Sparse Matrix Assembly on the GPU Through Multiplication Patterns

Rhaleb Zayer*, Markus Steinberger† and Hans-Peter Seidel*
*Max Planck Institute for Informatics
Saarbrücken, Germany
Email: {rzayer,hpseidel}@mpi-inf.mpg.de
†Graz University of Technology
Graz, Austria
Email: markus.steinberger@icg.tugraz.at

***Abstract*—The numerical treatment of variational problems gives rise to large sparse matrices, which are typically assembled by coalescing elementary contributions. As the explicit matrix form is required by numerical solvers, the assembly step can be a potential bottleneck, especially in implicit and time dependent settings where considerable updates are needed. On standard HPC platforms, this process can be vectorized by taking advantage of additional mesh querying data structures. However, on graphics hardware, vectorization is inhibited by limited memory resources. In this paper, we propose a lean unstructured mesh representation, which allows casting the assembly problem as a sparse matrix-matrix multiplication. We demonstrate how the global graph connectivity of the assembled matrix can be captured through basic linear algebra operations and show how local interactions between nodes/degrees of freedom within an element can be encoded by means of concise representation, *action maps*. These ideas not only reduce the memory storage requirements but also cut down on the bulk of data that needs to be moved from global storage to the compute units, which is crucial on parallel computing hardware, and in particular on the GPU. Furthermore, we analyze the effect of mesh memory layout on the assembly performance.**

I. Introduction

The computational effort within most variational problem formulations in general and finite elements in particular, revolves around two major steps: i) the system matrix setup, known as the assembly process, and ii) numerical solution of the resulting algebraic equations. The second step has received considerable attention even before the inception of modern computing [1] and continues to capture the focus of the numerical community. Interest in the assembly step is more recent and has grown out of challenges in high performance computing (HPC). Numerical evidence across various disciplines suggests that the assembly problem weighs heavily on performance and impedes scalability, see e.g., the numerical experimental in [2], [3], [4].

Our work targets the assembly step on modern graphics hardware and is motivated by several observations. **Memory requirements**: existing matrix assembly methods on classical HPC platforms make use of additional adjacency information, e.g [5], or gather scatter operations on worksets, e.g. [6]. The aggregate cost of building, moving, and updating adjacency data poses great challenges on infrastructures with limited memory resources such as graphics hardware and restricts the scope and scale of applications. On the other hand, efficient implementations of standard gather scatter like operation on the GPU, to the best of our knowledge, still rely on preprocessed CPU data. **Availability of direct solvers on the GPU**: while parallel direct solvers have been available for quite some time e.g. [7], their portability to graphics hardware has been challenged by weak parallelism, sparsity pattern fill-in, and random memory access patterns. Recent work, e.g. [8], [9], demonstrates the potential of such solvers on the GPU. **Performance through linear algebra**: There is a recent regain of interest into sparse matrix algebra methods for large graph representations, e.g. [10]. Substantial performance and scalability gains can be achieved by simply optimizing the underlying linear algebra kernels.

In order to accommodate the observations above, our solution departs from classical assembly strategies. We build upon the observation that the inner node-node relations inherent to the assembly processes can be captured through a matrix-matrix multiplication. Our solution does not require expensive intermediate edge-based representations for tracking mesh topology but rather a simple re-casting of the element table as an equivalent sparse matrix, the *mesh matrix*, an ordered element-node connectivity matrix. In order to encode various possible integrations between element nodes, we introduce action maps which act as the hands and arms of the mesh matrix and we show their use for assembly and for performing basic numerical evaluations on meshes in a concise and memory efficient manner. The mesh matrix representation highlights memory access patterns and thus can be tuned for better locality. The reduction of its bandwidth yields substantial performance gains to the assembly process.

As linear algebra primitives such as matrix-matrix multiplication are the back-bone of most numerical code, our approach can be easily incorporated within existing sparse matrix libraries without requiring substantial code modifications. Furthermore, given the tremendous effort to speed up those primitives, any performance gains would naturally profit our assembly approach. To the best of our knowledge, our approach

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

is the first to perform the whole assembly on the GPU without requiring any additional CPU preprocessing or postprocessing.

The rest of this paper is organized as follows, we will cover the related work in section II. The mesh matrix is introduced in section III and its use with action maps for matrix assembly is described in section IV. Techniques for reducing the storage requirement pertaining to mesh matrix are outlined in section V. The technical aspects related to our serial and GPU implementations are discussed in section VI. Aspects related to the optimization of the mesh matrix memory layout are outlined in section VII. Experimental results of our approach and performance comparisons are given in section VIII.

II. Related Work

Iterative solvers can bypass full matrix assembly by performing localized matrix vector products. Nonetheless, the combination of shared memory and the unstructured nature of general meshes gives rise to race conditions as multiple processors attempt to address the same memory location. Early treatments of the problem in [11], [12], [13], avoid race conditions by means of coloring schemes. The idea is to process the mesh elements in sequences (colors) where no two elements are adjacent to each other. In this way, concurrent programming is achieved for blocks of distinct colors and then synchronized globally. The efficiency of the synchronization process depends on the mesh structure. The impact of coloring on memory locality has been recently studied [14], in particular the interplay of the number of colors, vertex blocking and parallelism. In the context of direct solvers, load balancing strategies based on mesh partitioning are commonly used to reduce the computational burden [12]. Parallelism is achieved by forming sub-domain matrices. State-of-the-art partitioning methods such as MeTis [15] are commonly used. Note however, that when partitioning is used, assembly at the sub-mesh level still needs to be performed efficiently. Scalability can be addressed by divide and conquer strategies [5] or speculative coloring [16]. The portability of coloring and partitioning on the graphics process unit (GPU) has been studied by several authors [17], [18], [19]. Existing GPU implementations, e.g. [20], rely on mesh querying data structures to store the topology (vertex, face and edge connectivity information). Furthermore, crucial operations such as sparsity pattern computation are performed on the host. While the use of extensive data structures is commonly adopted and well justified on high performance computing clusters e.g., [5], their use on the GPU restricts the range of applications to moderately sized data-sets.

General purpose methods such as the well established serial *Sparse* function in Matlab [21], directly assemble coordinate triplets into a sparse matrix (please note that despite the age of the reference, the code is up to date). A variant known as *Sparse2*, which capitalizes on a different sorting scheme has been proposed in the SuiteSparse package [22]. Another breed of methods builds upon the remark that the cost of the aforementioned general methods stems from the nature of the compressed formats used during assembly. Instead, alternative formats can be used to build an initial matrix, which is then converted to the standard and computationally efficient formats. A stack based representation has been proposed in [23], hash tables have been used in [24], and index-based sorting was proposed in [25]. To the best of our knowledge, there is no GPU variant of the latter approaches.

III. The Mesh Matrix

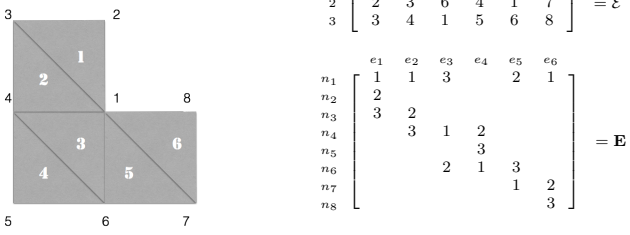
A mesh $\mathcal{M}$ is represented as a collection of n_e elements, $\mathcal{E} = \{\mathbf{e}_1, \ldots, \mathbf{e}_{n_e}\}$, where each element refers to the indices of its respective nodes. In general, meshes are oriented and the orientation is reflected in the order of traversal of the element (up to a cyclic permutation). We denote the table storing node coordinates by the array $\mathbf{P}$ of size $n_p \times d$, where $d = 2; 3$, depending on the dimensional setting of the problem. The elements can be of arbitrary nature (polygonal/polyhedral). However, to ease the understanding, we assume for now that they are all of the same kind.

The element table fully encodes the mesh representation, however it does not reveal its underlying topological structure. We propose to overlay this representation on a sparse matrix $\mathbf{E}$, of size $n_p \times n_e$, while preserving the prescribed element's orientation as follows:

$$\mathbf{E}(\mathbf{e}_i(k), i) = k; \tag{1}$$

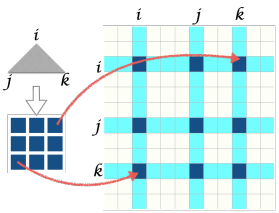
where i spans the elements and k spans the nodes of each element $\mathbf{e}_i$.

Consider the following simple case of the triangulation depicted below: The matrix $\mathbf{E}$ simply amounts to spreading the



columns of $\mathcal{E}$ to span the range of the nodes. Please note, the construction of such a matrix in compressed column format (CSC), or its transpose in compressed row format (CSR) is straightforward and computationally inexpensive. We will now re-examine the assembly in the light of this representation.

The assembly process consists of a local assembly at the individual elements level and a global assembly at the mesh level, as illustrated in the inset. The local assembly produces the contribution of individual elements to the global matrix, and it can be in principle carried out in parallel. The specifics of these local contributions depend on the design of the elements, and the reader is referred to standard desk references, e.g. [26], [27].



978-1-5386-3472-1/17/$31.00 ©2017 IEEE

The global assembly consists of putting the individual contributions together to set up the system matrix. It can be observed that the sparsity structure of the global matrix depends on the connectivity of the underlying mesh. This connectivity can be captured in a different way. Let us examine the generalized vertex-vertex adjacency matrix $\mathbf{S_v} = \overline{\mathbf{E}}\overline{\mathbf{E}}^\top$; where $\overline{\mathbf{E}}$ is the binary form of $\mathbf{E}$.

$$S_v = \begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \\ n_6 \\ n_7 \\ n_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} n_1 & n_2 & n_3 & n_4 & n_5 & n_6 & n_7 & n_8 \end{array} \\ \left[\begin{array}{cccccccc} 5 & 1 & 2 & 2 & & 2 & 2 & 1 \\ 1 & 1 & 1 & & & & & \\ 2 & 1 & 2 & 1 & & & & \\ 2 & & 1 & 3 & 1 & 2 & & \\ & & & 1 & 1 & 1 & & \\ & & & 2 & 1 & 3 & 1 & \\ 2 & & & & & 1 & 2 & 1 \\ 1 & & & & & & 1 & 1 \end{array}\right] \end{array}$$

The nonzero entries of $\mathbf{S}_v$ represent the number of elements common to any given two nodes. The diagonal entries count the number of faces common to a given node. To a certain extent, the multiplication captures the essence of the assembly process and we can simply use the product $\mathbf{E}\mathbf{E}^\top$ to fill the global matrix. The collisions between nonzero values of $\mathbf{E}$ and $\mathbf{E}^\top$ provide the location of the contribution in the local element matrix that needs to be inserted into the global matrix.

IV. Sparse matrix assembly

On parallel computing hardware, in particular on the GPU, memory access is considered the most costly operation. In the following we develop some ideas, which not only reduce the memory storage requirements but also cut down on the bulk of data that needs to be moved from global storage to the compute units. With the example from section III in mind, the sparsity pattern of the matrix product results from the collisions of nonzero entries of $\mathbf{E}$ and its transpose. The outcome of these collisions can be encoded by a linear mapping, which we call an action map. Since we would like to capture the initial counterclockwise orientation of the mesh, we can use a cyclic permutation matrix Q_3 defined as

$$Q_3 = \frac{1}{2}\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}; \tag{2}$$

Please note, that since Q_3 and its powers form a basis for all 3×3 circulant matrices, we can encode a variety of interactions within a face by means of action maps. For instance, the diagonal degree matrix D can be obtained from $Q_3^0 = \mathbf{I_{3\times 3}}$. The graph Laplacian through $Q_3^0 - (Q_3^2 + Q_3)$. If the mesh has boundary, the boundary adjacency can be captured by $Q_3 - Q_3^2$, traversal of positive entries will yield a counterclockwise oriented boundary, negative ones yield clockwise traversal.

To see the impact of action maps on the assembly in finite elements, let us consider the case of the constant strain triangle (CST). The corresponding element contribution will be a 3×3 matrix per triangle, given by

$$\mathbf{k_t} = \begin{pmatrix} \cot\theta_2 + \cot\theta_3 & -\cot\theta_3 & -\cot\theta_2 \\ -\cot\theta_3 & \cot\theta_3 + \cot\theta_1 & -\cot\theta_1 \\ -\cot\theta_2 & -\cot\theta_1 & \cot\theta_1 + \cot\theta_2 \end{pmatrix}.$$

We can see that the pattern of this matrix already resembles that of $Q_3^0 - (Q_3^2 + Q_3)$. Instead of storing the whole 3×3 matrix, we can simply associate a vector holding the cotangent of element angles and use the action map to distribute them during global assembly. In this case only 3 entries need to be stored and loaded instead of 9.

We can take the concept of action maps even further, and look at the case of a polygonal mesh consisting of counterclockwise oriented elements (n-gones) of the same kind. For an element, we associate the action map Q_n defined by the cyclic permutation

$$Q_n = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 \end{pmatrix} \tag{3}$$

Let's consider the case of a quadrilateral mesh. If we wish to capture only relations between diagonally opposed vertices within an element, we can simply use the action map Q_4^2. This would flag ones for diagonally opposed vertices.

For tetrahedral elements, the linear tetrahedron is given by:

$$\mathbf{k_t} = \frac{1}{6}\begin{pmatrix} \sum_1 & -l_{34}\cot\theta_{34} & -l_{24}\cot\theta_{24} & -l_{23}\cot\theta_{23} \\ & \sum_2 & -l_{14}\cot\theta_{14} & -l_{13}\cot\theta_{13} \\ & \text{sym.} & \sum_3 & -l_{12}\cot\theta_{12} \\ & & & \sum_4 \end{pmatrix};$$

where $\sum_i$ infers the sum of non-diagonal entries in row i.

We can regard the tetrahedron as a combination of doubly oriented edges or oriented faces and we can associate the following map $Q_{tet} = Q_4^4 - (Q_4^3 + Q_4^2 + Q_4)$. The use of action maps in this case requires storing only 6 entries per tetrahedron contribution instead of 16.

The action maps serve two purposes. First, they streamline the code by avoiding conditional statements, which is beneficial for concurrent programming as branching is avoided. Second, they reduce memory requirements, which is of utmost importance on platforms with limited memory resources such as graphics hardware.

V. Storage requirements reduction

In practice, Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC) are widely used for performing linear algebra operations. To illustrate the concept behind our storage reduction, we start from the CSC format, which proceeds by storing the matrix values and row indices and uses a column pointer for the locations of column starts. The storage of the mesh matrix of example III is given below.

$$\begin{array}{lll} vals & = [& 1\ 2\ 3\ 1\ 2\ 3\ 3\ 1\ 2\ \ 2\ \ 3\ 1\ \ 2\ \ 3\ 1\ \ 1\ \ 2\ 3] \\ rowind & = [& 1\ 2\ 3\ 3\ 4\ 1\ 1\ 4\ 6\ \ 4\ \ 5\ 6\ \ 1\ \ 6\ 7\ \ 1\ \ 7\ 8] \\ & & \uparrow\quad \uparrow\quad \uparrow\quad \uparrow\quad \uparrow\quad \uparrow \\ colptr & = [& 1\quad\ 4\quad\ 7\quad\ 10\quad\ 13\quad\ 16\quad\ ] \end{array}$$

For the sake of argument, let's assume the nonzero values are stored in double precision (double) and the indices are stored as integers (int). The storage requirement for a general matrix within this format amounts to $Nz(1{\cdot}double{+}1{\cdot}int){+}ncol{\cdot}int$.

Here, we build upon an idea of [28]. The CSC representation is not unique. The row indices and the corresponding values can be reordered within a given column without changing the matrix. We capitalize on this observation and show how the storage requirements can be cut down. Let us examine our example again. In the first and second column, the information contained in the values is redundant since it coincides with the order of the traversal of the rows along columns 1 and

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

2. Now, if we re-order the row indices of the third column according to their entries in $vals$ as shown below (in bold), the values corresponding to row 3 will become also redundant.

$vals$	= [	1	2	3	1	2	3	**1**	**2**	**3**	1	2	3	1	2	3	1	2	3]
$rowind$	= [	1	2	3	3	4	1	**4**	**6**	**1**	6	4	5	7	1	6	1	7	8]
		↑			↑			↑			↑			↑			↑		
$colptr$	= [	1			4			7			10			13			16		]

Once we perform this for all columns, the role of the values becomes obsolete. In fact, we know how many entries per column we have from the $colptr$ and we need simply to traverse them in the order of appearance.

So in practice, we do not need to store the mesh matrix but only its sparsity pattern, namely $rowind$ and $colptr$. Furthermore, if all elements are of the same type the $colptr$ can also be omitted and only inferred. Again this memory storage reduction has a practical value in concurrent infrastructures with limited memory resources.

VI. Assembly Implementation

a) serial: The sparse assembly based on multiplication patterns described in sections III and IV can be easily incorporated on top of existing sparse matrix-matrix multiplication kernels. As many existing codes such as the one available in Matlab [21] or csparse [29] rely on the classical SpGEMM description [28]. In general for the computation of the product $C = AB$ in (CSC) format, both A and B need to be accessed by columns, and the resulting matrix C is created one column at a time [29]. For the j-th column of B, $B_{*j} = [b_{1j} \cdots b_{kj}]^\top$, the column C_{*j} is given by

$$C_{*j} = \sum_{i=1}^{k} A_{*i} b_{ij} = [A_{*1} \cdots A_{*k}] \begin{bmatrix} b_{1j} \\ \vdots \\ b_{kj} \end{bmatrix}. \qquad (4)$$

In practice, the serial algorithm operates in two stages. A first stage where the number of nonzeros of the resulting matrix is evaluated and a second stage where the assignment is actually performed. Please note, that when the underlying mesh does not change within a simulation, the first stage needs to be performed only once. This yields considerable gains, knowing that the cost of the first stage in general is about $20 - 30\%$ of the overall computation time.

b) GPU: To parallelize the assembly process, we need to parallelize the underlying linear algebra operation. The core of our assembly process is formed by sparse matrix-matrix products with action maps. In order to achieve high performance, we aim to integrate action maps into the most efficient sparse BLAS routines. There has been a consistent research effort on the parallelization of sparse matrix-matrix products, e.g., [10], [30], [31], [32].

Our approach takes advantage of the algorithmic description of bhSparse [31] for spGEMM ($\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$). While our assembly approach could be built on top of other libraries, bhSparse achieved the best performance in our tests. The biggest challenges for sparse matrix-matrix multiplication include (1) the number of non-zero entries in the resulting sparse matrix is unknown, (2) parallel insertion operations for the resulting sparse matrix have a high potential cost, and (3) load imbalances for the general spGEMM can severely slow down the computation on parallel devices like the GPU. bhSparse tackles these issues in a four stage approach:

The first stage computes in parallel an upper bound for the number of non-zeroes in each column of the result matrix. For each estimate, the algorithm runs through all non-zero entries in the corresponding column in matrix $\mathbf{A}$ and accounts for the nonzeros of the column in $\mathbf{B}$ whose elements will be multiplied with entries found in $\mathbf{A}$. *The second stage* bins the columns based on the expected number of nonzeros for the columns of $\mathbf{C}$, as computed in (i). This binning step ensures that the most fitting algorithm for the expected nonzero count can be chosen and load balancing on the GPU becomes trivial. *The third stage* chooses for each column of $\mathbf{C}$ one of three methods for computing the final entries: the heap method [33], the ESC method [34], or the merge method [30]. The heap method and the ESC method can be carried out using efficient scratch-pad memory (i.e. shared memory in CUDA, local memory in OpenCL). The merge method requires multiple kernel launches and memory allocations. *The final stage* rearranges the results of the previous step in the final CSC format.

For our mapped matrix-matrix multiplication we perform the same stages. The first two stages work on the $rowind$ and $colptr$ only. Thus, no modifications are required for the general case. In case the $colptr$ has been omitted, we can again unroll the loop constructs in those kernels to speed up the computation. In the third stage, we replace the multiplication with the action map lookup based on the traversal order of the elements. To implement action maps on the GPU, we use textures. Textures use a separate cache that is similar in speed to scratch-pad memory. As scratch-pad memory is heavily used by the sparsity pattern multiplication itself, we refrain from using it to store the action maps. Because action maps are small, their entire contents usually stays in cache after first use. Thus, the overhead over a simple matrix-matrix multiplication is negligible. Additionally, for nonlinear or time dependent problems we can build on the fact that assembly is carried out in each iteration, thus we can perform binning only once and reuse the bins. In case the mesh is adjusted from one iteration to the next, we only need to re-bin for those columns that were affected by the mesh operation. In this way, we can save up to 40% of the overall assembly cost.

VII. Impact of data layout

Conventional sparse matrix assembly algorithms proceed by data insertion at random locations, which often results in poor memory access patterns. To illustrate this problem, let us consider the example in Fig. 1. Two layouts of the same mesh are considered, an original natural ordering with a poor layout and a cache friendly version obtained by means of the reverse CuthillMcKee algorithm (RCM) for node reordering [35] followed by a re-ordering of faces based on the smallest index of their nodes.

We can readily observe that there is a performance improvement for conventional methods such as Matlab *sparse* [36]

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

and the recent Fsparse [25] as the layout becomes more cache friendly. However the improvement is more dramatic for matrix multiplication based assembly.

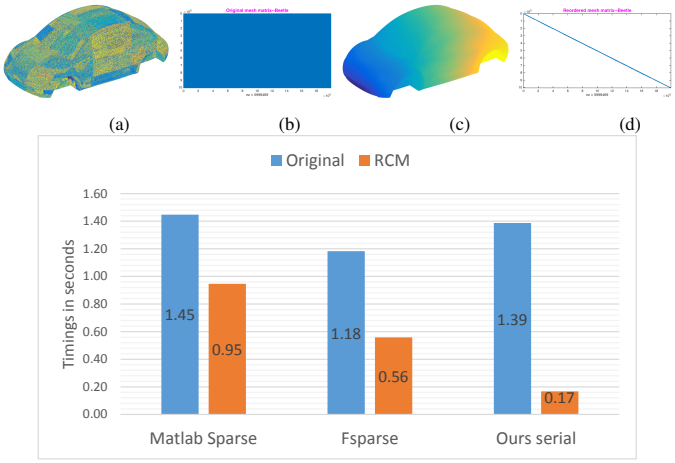


Fig. 1: Car mesh colored based on the original element ordering (a), and the Layout of its element-node matrix $\mathbf{E}$ (b). The same mesh is shown in (c) and (d) after an RCM re-ordering. The chart (bottom) shows the impact of the reordering on the assembly performance for different methods. (Assembly applied to **2M** elements using a linear triangle (**1** dof/node).

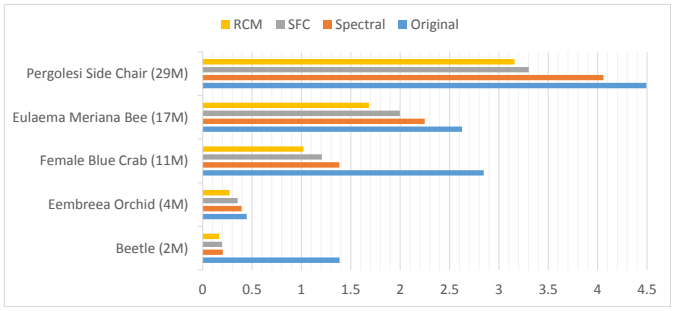


Fig. 2: Performance (in sec.) of our approach on the assembly of linear triangles (1 dof/node) for different mesh orderings.

To assess the effect of mesh reordering, we tested on various data sets from different academic and industrial sources. In our tests the original data (natural ordering), was reordered using different approaches: i) spectral, ii) space filling curves (Morton curves), and iii) RCM. i) Spectral ordering seeks to reduce the envelope of the graph Laplacian through the minimization of the sum of squared difference of node labels. By restating the problem as a continuous problem, a solution can be obtained as the eigenvector associated with the smallest positive eigenvalue of the Laplacian a.k.a Fiedler vector [37]. For problems of significant size, calculations need to be carried out in a multilevel fashion. In our experiment we adopt an algebraic multi-grid strategy driven by an aggressive Galerkin product based simplification of the Laplacian matrix [38], [39]. ii) Space filling curves (SFC) arise when attempting a linearization of higher dimensional spaces [40]. Two commonly used ordering are the Morton and the Peano-Hilbert orderings, which exhibit good locality preserving behavior. iii) The RCM ordering method originated from the need to reduce bandwidth for use with numerical solvers. It can be regarded as an iterative variant of the basic breadth first search (BFS). It first tags a given node as first node. The unnumbered neighbors of any ordered node are then ordered by increasing degree. The final order is then reversed. Two crucial elements in this algorithm are finding a starting node and breaking ties when degrees are similar. The variant proposed by George and Liu [35] obtains the starting node iteratively by searching for a pseudo-peripheral node. Ties are broken based on the initial ordering.

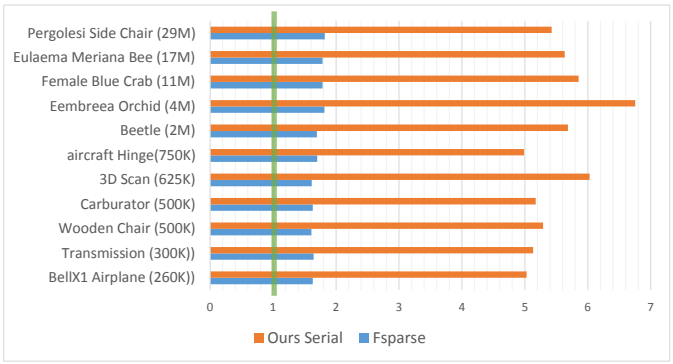
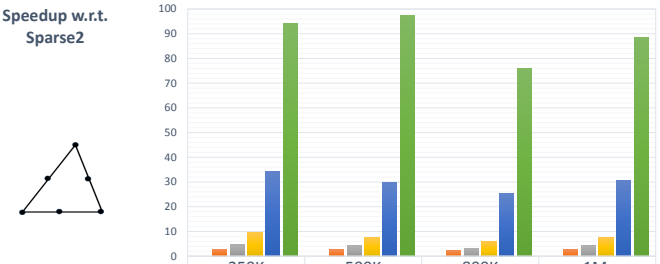


Fig. 3: Comparison of the speedup of Fsparse and our approach w.r.t. sparse2 (green line), for the linear triangle, with 1 DoF per node. Meshes were reordered using RCM.



Mesh size (#triangles)	250K	500K	800K	1M
Sparse2 (Timing)	3.76	6.83	9.11	15.07
MFEM (serial)	2.89	2.59	2.24	2.79
Ours CPU	4.70	4.32	3.28	4.33
Ours CPU (w. reordering)	9.40	7.42	5.95	7.39
Ours GPU	34.18	29.70	25.31	30.76
Ours GPU (w. reordering)	94.00	97.57	75.92	88.65

Fig. 4: Assembly of the stiffness matrix for the 6 nodes triangle with 2DoF per node for a rectangular plate mesh. Relative speedup of all approaches is shown w.r.t Sparse2 timings (in sec.).

Typical performance measurements are summarized for meshes of sizes ranging from 2 to 29 million elements in Fig 2. To reduce the effects of other factors we restricted the experiments to the case of basic linear triangles with one degree of freedom per node, see e.g. [27]. The data suggests that ordering can overshadow the real algorithmic performance and operation counts. The tighter matrix bandwidth generated by the RCM reordering, see Fig 1-d, is reflected in the performance.

The observed speedup can be explained algebraically. In fact, for a rectangular sparse matrix $\mathbf{A}$, the bandwidth of a column k, is the maximum width between its nonzero entries.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

The column bandwidth, β of the matrix $\mathbf{A}$ can be then defined as the maximum of its columns' bandwidths. It ensues that the matrix $\mathbf{A}\mathbf{A}^\top$ has a bandwidth of $b \leq \beta - 1$ [41]. This result implies that during multiplication, the maximum bandwidth used is bounded by the column bandwidth of $\mathbf{A}$.

VIII. Experimental Results

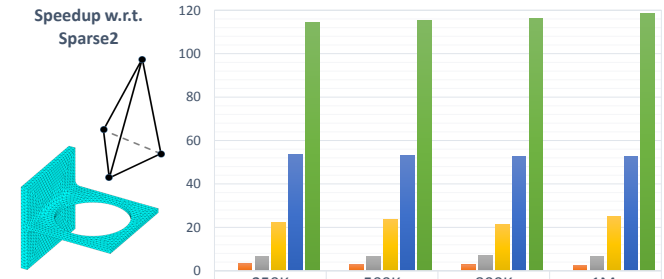
The hardware setup for our experiments consists of an Intel Xeon E5-2637 v3 CPU running at 3.50GHz, 32GB of memory and an NVIDIA Tesla K40m with 2880 compute cores and 12GB of memory running at 745MHz. For testing purposes, we used the CUDA version of our implementation. However, using our OpenCL implementation, our algorithm can also run on different graphics hardware.

We tested against the well established serial method, namely the *Sparse2* function from SuiteSparse package [22], which directly assembles the coordinates triplets into a sparse matrix. A more recent assembly method, which takes advantage of the multicore structure of modern CPUs was reported under the name *Fsparse* [25]. The chart in Fig.3 compares the performance of both approaches to ours on a toy problem, namely, the assembly of the linear triangle (CST) with one degree of freedom per node. Our serial implementation achieves speed-ups ranging from 5 to 7 w.r.t Sparse2. The speedup of the serial Fsparse on the other hand stays below 2. The reordering time is not included in the timings. Nonetheless reordering methods such as RCM are relatively cheap. In fairness to both methods, they are more general than our approach, in the sense that they can build arbitrary matrices.

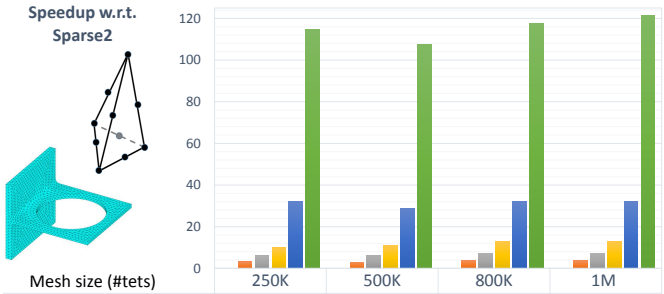
To evaluate the performance of our approach on more practical settings, we measured the performance of our approach on the 6 node triangle with 2 degrees of freedom on a rectangular plate mesh, Fig. 4, and the linear (4 points) as well as the quadratic (10 points) tetrahedron with 3 degrees of freedom on the mesh of a bracket with a hole, Fig. 5. In these tests, we used meshes with increasing element count, starting from 250K and finishing at 1M. For the rectangular plate experiment, we use the performance of *Sparse2* from the SuiteSparse package [22] as a reference timing. Additionally, we report the serial performance of the specialized finite element package MFEM [42] from LLNL for reference (According to the authors of the latter package, they have explored using sparse matrix-matrix multiplication (SpGEMM) for finite-element assembly, using Hypre's implementation of that computational kernel). With our approach, the speedups achieved on the GPU remain in the range of two orders of magnitude. The speedup of our serial implementation on the CPU ranges from 10 to 25.

Benchmarks on the assembly of structural shell elements with 6 degrees of freedom per nodes, in this case the standard DKT (Discrete Kirchhoff Triangular) element [43], [44], reveals a consistent speedup across varying meshes from different industrial sources. The speedup ranges from 125 to 168 on the GPU and reaches about 15 on the CPU for our serial implementation.

As discussed in the related work section, existing assembly approaches on the GPU rely on serial preprocessing or

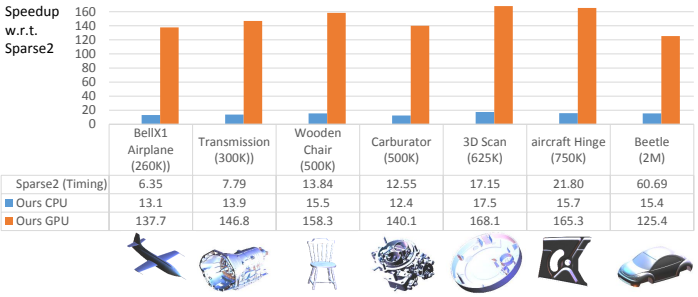


Mesh size (#tets)	250K	500K	800K	1M
Sparse2 (timing in s.)	4.42	9.80	15.08	21.24
MFEM (serial)	3.4	3.1	2.7	2.6
Ours CPU	6.7	6.7	7.1	6.4
Ours CPU (w. reordering)	22.3	23.6	21.5	25.0
Ours GPU	53.7	52.9	52.5	52.8
Ours GPU (w. reordering)	114.4	115.5	116.4	118.6



Mesh size (#tets)	250K	500K	800K	1M
Sparse2 (timing)	20.0	47.3	87.7	115.4
MFEM (serial)	3.1	3.0	4.0	3.7
Ours CPU	6.0	6.2	7.3	7.3
Ours CPU (w. reordering)	9.9	10.9	12.9	12.7
Ours GPU	32.1	28.8	32.1	32.2
Ours GPU (w. reordering)	114.5	107.3	117.7	121.6

Fig. 5: Assembly of the stiffness matrix for the 4-nodes (top), and 10-nodes (bottom) tetrahedron, 3DoF per node, for the bracket with hole shown to the left. Relative speedup of all approaches is shown w.r.t Sparse2 timings (in seconds).



	BellX1 Airplane (260K))	Transmission (300K))	Wooden Chair (500K)	Carburator (500K)	3D Scan (625K)	aircraft Hinge (750K)	Beetle (2M)
Sparse2 (Timing)	6.35	7.79	13.84	12.55	17.15	21.80	60.69
Ours CPU	13.1	13.9	15.5	12.4	17.5	15.7	15.4
Ours GPU	137.7	146.8	158.3	140.1	168.1	165.3	125.4

Fig. 6: Assembly of the stiffness matrix for shells (Discrete Kirchhoff Triangular element DKT [43], [44]) 6 DoF per node for different meshes. Relative speedup of all approaches is shown w.r.t Sparse2 timings (in seconds).

postprocessing steps. In table I, we compare the best performing method of [18], namely SharedNZ to ours. Their test data is supplied with preprocessing results (partitioning and coloring) and their timing does not include those preprocessing steps.

IX. Conclusion

In this paper we demonstrated a simple, yet efficient approach to sparse matrix assembly based on sparse multiplication

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

patterns. We analyzed the impact of memory layout on performance and provided the concept of action maps to reduce the memory footprint of our underlying representation. The experimental results on the CPU and on the GPU reveal considerable gains in performance. The simplicity and modularity of our approach makes it suitable for integration into various platforms.

TABLE I: Sparse assembly comparison, timings in seconds

	Assembly	
	SharedNZ [18]	Ours (GPU)
2D Mesh ($1.2M\Delta$)	0.14	0.096

REFERENCES

[1] V. N. Faddeeva, *Computational methods of linear algebra / by V.N. Faddeeva ; authorized translation from the Russian by Curtis D. Benster*. Dover New York, 1959.

[2] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable hybrid mpi/openmp unstructured finite element model," *Computers & Fluids*, vol. 110, pp. 227 – 234, 2015, parCFD 2013.

[3] N. Jansson, J. Hoffman, and M. Nazarov, "Adaptive simulation of turbulent flow past a full car model," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov 2011, pp. 1–8.

[4] M. Jehl, A. Dedner, T. Betcke, K. Aristovich, R. Klfkorn, and D. Holder, "A fast parallel solver for the forward problem in electrical impedance tomography," *IEEE Transactions on Biomedical Engineering*, vol. 62, no. 1, pp. 126–137, Jan 2015.

[5] L. Thébault, E. Petit, and Q. Dinh, "Scalable and efficient implementation of 3d unstructured meshes computation: A case study on matrix assembly," *SIGPLAN Not.*, vol. 50, no. 8, pp. 120–129, Jan. 2015.

[6] R. P. Pawlowski, E. T. Phipps, A. G. Salinger, S. J. Owen, C. M. Siefert, and M. L. Staten, "Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, part ii: Application to partial differential equations," *Sci. Program.*, vol. 20, no. 3, pp. 327–345, Jul. 2012.

[7] X. Li, J. Demmel, J. Gilbert, iL. Grigori, M. Shao, and I. Yamazaki, "SuperLU Users' Guide," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-44289, September 1999, http://crd.lbl.gov/ xiaoye/SuperLU/. Last update: August 2015.

[8] S. C. Rennich, D. Stosic, and T. A. Davis, "Accelerating sparse cholesky factorization on gpus," in *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3 '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 9–16.

[9] Nvidia, "CUDA toolkit documentation v8.0," 2016.

[10] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, Sept 2013, pp. 1–2.

[11] P. Berger, P. Brouaye, and J. C. Syre, "A mesh coloring method for efficient MIMD processing in finite element problems," in *International Conference on Parallel Processing, ICPP'82, August 24-27, 1982, Bellaire, Michigan, USA*, 1982, pp. 41–46.

[12] C. Farhat and L. Crivelli, "A general approach to nonlinear fe computations on shared-memory multiprocessors," *Comput. Methods Appl. Mech. Eng.*, vol. 72, no. 2, pp. 153–171, Feb. 1989.

[13] T. J. R. Hughes, R. M. Ferencz, and J. O. Hallquist, "Large-scale vectorized implicit calculations in solid mechanics on a cray x-mp/48 utilizing ebe preconditioned conjugate gradients," *Comput. Methods Appl. Mech. Eng.*, vol. 61, no. 2, pp. 215–248, Mar. 1987.

[14] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proceedings of SC '14*. IEEE Press, 2014, pp. 945–955.

[15] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0," http://www.cs.umn.edu/ metis, University of Minnesota, Minneapolis, MN, 2009.

[16] E. G. Boman, D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, and F. Manne, *A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 241–251.

[17] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda," *J. Parallel Distrib. Comput.*, vol. 69, no. 5, pp. 451–460, May 2009.

[18] C. Cecka, A. J. Lew, and E. Darve, "Assembly of finite element methods on graphics processors," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 5, pp. 640–669, 2011.

[19] I. Z. Reguly and M. B. Giles, "Finite element algorithms and data structures on graphical processing units," *Int. J. Parallel Program.*, vol. 43, no. 2, pp. 203–239, Apr. 2015.

[20] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly, "Pyop2: A high-level framework for performance-portable simulations on unstructured meshes," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. IEEE Computer Society, 2012, pp. 1116–1123.

[21] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 1, pp. 333–356, 1992.

[22] T. Davis, "SuiteSparse: A suite of sparse matrix packages," http://www.cise.ufl.edu/ davis/, 2016.

[23] N. Jansson, *Optimizing Sparse Matrix Assembly in Finite Element Solvers with One-Sided Communication*. Springer, 2013, pp. 128–139.

[24] M. Aspnäs, A. Signell, and J. Westerholm, "Efficient assembly of sparse matrices using hashing," in *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, ser. PARA'06. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 900–907.

[25] S. Engblom and D. Lukarski, "Fast matlab compatible sparse assembly on multicore computers," *Parallel Comput.*, vol. 56, no. C, pp. 1–17, Aug. 2016.

[26] R. H. MacNeal and R. L. Harder, "A proposed standard set of problems to test finite element accuracy," *Finite Elements in Analysis and Design*, vol. 11, pp. 3–20, 1985.

[27] R. D. Cook, D. S. Malkus, M. E. Plesha, and R. J. Witt, *Concepts and Applications of Finite Element Analysis*. John Wiley & Sons, 2007.

[28] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," vol. 4, no. 3, pp. 250–269, 1978.

[29] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[30] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015.

[31] W. Liu and B. Vinter, "A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, no. C, pp. 47–61, Nov. 2015.

[32] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *Proceedings of ISC High Performance 2015*, M. J. Kunkel and T. Ludwig, Eds. Springer, 2015, pp. 48–57.

[33] J. R. Gilbert, W. W. Pugh Jr, and T. Shpeisman, "Ordered sparse accumulator and its use in efficient sparse matrix computation," Nov. 9 1999, uS Patent 5,983,230.

[34] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[35] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981.

[36] MATLAB, *version (R2015b)*. Natick, Massachusetts: The MathWorks Inc., 2015.

[37] S. T. Barnard, A. Pothen, and H. D. Simon, "A spectral algorithm for envelope reduction of sparse matrices," in *Proceedings of the 1993*

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

ACM/IEEE Conference on Supercomputing, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 493–502.
[38] K. Stuben, "Algebraic multigrid (AMG): an introduction with applications," GMD, Sankt Augustin, Germany, GMD Report 70, 1999.
[39] Y. F. Hu and J. A. Scott, "A multilevel algorithm for wavefront reduction," *SIAM Journal on Scientific Computing*, vol. 23, no. 4, pp. 1352–1375, 2001.
[40] H. Sagan, *Space-filling curves*, ser. Universitext. Springer-Verlag, New York, 1994.
[41] A. Björck, *Numerical methods in matrix computations.* Texts in Applied Mathematics 59. Cham: Springer. xvi, 800 p. , 2015.
[42] "MFEM: Modular finite element methods," mfem.org, 2016.
[43] J. Stricklin, W. Haisler, P. Tisdale, and R. Gunderson, "A rapidly converging triangular platelement." *AIAA J.*, vol. 7, pp. 180–181, 1969.
[44] J.-L. Batoz, K.-J. Bathe, and L.-W. Ho, "A study of three-node triangular plate bending elements," *International Journal for Numerical Methods in Engineering*, vol. 15, no. 12, pp. 1771–1812, 1980.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE