular discretization schemes.

Matrix	Max.	Min.	Avg.	Overhead
Ship	99	21	52.5	2.4%
Bridge	42	6	7.8	3.6%
Armadillo	312	27	75.3	13.2%
Bunny	552	27	77.0	12.8%
Pensatore	69	15	38.4	10.8%
Cloth	48	18	47.3	0.4%

Table 2: *Maximum, minimum and average row lengths and memory overhead of the test matrices.*

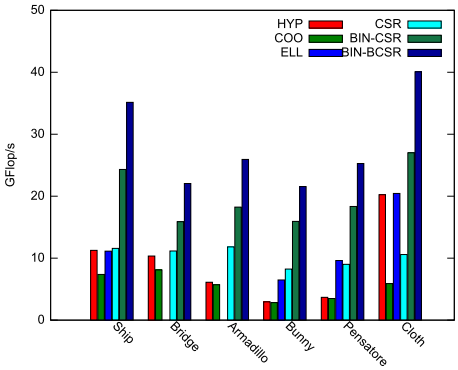

Figure 5: *Performance evaluation of data structures available in the CUSP library and our proposed BIN-CSR and BIN-BCSR data structures on a GeForce GTX 470. The library failed for tests with the bridge and the Armadillo model using the ELL format.*

The charts in Fig. 5 show the performance of the CUSP variants and our BIN-CSR and BIN-BCSR data structures for the test matrices. The measurements clearly show that our approach results in a significantly higher performance than the matrix types available in CUSP.

In order to evaluate the impact of the novel data structure and the merged conjugate gradient (MCG) algorithm, we performed several tests with matrices originating from simulations with linear finite elements. Therefore, we used 1000 iterations for each conjugate gradient run and removed the convergence criterion to compare the methods. In Fig. 6 the curves show the speed-up w.r.t the number of non-zero entries of the matrices. Using our BIN-BCSR data structure results in a nearly constant speed-up (black curve). There, a standard conjugate gradient (CG) implementation with nine kernel calls has been used. Employing our merged conjugate gradient (MCG) algorithm results in an additional improvement (blue curve). This speed-up is independent of the mesh resolution and the matrix size, since the saved overhead for kernel launches is constant. So, the curve and the impact of this optimization decreases with increasing resolution.

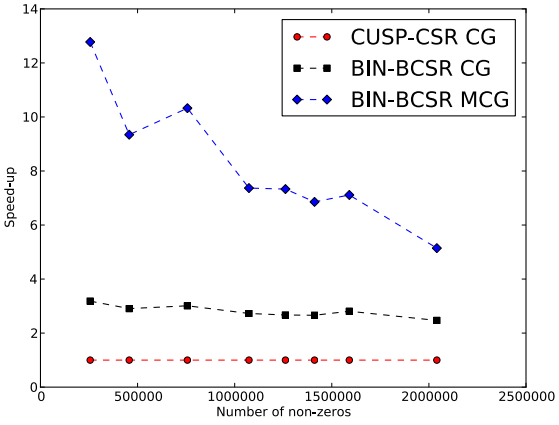

Figure 6: *Speed-up of the BIN-BCSR data structure and the merged conjugate gradient (MCG) algorithm w.r.t the number of non-zero elements in the matrix: (Red) CUSP-CG algorithm. (Black) BIN-BCSR data structure with CG algorithm. (Blue) BIN-BCSR data structure with MCG algorithm.*

Model	Simulation	CPU	GPU	Speed-up
Bridge	Rigid Body	298	21	14.19
Armadillo	Quadr. FE	1009	75	13.45
Pensatore	Linear FE	713	33	21.61

Table 3: *Average times for one simulation step (in ms) with the CPU- and GPU implementations, respectively. All CPU simulations run on one core only.*

However, this optimization is beneficial for meshes that are suitable for the interactive applications presented in this paper.

Next, we evaluate the convergence behavior of the MCG algorithm and compare the residuals with a CPU implementation using single and double precision. We therefore solve one system of linear equations from the Armadillo scenario (see Table 1). Fig. 7 top shows the residuals w.r.t the iterations. The convergence behavior of the GPU algorithm shows no significant difference compared to CPU single and double precision. The bottom diagram shows the residuals w.r.t. computation time of the GPU algorithm, of a multi-core and of a single-core implementation, all with single precision. For this test we used a GeForce GTX 580 and an Intel Core i7-2600 with 3.4 GHz. This graph clearly shows the significant performance increase using our GPU solver.

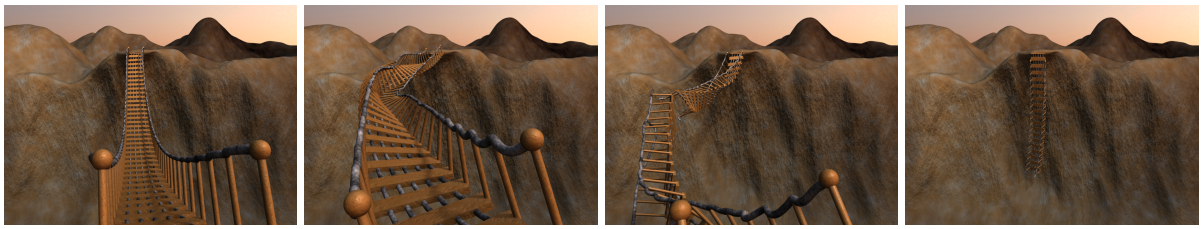

Figure 8: *Simulation of bridge with rigid body elements.*

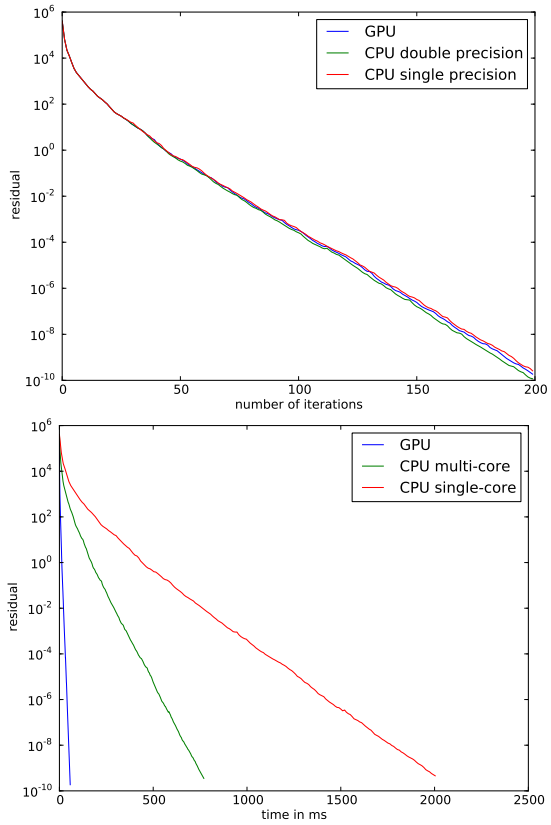

Figure 7: *Convergence behavior of the CG algorithm solving a linear system arising from quadratic finite elements. Top: Convergence behavior for GPU, CPU single precision and CPU double precision w.r.t. the number of iterations. Bottom: Convergence behavior w.r.t. computation time with GPU, multi-core and single-core implementation.*

5.2. Examples

In this section we show how different simulations benefit from the proposed algorithms. Figure 1 shows snapshots of the simulations. The number of CG-iterations was fixed to 30. Table 3 shows the timings for different simulation scenarios where all CPU simulations were performed on a single core. These timings include the update of the element matrices, the assembly of the system matrix, force computation, the setup and solution of the linear system. With our multi-core CPU implementation we achieved a speed-up factor of approximately 3 on the four available cores. In all cases, an interactive simulation on the CPU with the simulation models is not realizable as the time for computing one simulation step is far too high. However, with our proposed GPU-based solver, interactive frame rates are achieved allowing for more degrees of freedom implying higher accuracy. A key factor of the speed-up is that there is no memory transfer between CPU and GPU memory during the simulation.

Up to 100K tetrahedral linear finite elements with 30K degrees of freedom can be simulated at interactive rates. For quadratic finite elements with ten nodes per tetrahedron interactivity can be achieved with up to 13K tetrahedral elements with 28K degrees of freedom. In both cases the assembly of the system matrix takes only a fraction of the whole simulation step. Examples are shown in Fig. 1. Fig. 8 shows the real-time simulation of a complex bridge model which consists of 14K rigid bodies and has a system of linear equations with the dimension 41K. Interactive cloth simulations can be performed with more than 26K particles.

5.3. Discussion

Our proposed data structures are designed for fast SpMV operations. This is achieved by optimal coalesced memory loads without the need for a subsequent reduction of any intermediate result. It is very well suited for huge sparse matrices with varying row sizes that typically arise in finite element discretizations or in methods with local dependencies. Table 2 shows the high row variation among the test matrices. The adaption of the row lengths per bin provides a good trade-off between memory overhead and performance benefits. Our current implementation uses single precision. In our applications we did not notice any problems w.r.t. convergence or accuracy. For

a possible extension to double precision the data layout must be revised in order to achieve optimal coalescing. The data structure is not suited for dense matrices, as there is a memory and performance overhead due to the additional information of offsets and column indices. The additional memory consumption for the padding is rather low, as it depends locally on the row length within a bin (see Table 2). E.g., in the ELLPACK-R format [VOFG10] the data is padded depending on the maximum row length of the whole matrix. This format is more suitable for matrices with nearly constant row sizes.

There are other approaches optimizing the access pattern for coalesced loading on the GPU. The HYP-format [BG09], the optimizations of Baskaran et al. [BB09] and the ELLPACK-R format [VOFG10] use one thread per matrix entry. To compute the final result a subsequent reduction is needed, which either requires an additional kernel call or a limit in the row length for the sparse matrices that can be multiplied (ELLPACK-R format).

Buatois et al. [BCL09] presented a method where non-zero entries are clustered in 2×2 and 4×4 blocks. This clustering enables for the reuse of the values in the y-vector, but this can also be achieved with texture memory cache. Furthermore, they use a compressed row storage format (CRS), a synonym for CSR, that does not take care of memory alignment. So, they do not exploit coalesced loading reducing the effective memory bandwidth, which is crucial for high performance.

Allard et al. [ACF11] present a linear finite element simulation running completely on the GPU. A major difference to our work is the emulation of SpMV by computing the individual element contributions separately. This emulation is comparable to our GPU matrix construction step which is very difficult to optimize as the memory access patterns are highly irregular. It is beneficial to perform this operation only once per simulation step. We also note that for a higher number of CG-iterations our method is more efficient since a faster SpMV operation has a higher impact on the overall performance.

In the simulation of articulated bodies with kinematic loops the matrix of the system of linear equations which has to be solved may get singular and a special treatment is required. However, the bridge model (see Fig. 8) contains several closed loops and the matrix never got singular during the whole simulation.

For the matrix update the dependencies between the local and the global matrix entries must be determined in a precomputation step. If a method requires topological changes of the mesh these dependencies must be recomputed. In this case our matrix update is not directly suitable.

6. Conclusions

We have presented novel GPU data structures and algorithms that significantly accelerate various physically based simulations. We introduced a data structure that optimizes the sparse matrix vector products for matrices with varying row sizes. Furthermore, we presented a preconditioned conjugate gradient algorithm for GPUs with a minimum number of kernel calls to utilize memory bandwidth optimally. The resultant speed-up of a factor 13 at maximum allows for a significantly increased number of degrees of freedom while maintaining interactivity.

Our goal for the future is to perform a full simulation cycle on the GPU. Therefore, we are currently working on the integration of the GPU-based collision detection of Lauterbach et al. [LMM10]. After this integration, we will also implement the collision response with friction on the GPU. The goal is to obtain a collision handling without the need for expensive memory transfers between the CPU and GPU.

Acknowledgments

The work of Daniel Weber is supported by the EU-project VISTRA (ICT-285176). The work of Jan Bender is supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at Technische Universität Darmstadt. The geometry models used in this work are provided by Aim@Shape (pensatore) and Stanford Shape Repository (Armadillo, Stanford Bunny). The tetrahedral meshes were generated with CGAL library.

References

[ACF11] Allard J., Courtecuisse H., Faure F.: Implicit FEM solver on GPU for interactive deformation simulation. In *GPU Computing Gems Jade Edition*. NVIDIA/Elsevier, Sept. 2011, ch. 21, pp. 281–294. 3, 11

[Bar96] Baraff D.: Linear-time dynamics using lagrange multipliers. In *SIGGRAPH '96: Proceedings of the conference on CG and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 137–146. 3

[BB09] Baskaran M. M., Bordawekar R.: *Optimizing Sparse Matrix-Vector Multiplications on GPUs*. Tech. rep., 2009. 2, 4, 5, 11

[BCL09] Buatois L., Caumon G., Levy B.: Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst. 24* (June 2009), 205–223. 2, 4, 5, 11

[BETC12] Bender J., Erleben K., Trinkle J., Coumans E.: Interactive Simulation of Rigid Body Dynamics in Computer Graphics. In *EG 2012 - State of the Art Reports* (Cagliari, Sardinia, Italy, 2012), Cani M.-P., Ganovelli F., (Eds.), Eurographics Association, pp. 95–134. 8

[BFA02] Bridson R., Fedkiw R., Anderson J.: Robust treatment of collisions, contact and friction for cloth animation. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM, pp. 594–603. 8

[BFGS03] Bolz J., Farmer I., Grinspun E., Schröder P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph 22* (2003), 917–924. 2

[BG08] Bell N., Garland M.: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008. 2, 5

[BG09] Bell N., Garland M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 18:1–18:11. 2, 4, 5, 8, 11

[BG10] Bell N., Garland M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2010. http://cusp-library.googlecode.com. 2, 8

[BS06] Bender J., Schmitt A.: Fast dynamic simulation of multi-body systems using impulses. In *Virtual Reality Interactions and Physical Simulations (VRIPhys)* (Madrid (Spain), Nov. 2006), pp. 81–90. 3, 7

[BW98] Baraff D., Witkin A.: Large steps in cloth simulation. *Computer Graphics 32* (1998), 43–54. 5, 6, 7

[CA09] Courtecuisse H., Allard J.: Parallel dense gauss-seidel algorithm on many-core processors. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 139–147. 3

[CK02] Choi K.-J., Ko H.-S.: Stable but responsive cloth. *ACM ToG 21*, 3 (2002), 604–611. 8

[CP03] Cline M., Pai D.: Post-stabilization for rigid body simulation with contact and constraints. *Proceedings of IEEE International Conference on Robotics and Automation 3* (September 2003), 3744 – 3751. 3, 7

[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. 3

[Fea07] Featherstone R.: *Rigid Body Dynamics Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. 3

[GW08] Georgii J., Westermann R.: Corotated finite elements made fast and stable. In *VRIPHYS* (2008), pp. 11–19. 3

[KW03] Krüger J., Westermann R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG) 22*, 3 (2003), 908–916. 2

[LMM10] Lauterbach C., Mo Q., Manocha D.: gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum 29*, 2 (2010), 419–428. 8, 11

[MCG04] Mellor-Crummey J., Garvin J.: Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl. 18*, 2 (May 2004), 225–236. 3

[MG04] Müller M., Gross M.: Interactive virtual materials. In *GI* (2004), pp. 239–246. 3, 7

[MLA10] Monakov A., Lokhmotov A., Avetisyan A.: Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers*. Springer Berlin / Heidelberg, 2010. 3, 4, 5

[MTPS08] Mezger J., Thomaszewski B., Pabst S., Strasser W.: Interactive physically-based shape editing. In *SPM* (2008). 3

[NVI11a] NVIDIA: *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2011. Version 4.0, http://nvidia.com/cuda. 2, 3, 8

[NVI11b] NVIDIA: NVIDIA CUDA sparse matrix library, 2011. http://developer.nvidia.com/cuSPARSE. 2

[OH99] O'Brien J. F., Hodgins J. K.: Graphical modeling and animation of brittle fracture. In *SIGGRAPH 1999* (1999), pp. 137–146. 3

[OSV11] Oberhuber T., Suzuki A., Vacata J.: New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda. *Acta Technica 4* (2011), 447–466. 3, 4, 5

[PKS10] Pabst S., Koch A., Strasser W.: Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Computer Graphics Forum 29*, 5 (2010), 1605–1612. 8

[SG04] Schenk O., Gärtner K.: Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems 20*, 3 (2004), 475–487. 3

[She94] Shewchuk J. R.: *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Tech. rep., Pittsburgh, PA, USA, 1994. 5

[TMLT11] Tang M., Manocha D., Lin J., Tong R.: Collision-streams: fast gpu-based collision detection for deformable models. In *Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2011), I3D '11, ACM, pp. 63–70. 8

[TW88] Terzopoulos D., Witkin A.: Deformable models. *IEEE CGA 8*, 6 (November 1988), 41–51. 3

[VOFG10] Vazquez F., Ortega G., Fernández J.-J., Garzón E. M.: Improving the performance of the sparse matrix vector product with GPUs. In *CIT'10* (2010), pp. 1146–1151. 2, 4, 5, 11

[WKS*11] Weber D., Kalbe T., Stork A., Fellner D., Goesele M.: Interactive deformable models with quadratic bases in Bernstein-Bézier-form. *TVC 27* (2011), 473–483. 3, 7

[WTF06] Weinstein R. L., Teran J., Fedkiw R.: Dynamic simulation of articulated rigid bodies with contact and collision. In *IEEE Transactions on Visualization and CG* (2006), vol. 12, pp. 365–374. 3, 7, 8