Electronic Transactions on Numerical Analysis.
Volume 37, pp. 263-275, 2010.
Copyright © 2010, Kent State University.
ISSN 1068-9613.

A STREAMING APPROACH FOR SPARSE MATRIX PRODUCTS AND ITS APPLICATION IN GALERKIN MULTIGRID METHODS[*]

JOACHIM GEORGII[†] AND RÜDIGER WESTERMANN[†]

Abstract. In this paper, we present a numerical algorithm for computing products of the form RKR^{T}, where R, R^{T}, and K are sparse matrices. By reformulating the problem into the simultaneous processing of a sequential data and control stream, cache miss penalties are significantly reduced. Even though the algorithm increases memory requirements, it accelerates sparse matrix products on recent processor architectures by a factor of up to 4 compared to previous approaches. We apply the algorithm to compute consistent system matrices at different resolution levels in a dynamic multigrid elasticity simulation, and we show its efficiency for nested and non-nested mesh hierarchies.

Key words. sparse matrix products, cache-awareness, multigrid, Galerkin update

AMS subject classifications. 65F50, 65M55, 65M60, 65Y20, 68W01, 74B99, 74H15

1. Introduction.

The core computations of many numerical simulation techniques are based on sparse matrices, making it important to provide general purpose high performance software libraries for such structures. The multiplication of two sparse matrices, in particular, is often required when solving partial differential equations on unstructured finite element hierarchies using geometric multigrid schemes. To ensure a consistent calculation of quantities on different resolution levels, the coarse grid operator can be obtained by computing products of the form RKR^{T}. Here, R and R^{T} are, respectively, the restriction and interpolation operators used to transfer quantities between different resolution levels, and K is the fine grid operator. It is said that coarse grid operators constructed in this way satisfy the Galerkin condition, and it has been shown in previous work that such a construction is the natural choice to define the coarse grid operator [5, 23].

In case of unstructured mesh hierarchies, the corresponding matrix representations of the involved operators are sparse and non-zero entries are randomly scattered. If the non-linear strain tensor is used in the dynamic simulation of the system over time, these entries, and thus the entire multigrid matrix hierarchy, have to be updated in every simulation time step. As we will show later in this paper, the update of the matrix hierarchy dominates the overall performance of multigrid methods, taking about an order of magnitude longer than the system solver. The reason for this is that sparse matrix products operate significantly below the CPU's peak performance due to the bottleneck of data transfer in the CPU memory hierarchy. A direct implication thereof is that standard algorithms for computing sparse matrix products can hardly achieve real-time performance in dynamic simulations for reasonably sized grid hierarchies.

To overcome this limitation, we focus on improving the memory access patterns of sparse matrix operations in this paper. We present a linear layout of computational cores for sparse matrix multiplication, which can effectively reduce the average memory access time. By reformulating the problem into the simultaneous processing of a sequential data and control stream, the locality of memory access operations can be improved, resulting in considerably less cache miss penalties. In addition, we present further improvements based on the particular form of products to be computed in multigrid Galerkin methods. We include the proposed matrix operation into a multigrid approach for deformable body simulation based

[*]Received December 7, 2007. Accepted for publication April 29, 2010. Published online September 7, 2010. Recommended by A. Frommer.

[†]Computer Graphics & Visualization Group, Technische Universität München, Germany ({georgii,westermann}@tum.de).

on a non-linear elasticity model. By exploiting symmetry considerations and symbolic calculations to optimize in-place updates of matrix entries, we show a significant acceleration of the multigrid method.

Our approach falls into the category of algorithmic techniques for efficient memory access in matrix operations. In contrast to previous work on the optimization of dense matrix operations [1, 6, 11], sparse matrix-vector operations [16], sparse-dense matrix products [9], and sparse matrix transposition [7], in this work we focus on improving the compute-to-memory ratio in sparse matrix multiplications. For such matrices, the Yale sparse matrix format [10, 14, 19] was introduced in the early eighties, and specific variants were developed to exploit the density of subblocks of such matrices [3, 20]. Parallelization strategies of sparse matrix operations have been discussed in [4, 22], and most recently the complexity of sparse matrix algorithms has been analyzed theoretically [24]. It is worth noting here that, despite all the different optimization strategies, sparse matrix products are still not standard in sparse libraries such as PETSc [2].

2. Matrix data structure.

The matrix data structure we use in the computation of sparse matrix products is row-based (Yale or compressed row format [10]). For a sparse matrix K, non-zero entries and their respective column indices are stored in two separate arrays, row by row. In addition, for every row i an index to the first non-zero element in this row is stored as depicted in Figure 2.1. We denote by S_i^K the set of indices to non-zero elements for row i. In the following, we will refer to this format as *row-compressed* (RC) matrix format.

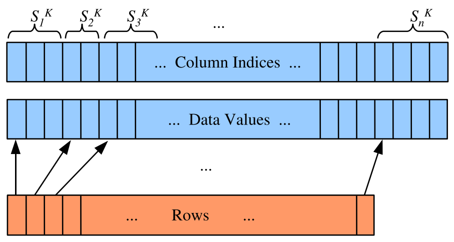

FIG. 2.1. *Row-compressed matrix format.*

We extend the RC matrix format in such a way that each entry in the data array can store a block of non-zero data values instead of just one value. This is beneficial in the 3D simulation we perform, where the system matrix consists of 3×3 blocks of non-zero elements. In this case, the memory that is required to store the column indices can significantly be reduced. In the following, we will refer to this format as *block-row-compressed* (BRC) matrix format. For the sake of clarity, we will restrict the following discussion to the *row-compressed* format, keeping in mind that the extension to the BRC format is straightforward.

3. Sparse matrix products.

We now illustrate different algorithms for computing products of sparse matrices of the form $R\,K\,R^{\mathrm{T}}$. Both the matrix K and the matrix R are assumed to be sparse, and they are stored using an RC matrix format. We begin with a description of the so called naïve approach. Next, we introduce an optimized 1-step approach, which avoids the intermediate representation of the naïve approach. Finally, we improve the memory behavior of the 1-step approach by reformulating the problem into the processing of sequential data and control streams, which significantly reduces cache miss penalties.

Generally, we distinguish two settings for sparse matrix products. First, the algorithms are required to construct the sparse matrix structure of the result matrix in a pre-processing step. This setting is referred to as the symbolic processing, since it constructs an entry in the result matrix for each potentially non-zero element. Second, the proposed algorithms allow for efficient in-place updates after the data values of K or R have been updated, which utilize the pre-computed matrix structure to only update the potentially non-zero elements. In this paper, we will focus on the in-place updates and analyse different algorithms in detail.

3.1. Naïve approach. The naïve approach to perform a multiplication of the form $R\,K\,R^{\mathrm{T}}$, where R and K are sparse matrices, first computes an intermediate representation $F = R\,K^{\mathrm{T}}$, which is then used to compute $E = R\,F^{\mathrm{T}}$. Splitting the product in this way is best suited for the RC matrix format, because it requires only the calculation of sparse dot products and therefore allows accessing the data structure in an optimal way. Note that in both single products the second matrix is transposed, allowing the rows of the data structure of the non-transposed matrix to be accessed one by one.

3.2. 1-step approach. To avoid storing and computing the intermediate matrix F, let us have a closer look at the matrix product to be performed. Expanding the matrix product $E = R\,K\,R^{\mathrm{T}}$ yields

$$E_{ij} = \sum_{l \in S_i^R} R_{il} \left(\sum_{k \in S_l^K \cap S_j^R} K_{lk} R_{jk} \right).$$

The outer sum is evaluated only for non-zero entries in the index set S_i^R. The calculation of the inner sum can be optimized by only considering indices in the intersection of the two index sets S_l^K and S_j^R, as in all other cases the resulting terms of the sum are zero. In the calculation of all sums the data structures are now accessed row-wise, resulting in cache-friendly memory access patterns. To perform an in-place update of the matrix E (assuming the structure of E to be known), only indices $j \in S_i^E$ have to be considered. Pseudo-code for this operation is given in Algorithm 1. To create the matrix structure of E, the respective loop is performed for all indices j from 0 to E.numCols -1. An entry in the sparse matrix structure is only created if $E_{ij} \neq 0$.

On the downside, in the 1-step approach the inner sum in the product calculation might have to be traversed several times because the same values of j and l can occur for different indices i. Therefore, the in-place 1-step approach performs slightly worse than the in-place

Algorithm 1 1-step multiplication (in-place)

```
Require: Matrices $K, R$, matrix structure of $E$
Ensure: $E = RKR^{\mathrm{T}}$

for $i = 0$ to $E$.numRows do
    for $j \in S_i^E$ do
        $E_{ij} = 0$;
        for $l \in S_i^R$ do
            double $sum = 0$;
            for $k \in S_l^K \cap S_j^R$ do
                $sum = sum + K_{lk} \cdot R_{jk}$;
            end for
            $E_{ij} = E_{ij} + sum \cdot R_{il}$;
        end for
    end for
end for
```

variant of the naïve approach. These observations are validated in the result Section 5. However, we use the 1-step approach as the initialization phase for the streaming acceleration approach we describe next.

We describe the design and implementation of an acceleration structure for *in-place* sparse matrix multiplication. Although this approach comes at the expense of additional memory requirements to store the acceleration structure, it achieves an acceleration of up to a factor of 30 compared to the naïve approach described in Section 3.1.

3.3. 1-step stream acceleration. The performance of the 1-step approach is mainly limited due to the following properties: First, to determine the intersections $S_l^K \cap S_j^R$ in Algorithm 1, the entire (ordered) sets S_l^K and S_j^R have to be processed even though their intersection is typically very small or even empty. Second, the indices l and j themselves are determined by processing sparse index sets. Accessing these sets, namely S_l^K and S_j^R, produces scattered read operations that can probably not be served from cache. The matrix products we focus on in this section have the property that the main matrix K is supposed to be dynamic and thus subject to frequent changes. The matrix R, on the other hand, is supposed to be static. These assumptions hold in applications to geometric multigrid approaches, where the transfer operations between different resolutions levels do not change. Although the algorithm presented is not strictly limited to this setting, dynamic in the matrix R introduces performance drawbacks.

To address the first issue, we propose a novel acceleration data structure that stores the intersection of the index sets S_l^K and S_j^R for all indices l and j. To address the second issue, we construct a data and control stream that is aligned with the data structure of the matrix K. Due to this particular layout, scattered memory read operations to access pre-computed intersections $S_l^K \cap S_j^R$ can be avoided. Because the matrices R and R^{T} are the same except for transposition and do not change over time, their contributions to the product can also be encoded into the stream.

In summary, we build a data stream that encodes data values of R along with indices into the destination matrix. Additionally, a control stream is used to encode how many pairs of data values and indices have to processed for each non-zero entry of the matrix K. The indices are used to scatter the multiplied entries from K and R into the destination matrix E. In this way, only the final write operation accesses the memory randomly. Due to the fact that the destination matrix E is smaller in size than the source matrix K, memory access operations are reduced compared to the setting where we loop over the matrix E while randomly accessing values of K. An overview of the streaming approach is given in Figure 3.1.

3.3.1. Stream design. The acceleration data structure is aligned with the sparse matrix data structure of K, and it consists of two different streams: A *control stream* containing control flags and a *data stream* containing values of R and respective indices to E. These streams store the information required to scatter a single entry of K into the respective positions of E. A single byte of the control stream is interpreted as follows: The first bit indicates whether the next non-zero entry of the matrix K should be fetched or the previous entry of K is used in the current calculation. The remaining seven bits indicate the number of data value/index pairs from the data stream that have to be processed. Note that with this scheme at most a number of 127 pairs can be encoded in one single control byte. If an entry of K is scattered into the result matrix more than 127 times, an additional control byte has to be used with the first bit set to 0. However, in all examples used throughout this paper we never exceeded the limit of 127 scatter operations.

3.3.2. Stream construction. Stream construction can be performed analogously to a 1-step multiplication as described in Algorithm 1. However, this approach performs the op-

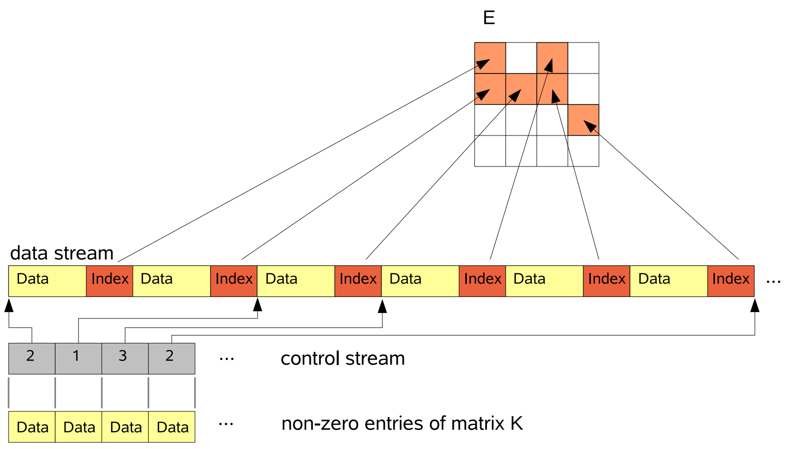

FIG. 3.1. *Overview of the 1-step stream acceleration.*

erations using a non-optimal memory layout, since the outer loops process the destination matrix E rather than the matrix K. Therefore, we change the ordering of the loops. The outer loops running over all entries of the destination matrix now become the innermost loops, yielding outer loops over all elements of the matrix K (using the indices l and k). Then, for each entry of the matrix K all products $R_{il} \cdot R_{jk}$ and indices i, j into the destination matrix E are determined and can be directly encoded into the data stream.

Algorithm 2 lists the pseudo-code for the stream construction phase. The method E.getIndex(i, j) calculates the index of the element in the linearized data array of E. This index is used to quickly access the respective element in the stream processing stage. The stream's push() operation stores the previous value and index into the data stream and increments the number of pairs stored in the last control byte. If the maximum number of 127 is exceeded, a new control byte with the first bit set to 0 is appended to the control stream. The stream's setNext() operation creates a new control byte with the first bit set to 1, thus advancing to the next non-zero element of K.

3.3.3. Stream processing. Processing the stream to update the destination matrix E is performed in two steps. These steps are repeated until the entire stream has been processed, i.e., until all non-zero entries of K have been processed. In the following description of the two steps, we denote by l and k the row and column indices of the first non-zero entry of K:

Step 1: If the first bit of the control byte is 1, the index k is advanced to the next non-zero entry in row l. If no such entry is available, the row index l is incremented to the next non-empty row and k is set to the respective first non-zero column index. The value K_{lk} is stored in a temporary register t. From the control byte, the number p of weight/index pairs that have to be processed next are determined.

Step 2: The following steps are performed p times:
A data value w and an index value i are read from the data stream. The product $w \cdot t$ is added to the value at position $E(i)$, where $E(i)$ addresses the i-th position in the linearized representation of E.

3.3.4. Stream optimization. The constructed stream can be further optimized with respect to the data values stored in the stream. If the same data value is repeated several times in the stream, we can save memory by storing this value only once and by accompanying it by the set of destination indices. For instance, this is the case in nested grid hierarchies.

Algorithm 2 Stream construction (in-place)

```
Require: Matrices $K, R^{\mathrm{T}}$, structure of matrix $E$
Ensure: $E = RKR^{\mathrm{T}}$

  for $i = 1$ to $E$.numRows do
    for $j \in S_i^E$ do
      $E_{ij} = 0$;
    end for
  end for
  for $l = 0$ to $K$.numRows do
    for $k \in S_l^K$ do
      for $i \in S_l^{R^{\mathrm{T}}}$ do
        for $j \in S_i^E \cap S_k^{R^{\mathrm{T}}}$ do
          $E_{ij} = E_{ij} + K_{lk} \cdot R^{\mathrm{T}}_{li} \cdot R^{\mathrm{T}}_{kj}$;
          stream.push($R^{\mathrm{T}}_{li} \cdot R^{\mathrm{T}}_{kj}$, $E$.getIndex$(i, j)$);
        end for
      end for
      stream.setNext();
    end for
  end for
```

Since the hierarchy is generated by inserting the middle vertex on each edge, the values in the matrix R are either 1 or 0.5. Therefore, only three different types of values $1, 0.5,$ and 0.25 may have to be stored in the stream. To allow for this kind of optimization, the data value/index pairs are sorted with respect to their values after all pairs belonging to a single entry of K have been generated. Finally, the control stream needs to be adjusted to store for each data value w the number of destination indices to be considered.

3.4. Symmetry optimization. If the matrix K is symmetric, then the 1-step algorithm and the stream acceleration can be performed nearly twice as fast. This is due to the fact that only the upper triangular matrix of E has to be computed, and the lower triangular part can be determined from the respective mirrored entries. We do not introduce a symmetric row-compressed format, as in this case matrix-vector products cannot be processed at full performance rates due to the improper memory access patterns. A symmetric row-compressed format only stores the upper triangular matrix of K. On average, a single row-vector product then can only access half of the data values of K efficiently, while the other half of the values have to be fetched from different rows; see Figure 3.2.

For this reason, we do not change the matrix format. Instead, the lower triangular matrix is determined from the upper triangular part. If the block-row-compressed matrix format is used, this step can be performed efficiently, as 3×3 blocks can be copied at once. For the pure row-compressed format, this symmetry optimization is not as efficient since single data values have to be copied.

3.5. Parallelization. The 1-step stream acceleration algorithm can be parallelized on N compute nodes by partitioning the data and control stream into N disjoint parts, and by distributing these parts to the nodes. In the partitioning process, the stream is only split at control bytes with the first bit equal to 1. To split the stream into disjoint parts for which approximately the same number of operations are performed, we first count the number of non-zero entries of K as well as the number of write operations (addresses into the matrix E^k). The

ETNA
Kent State University
http://etna.math.kent.edu

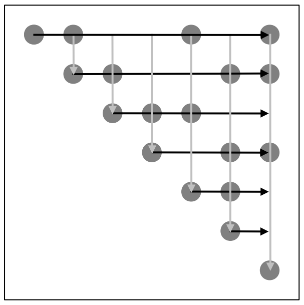

FIG. 3.2. *Matrix-vector products using a symmetric sparse matrix format: On average, a row-vector product can only access half of the data values of K memory-efficiently (black), while the other half of the values have to be fetched from different rows (light grey).*

stream is then partitioned by considering the resulting numbers.

Each node also stores a copy of the destination matrix, E^k. For the sake of clarity we assume that the matrix K is duplicated on each node, even though only the non-zero elements corresponding to the respective parts of the stream are required. Once the local computations on every node have been finished, the matrices E^k are joined into the result matrix by adding the per-node contributions: $E = \sum_{k=1}^{N} E^k$. Since all matrices E^k have the same structure, this summation can be carried out in place.

4. Application to elasticity problems.

The motion of a deforming volumetric object can be simulated by a displacement field in an elastic solid. Given such a solid in the reference configuration $x \in \Omega \subseteq \mathbb{R}^3$, the deformed solid is modeled using a displacement function $u(x), u : \mathbb{R}^3 \rightarrow \mathbb{R}^3$.

Driven by external forces, the dynamic behavior of the deformed solid is governed by the Lagrangian equation of motion,

$$M\ddot{u} + C\dot{u} + Ku = f, \tag{4.1}$$

where M, C, and K are, respectively, known as the system's mass, damping, and stiffness matrices. The vector u consists of the linearized displacement vectors of all vertices and f is the linearized force vectors applied to these vertices.

By discretization of u,$\dot{u}$ and $\ddot{u}$ with respect to time, the differential equation can be transformed into a set of difference equations. Most commonly, either an implicit Euler integration or a second order accurate Newmark scheme can be used for time integration. Applying the integration scheme to equation (4.1) yields a system of linear equations,

$$\tilde{K}u^{t+dt} = \tilde{f}^{t+dt}.$$

A rotational invariant formulation of the Cauchy strain tensor is obtained by using the so-called co-rotated strain of linear elasticity [21]. In this formulation finite elements are first rotated into their initial configuration before the strain is computed. In this way, although strain is still approximated linearly, artificial forces as they are computed for large deformations using the Cauchy strain are significantly reduced. Rotations are calculated per element using a polar decomposition of the deformation gradient $\nabla(x + u(x))$ [12, 15] or an energy minimization [13]. In particular, we show that an efficient computation of the sparse matrix product $R\,K\,R^{\mathrm{T}}$ is the most important step to obtain fast geometric multigrid methods based on the Galerkin approach. This is due to the fact the resulting system of linear equations changes in every simulation time step and thus the coarse grid matrices of the multigrid solver have to be re-computed at run time using the product $R\,K\,R^{\mathrm{T}}$.

4.1. Multigrid method. We apply a geometric multigrid method to solve the system of linear equations $\tilde{K}u^{t+dt} = \tilde{f}^{t+dt}$ in every simulation step. To transfer quantities from a finer to a coarser grid and vice versa, geometry-specific restriction and interpolation operators are established. These operators are described for nested and non-nested hierarchies of meshes.

Given a finite element mesh at the coarsest resolution level, a hierarchy can be generated in a top-down fashion by uniformly splitting each finite element, yielding a nested hierarchy. Although the transfer operators can be defined in a straight forward way, this approach fails in precisely approximating the object's boundaries at finer resolution levels. Furthermore, subsequent subdivisions might lead to ill-shaped elements that can cause numerical problems in the simulation. To avoid these disadvantages, we use linear transfer operators that do not require a nested hierarchy of meshes. These operators establish geometric relations in a multilevel hierarchy of unstructured meshes by means of barycentric interpolation as proposed by Georgii and Westermann [12].

A coarse grid correction performs as follows on the fine grid h. It requires a linear transfer operator R_h, a coarse grid operator K^H, and an initial approximation $\hat{u}^h$ of the solution:

① Pre-Smoothing of $\hat{u}^h$	
② Compute residual	$r^h = f^h - K^h\hat{u}^h$
③ Restrict residual to coarse grid	$r^H = R_h r^h$
④ Solution on coarse grid	$K^H e^H = r^H$
⑤ Transfer correction	$e^h = R_h^T e^H$
⑥ Correction	$u^h = \hat{u}^h + e^h$
⑦ Post-Smoothing of u^h	

By recursive application of the coarse grid correction to stage ④ and by using any direct solver to compute its solution on the coarsest grid, a full multigrid V-cycle is obtained.

4.2. Galerkin multigrid approach. Based on the geometric restriction and interpolation operators, the coarse grid matrices are constructed using the Galerkin property [5]. In particular, for all but the finest hierarchy level the system matrices are computed as

$$K^H = R_h K^h R_h^T.$$

This approach ensures consistent calculations on different resolution levels, and it is especially suited to construct black-box multigrid solvers [23].

5. Results. In this section, we analyze the performance and memory requirements of the proposed algorithm for computing sparse matrix products. We use this algorithm to compute consistent system matrices at different resolution levels in a dynamic multigrid elasticity simulation (see Figure 5.1 for some example models), and we give detailed timing statistics for this particular application. All timings except for the parallelization timings were measured on one CPU core of an Intel Core™ 2 Duo 2.4 GHz equipped with 2 GB RAM.

In the following, we apply the 1-step stream acceleration approach to update the coarse grid operators in a Galerkin multigrid approach. This requires the computation of matrix products of the form RKR^{T}, where R and R^{T} are, respectively, the restriction and interpolation operator used to transfer quantities between different resolution levels in a mesh hierarchy and K is the fine grid operator. We distinguish between nested and non-nested mesh hierarchies to demonstrate how the performance depends on the fill rate of the restriction and prolongation matrices.

Table 5.1 shows the respective timings for a nested hierarchy of simulation grids. We show timings for the *in-place* variants of different algorithms that can be used at run time

TABLE 5.1

Timings in [ms] for the update of the coarse grid operators in a nested mesh hierarchy using the BRC sparse matrix format. The numbers in parentheses give the timings for the symmetric variants; see Section 3.4. The last columns are reference timings determined with the SuiteSparse library.

model	dimension / fillratio	naïve (in-place)	1-step (in-place)	stream (init)	stream (in-place)	stream (mem)	SuiteSparse (Reference)
bridge3k 3 levels	2.46k 1.4%	24	22 (13)	27 (26)	**2** (**1**)	2.1 MB (1.7 MB)	7
bridge24k 4 levels	15.7k 0.25%	229	180 (103)	233 (207)	**12** (**11**)	16 MB (13 MB)	43
bridge128k 5 levels	111.8k 0.25%	2073	1301 (728)	1797 (1561)	**107** (**98**)	118 MB (93 MB)	387

TABLE 5.2

Timings in [ms] for the update of the coarse grid operators in a non-nested mesh hierarchy using the BRC sparse matrix format. The numbers in parentheses give the timings for the symmetric variants; see Section 3.4. The last columns are reference timings determined with the SuiteSparse library.

model	dimension / fillratio	naïve (in-place)	1-step (in-place)	stream (init)	stream (in-place)	stream (mem)	SuiteSparse (Reference)
liver3k 2 levels	2.52k 1.4%	158	212 (102)	180 (131)	**7** (**5**)	15 MB (9 MB)	10
bunny11k 2 levels	9.00k 0.39%	732	997 (514)	566 (424)	**26** (**17**)	46 MB (28 MB)	47
horse50k 3 levels	36.7k 0.10%	3347	6654 (3646)	4198 (2664)	**167** (**107**)	300 MB (179 MB)	171

once the matrix structures are known. For each example we give timings for the block-row-compressed (BRC) matrix format. As this format benefits from the symmetry optimization described in Section 3.4, respective timings are given in parentheses if applicable.

Column three shows the time required by the naïve approach, column four lists the time for the 1-step approach. The next three columns show the initialization time, the update time, and the memory requirement of the 1-step stream accelerated approach. Note that both initialization and update are performed *in-place*, which implies that the structure of the result matrix has already been determined. The last column gives reference timings determined with the SuiteSparse library [8]. To the best of our knowledge this is the only library that supports optimized sparse matrix products. Table 5.2 contains the same columns as Table 5.1, but now the models are hierarchically represented by a non-nested mesh hierarchy. From the given performance measurements the following results can be concluded:

1. The *in-place* variant of the 1-step algorithm can only outperform the naïve approach if nested mesh hierarchies are used. In case of non-nested hierarchies, the naïve approach computes the results faster than the 1-step approach. However, the naïve approach requires additional memory to store the temporary matrix F.
2. The stream-accelerated 1-step algorithm for the *in-place* update outperforms the naïve approach by a factor of 10–30. It comes at the expense of additional memory to store the data and control streams, but the data can be streamed efficiently through the CPU memory hierarchy. The performance gain clearly compensates for

the additional memory requirements.

3. In direct comparison to the optimized sparse matrix package *SuiteSparse*, we still can achieve a performance gain of up to a factor of 4. We found that our stream acceleration approach is mainly beneficial in case of nested hierarchies, where the restriction matrix R contains fewer non-zero entries than in case of non-nested hierarchies. However, the *SuiteSparse* library uses another sparse matrix format than our approach. Therefore, operations, such as the Gauss-Seidel relaxation required by the multigrid method, might induce performance issues with this library (the library does not support Gauss-Seidel relaxation natively). On the other hand, the proposed stream acceleration approach directly updates the RC or BRC matrix structure, which allows for very efficient matrix vector multiplication as well as Gauss-Seidel relaxation, and the approach is always faster than the SuiteSparse library.

5.1. Cache analysis. We have analyzed our algorithms using Intel's VTune$^{\text{TM}}$ performance analyzer [17]. It allows to directly read the CPU's performance counters related to single methods of the application. Especially, it allows us to count the cache misses in our algorithms. In Table 5.3, we list the number of cache misses reported by the VTune$^{\text{TM}}$ as well as the number of cache read operations for the level 1 data cache of the Core$^{\text{TM}}$ 2 Duo processor (32 KB). Although the streaming approach requires much more memory, the number of memory read operations is significantly reduced compared to the 1-step approach due to the pre-computations. As a consequence, the number of cache misses is significantly reduced for all examples we considered. Furthermore, we observe that for non-nested hierarchies the stream acceleration approach even increases the instructions per cycle the CPU can process.

TABLE 5.3
Cache analysis of a complete in-place multigrid hierarchy update for different models using nested and non-nested hierarchies. We compare the 1-step algorithm with the optimized stream acceleration approach.

model	algorithm	cache misses	cache read ops	miss ratio	instructions per cycle	instructions
horse50k	1-step (in-place)	74.8M	2691M	2.8 %	1.13	7800M
	1-step stream accel.	0.9M	249M	0.3 %	1.72	673M
bridge128k	1-step (in-place)	3.9M	875M	4.4 %	1.29	2347M
	1-step stream accel.	0.7M	199M	0.3 %	1.27	544M

5.2. Parallelization. In this section, we demonstrate that the proposed stream acceleration approach allows effective parallelization. We analyze the performance gain we can achieve on a Xeon$^{\text{TM}}$1.8GHz Quad Core processor architecture. The implementation is realized with the Threading Building Blocks library by Intel [18]. We achieve a speed-up of 1.7 on two cores, and a speed-up of 2.2-2.5 on four cores as shown in Table 5.4. The limitation of the speed-up achieved for four cores is mostly due to memory bottlenecks on the multicore architecture. The computational overhead induced by the final reduce step is less than 20% of the overall time.

TABLE 5.4
Parallelization of the stream acceleration approach on a Xeon$^{\text{TM}}$1.8GHz Quad Core processor.

model	1 thread	2 threads	3 threads	4 threads
horse50k	293 ms	175 ms	143ms	117ms
bridge128k	255ms	151ms	131ms	112ms

ETNA
Kent State University
http://etna.math.kent.edu

5.3. Galerkin multigrid. In the following, we demonstrate the performance of the proposed algorithm in a Galerkin multigrid finite-element approach. Our intention is to show the performance of the algorithm in the interplay of matrix assembly and system solution, and thus to demonstrate that in the particular scenario fast sparse matrix products are mandatory.

The implementation we analyze is based on the BRC matrix format. We use the symmetric 1-step algorithm in the pre-processing stage and the respective stream accelerated in-place variant to build or update the multigrid matrix hierarchy at run time. Table 5.5 gives timing statistics for the simulation of different deformable models based on nested and non-nested tetrahedral mesh hierarchies. Column 8 summarizes the total time for one simulation step using the co-rotational simulation. Columns 5 to 7 show the time that is required for the calculation of element rotations and matrix assembly, the update of the multigrid hierarchy, and the system solver. It can be observed that the multigrid update dominates the overall performance of the entire simulation system if the naïve approach is used; see Table 5.1 and 5.2. The time required by the stream accelerated approach, on the other hand, is of the same order as the solution time and even falls below that time. Therefore, the stream acceleration approaches significantly improves the Galerkin multigrid approach in this specific application. Some example models are shown in Figure 5.1.

TABLE 5.5
Timing statistics for dynamic deformable body simulations using the co-rotated Cauchy strain. The simulation times include the per-frame update of the system matrix including the computation of element rotations (Assem.), the per-frame update of the multigrid matrix hierarchy using the BRC matrix format and symmetric 1-step stream acceleration (Update), and the time required by the system solver (Solve).

model	# level	# tet	# vert	time [ms]			
				Assem.	Update	Solve	Total
bridge3k	3*	3072	825	8	2	3	13
liver8k	3	8078	1915	16	10	12	38
bunny11k	2	11206	3019	25	17	19	61
bridge24k	4*	24576	5265	51	11	17	79
horse50k	3	49735	12233	153	107	58	318

It is worth noting that the presented multigrid hierarchy update can be used in settings with the full Green strain tensor and non-linear material laws, too. The derived system of non-linear equations is solved with a standard solver, e.g., a Multigrid Newton method [5]. However, each Newton step then requires to rebuild the matrix hierarchy of the appropriate Jacobian matrix, which can be achieved efficiently with the proposed algorithm.

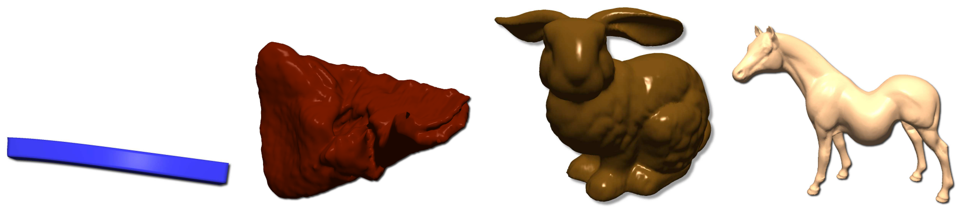

FIG. 5.1. *Visualizations of the deformable bodies used to analyze the performance of the proposed matrix operation in a Galerkin multigrid scheme. From left to right: bridge24k, liver8k, bunny11k and horse50k.*

6. Conclusion. In this paper, we have presented novel algorithms for the calculation of sparse matrix products. By reformulating the problem into the simultaneous processing of a data and a control stream, cache miss penalties could be significantly reduced. The new approach outperforms previous approaches to compute sparse matrix products.

In particular, we have shown that the proposed algorithm can be used efficiently in Galerkin multigrid approaches to update the hierarchy of system matrices. The 1-step stream acceleration approach is especially designed to support the computation of matrix products as they arise in such scenarios. It can thus be seen as a generic basis for constructing Galerkin multigrid solvers.

The proposed matrix algorithms can be used in other applications as well. The streaming approach can be easily modified to compute single matrix products if one of the matrices is constant. Furthermore, it can be extended to account for dynamic in both matrices. By parallelizing the approach on multiple compute nodes even large matrices can be handled efficiently.

REFERENCES

[1] M. BADER AND C. ZENGER, *Cache oblivious matrix multiplication using an element ordering based on a Peano curve*, Linear Algebra Appl., 417 (2006), pp. 301–313.

[2] S. BALAY, K. BUSCHELMAN, V. EIJKHOUT, W. D. GROPP, D. KAUSHIK, M. G. KNEPLEY, L. C. MCINNES, B. F. SMITH, AND H. ZHANG, *PETSc users manual*, Tech. Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[3] A. J. C. BIK AND H. A. G. WIJSHOFF, *Automatic nonzero structure analysis*, SIAM J. Comput., 28 (1999), pp. 1576–1587.

[4] R. H. BISSELING AND W. MEESEN, *Communication balancing in parallel sparse matrix-vector multiplication*, Electron. Trans. Numer. Anal., 21 (2005), pp. 47–65. http://etna.math.kent.edu/vol.21.2005/pp47-65.dir.

[5] W. L. BRIGGS, V. E. HENSON, AND S. F. MCCORMICK, *A Multigrid Tutorial*, Second ed., SIAM, Philadelphia, 2000.

[6] S. CHATTERJEE, V. V. JAIN, A. R. LEBECK, S. MUNDHRA, AND M. THOTTETHODI, *Nonlinear array layouts for hierarchical memory systems*, in ICS '99: Proceedings of the 13th international conference on Supercomputing, New York, NY, USA, ACM, 1999, pp. 444–453.

[7] S. CHATTERJEE AND S. SEN, *Cache-efficient matrix transposition*, in Proceedings of the International Symposium on High-Performance Computer Architecture, Los Alamitos, CA, USA, IEEE Computer Society, 2000, p. 195.

[8] T. DAVIS, *Direct Methods for Sparse Linear Systems*, SIAM Series on the Fundamentals of Algorithms, SIAM, Philadelphia, 2006.

[9] R. DOALLO, B. FRAGUELA, AND E. ZAPATA, *Direct mapped cache performance modeling for sparse matrix operations*, in Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing '99, IEEE, Funchal, Portugal, 1999, pp. 331–338.

[10] S. C. EISENSTAT, M. C. GURSKY, M. H. SCHULTZ, AND A. H. SHERMAN, *Yale sparse matrix package*, Internat. J. Numer. Methods Engrg., 18 (1982), pp. 1145–1151.

[11] J. D. FRENS AND D. S. WISE, *Auto-blocking matrix multiplication, or tracking BLAS3 performance from source code*, in Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Program., ACM, Las Vegas, NV, Vol. 32, July 1997, pp. 206–216.

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

[13] ———, *Corotated finite elements made fast and stable*, in Proceedings of the 5th Workshop On Virtual Reality Interaction and Physical Simulation, Eurographics Association, Grenoble, France, 2008, pp. 11–19.

[14] F. G. GUSTAVSON, *Two fast algorithms for sparse matrices: Multiplication and permuted transposition*, ACM Trans. Math. Software, 4 (1978), pp. 250–269.

[15] M. HAUTH AND W. STRASSER, *Corotational simulation of deformable solids*, in Proceedins of WSCG - The International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision '04, Union Agency - Science Press, Plzen, Czech Republic, 2004.

[16] E.-J. IM, K. YELICK, AND R. VUDUC, *Sparsity: Optimization framework for sparse matrix kernels*, Int. J. High Perform. Comput. Appl., 18 (2004), pp. 135–158.

[17] INTEL CORPORATION, *Intel VTune Performance Analyzer*. http://software.intel.com/en-us/intel-vtune/.

[18] ———, *Threading Building Blocks*. http://www.threadingbuildingblocks.org/.

[19] J. M. McNamee, *A sparse matrix package - part II: Special cases*, ACM Trans. Math. Software, (1983), pp. 344–345.

[20] A. Pinar and V. Vassilevska, *Finding nonoverlapping substructures of a sparse matrix*, Electron. Trans. Numer. Anal., 21 (2005), pp. 107–124. http://etna.math.kent.edu/vol.21.2005/pp107-124.dir.

[21] C. Rankin and F. Brogan, *An element-independent co-rotational procedure for the treatment of large rotations*, ASME J. Pressure Vessel Tchn., 108 (1986), pp. 165–174.

[22] P. Sulatycke and K. Ghose, *Caching-efficient multithreaded fast multiplication of sparse matrices*, in Proceedings of the Parallel Processing Symposium '98, IEEE, Orlando, FL, 1998, pp. 117–123.

[23] P. Wesseling, *A robust and efficient multigrid method*, in Multigrid Methods, W. Hackbusch and U. Trottenberg, eds., Lecture Notes in Math., Vol. 960, Springer, Berlin, 1982, pp. 614–630.

[24] R. Yuster and U. Zwick, *Fast sparse matrix multiplication*, ACM Trans. Algorithms, 1 (2005), pp. 2–13.