

A GPU-Adapted Structure for Unstructured Grids

Rhaleb Zayer Markus Steinberger Hans-Peter Seidel

Max Planck Institute for Informatics, Saarland Informatics Campus, Germany

Abstract

A key advantage of working with structured grids (e.g., images) is the ability to directly tap into the powerful machinery of linear algebra. This is not much so for unstructured grids where intermediate bookkeeping data structures stand in the way. On modern high performance computing hardware, the conventional wisdom behind these intermediate structures is further challenged by costly memory access, and more importantly by prohibitive memory resources on environments such as graphics hardware. In this paper, we bypass this problem by introducing a sparse matrix representation for unstructured grids which not only reduces the memory storage requirements but also cuts down on the bulk of data movement from global storage to the compute units. In order to take full advantage of the proposed representation, we augment ordinary matrix multiplication by means of action maps, local maps which encode the desired interaction between grid vertices. In this way, geometric computations and topological modifications translate into concise linear algebra operations. In our algorithmic formulation, we capitalize on the nature of sparse matrix-vector multiplication which allows avoiding explicit transpose computation and storage. Furthermore, we develop an efficient vectorization to the demanding assembly process of standard graph and finite element matrices.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques — Graphics data structures and data types. I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems. I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors. G.1.3 [Mathematics of Computing]: Numerical Linear Algebra — Sparse, structured, and very large systems. G.1.0 [Mathematics of Computing]: General—Parallel algorithms.

1. Introduction

Unstructured grids arise across a variety of disciplines spanning simulation, manufacturing, health care, and entertainment, thus harnessing their unstructured nature is of paramount importance as it deeply impacts the overall performance of algorithms. While structured grids can easily take advantage of existing high performance linear algebra machinery, unstructured grids are hampered by the need for intermediate traversal data structures. Traditionally, grid data is stored as a table of cells or elements. This basic representation can be especially efficient when coupled with sparse matrix formulations as has been demonstrated in finite elements codes over decades, e.g., [Tay70, PT95], but falls short when connectivity queries are required. Such queries can be facilitated by edge-centric neighborhood structures, e.g., [Bau72, Män89]. Descendants of these graph-like representations, e.g., half-edge, have shaped the algorithmic landscape in such a way that familiarity with them has become almost a prerequisite for understanding algorithms. Differential forms offer an alternative tensor based representation, e.g., Abrams et al. [AMR88]. This formalism which is suitable for fields such as electromagnetism and gravitation, where forms of different orders naturally correspond to measurable quantities such as potentials, electric fields, and magnetic fluxes, might

not be well justified for general purpose use due to the cost of storing and updating the corresponding boundary operators. While the above mentioned representations improve accessibility they lead to redundant data creation, and the aggregate cost creating and maintaining them is prohibitive to many applications. More importantly, as the computing landscape is drastically changing towards ubiquitous parallelism, data movement and the creation and update of extensive indirection lists poses challenging problems and there is pressing need for rethinking the grid interfacing problem from the ground up.

The key to supplying a simple interface lies in choosing the right abstraction level. Our aim is to provide a representation that is familiar to a majority of practitioners and enthusiasts and to allow them to manipulate it in an intuitive way. Our algorithmic formulation is centered around objects like matrices, vectors and permutations, and acts by means of operations like sparse matrix-vector multiplication, sparse matrix-matrix multiplication, and maps. This linear algebra flavored representation serves several purposes: (a) *compactness and readability*. Algorithms are concise and easy to interpret. Dispensing with intermediate data structures reduces code bloat and broadens accessibility. (b) *reusability*. The same machinery used for numerical optimization can be used for mesh

processing. (c) *performance*. Data access patterns can deeply weigh on cache and memory related performance. Array-based algorithms bring forward data access patterns and can be readily optimized.

The central elements in our abstraction are the mesh matrix, which casts the topological information encapsulated in the basic cell table into a lean sparse matrix representation, and action maps, which act as a vehicle for translating cell processing tasks into matrix algebra operations. Throughout this abstraction, algorithms can be formulated in the clear and concise language of linear algebra. In this way, improving code performance translates into optimizing algebra operations. In particular, vectorization amounts to the parallelization of key linear algebra primitives. In this respect, advantage can be taken of the tremendous efforts made by manufacturers and the numerical computing community to streamline these primitives on graphics hardware, e.g., [BG09, NVI15, YT12, RG15]. More importantly, mesh management and numerics can be performed without the need for duplicating and transferring data across different structures which is of utter importance on high performance hardware. This importance is accentuated on graphics hardware where memory resources are limited.

A crucial but often neglected aspect when performing computations on meshes is the assembly of linear systems into sparse matrices. Numerical evidence across various disciplines suggests that the assembly cost weighs heavily on performance and impedes scalability [GLG*15, JHN11, JDB*15] especially when the matrix has to be re-assembled several times as in dynamic and nonlinear settings. In graphics, these scenarios are commonly encountered in simulation and animation [PO09, HLSO12]. To address the assembly problem, we develop an efficient approach for constructing standard graph matrices on meshes, e.g., adjacency and uniform Laplacian and extend them to standard finite element matrices.

In this paper, we make the following contributions:

- A lean general purpose sparse matrix representation for arbitrary meshes.
- Action maps for recasting standard mesh operations in a linear algebra formalism
- Efficient numerical routines for mesh handling on serial and parallel high performance hardware

We are aware that a single structure cannot be a Swiss-knife for all problems—as reflected by the myriad of existing specialized linked lists. We emphasize that the goal of this work is not to mimic existing half-edge like operations but rather reformulate problems within a different mind set. The algebraic operations described herein form the building blocks of MeshBLAS [ip17], which takes inspiration from BLAS (basic linear algebra subprograms), and aims to provide standardized subroutines for mesh management.

In the remainder of this paper, we will first discuss previous work (Section 2), which we restrict to closely related approaches given the wide scope of topics under investigation. Then, we outline the construction of the mesh matrix in Section 3. The corresponding storage format and memory footprint are analyzed in Section 4. Action maps for numerical computations on meshes are introduced in Subsection 5.1 with focus on sparse matrix-vector multiplication. We extend these maps to the case of sparse matrix-matrix multiplication (Subsection 5.2). The construction of standard graph

and finite element matrices is outlined in Section 6. The parallelization of linear algebra routines and details of our GPU implementation are outlined in Section 7. In Section 8, we develop linear algebra operation which correspond to standard topological operations commonly used in mesh simplification. Numerical results on a set of practical scenarios are presented in Section 9.

2. Related work

“Plato taught that we do not learn new things; we merely remember things we have forgotten” [Wor81]. This particularly holds for matrix representations of graph-like structures, which can be traced back to early electrical circuits [Kir47]. The importance of this graph-matrix analogy has been recognized early on in graph theory [Har67], but it neither translated into practical solutions, nor widespread use, due to the lack of efficient representations of sparse arrays at the time. This recoil has been echoed across a variety of disciplines. In graphics and vision, the graph-based mesh representation of Baumgart [Bau72] known as the *winged-edge* data structure, which corresponds to the mathematical notion of *combinatorial maps* [Edm60], has shaped how meshes are apprehended. Descendants of this representation differ mainly in the amount of stored data and its organization, e.g., quad-edge [GS85] and half-edge [Män89, Lie94, CKS98, Ket98, BSBK02]. In order to cope with high frame rates and limited transmission bandwidth, pioneering efforts attempted to reduce vertex repetitions as well as memory footprint based on the concept of generalized triangle meshes (stripes) [Dee95, Cho97, Hop99]. Most of the work on the subject, see e.g., the survey [MLDH15], assumes triangle mesh connectivity and does not extend naturally to arbitrary unstructured grids. As the majority of these methods are built with either rendering or compression in mind, their suitability for more general settings has yet to be confirmed.

In linear algebra, systems are commonly represented as matrices. System variables are often loosely coupled, therefore, the canonical full matrix is set aside in favor of a sparse representation which better suits memory requirements. As such systems are ubiquitous in science and engineering, there has been a steady effort within the numerical computing community to represent and process sparse matrices efficiently, see e.g., [GMS92, Dav06] and the references therein. These advances combined with the rise of big data, e.g., [CDG*08], are behind the most recent regain of interest in array-based methods for large graph analysis, e.g. [MBB*13].

The availability of efficient sparse matrix representations revived interest in computational aspects of differential forms on simplicial complexes [Bos98]. Modular frameworks such as, e.g. [CRW05], allow the construction of p -forms and the discretization of higher order finite elements within this formalism. Similar developments followed in computer graphics motivated by ideas from algebraic topology and exterior calculus [GY03, DKT06, DMPS07]. Many existing implementations, to the best of our knowledge, require the half-edge representation alongside sparse matrices. The resulting operators induce a large memory footprint which impedes deployment in high performance computing environments, especially, on graphics processing units (GPUs). Moreover, the adaption of these representations for performing basic operations in mesh management is challenging as it requires familiarity with exterior calculus

and/or algebraic topology. The interpretation of forms is more intuitive in disciplines where they are naturally associated with physical quantities as in electromagnetism, or relativistic mechanics where the concept originated from, e.g. [MTW73].

More pragmatic approaches have been adopted in the finite element method since its inception. Existing software packages offer ingenious ways of coupling the basic face table with sparse matrices for the treatment of a wide range of multi-physics mesh-related problems, e.g., [PT95, Com16]. The combination of the face table and sparse matrices has been also used in geometry processing applications, e.g., [Zay07]. However, in these settings, the mesh representation is adapted to existing standard linear algebra operations which unfortunately leads to the creation of redundant data and suboptimal memory use.

Instead, in this paper, we will adapt linear algebra to the mesh representation; with special focus on two key sparse matrix algebra primitives, namely, sparse matrix vector multiplication (*SpMV*), and sparse matrix-matrix multiplication (*SpMM*). For an overview of standard methods in the serial setting, the reader is referred to [GMS92, Dav06]. On graphics hardware, there readily exists a set of efficient implementations for *SpMV*, e.g., [BG09, BDO12]. Empirical evidence suggests that the performance of the transposed matrix multiplication is often about ten times slower than direct matrix multiplication. This contrast with the steady performance in serial implementations suggests challenging aspects of sparse matrix algebra vectorization. Still, more challenging is the sparse matrix-matrix multiplication which has received considerable attention recently. Significant performance gains have been reported by several authors, e.g., [Dem12, GHS*15, LV15].

Despite great strides in solving linear systems, the problem of assembling sparse matrices themselves still poses great challenges, for instance, direct solvers were abandoned altogether due to assembly cost as in [TMDK15, DMZ*16]. For serial matrix assembly it can be observed that a large part of the cost stems from the nature of the standard compressed matrix storage formats [GMS92, Dav]. As a remedy, alternative representations can be used to build an initial matrix, which can be then converted to more computationally efficient formats. In this respect, representations based on stacks [Jan], hash tables [ASW06], and index-based sorting [EL14] have been proposed. On parallel architectures, the challenge stems from race conditions as multiple processors attempt to address the same memory location. While there has been growing interest in speeding up assembly on the GPU [WBS*13, TWT*16], these methods still require additional data structures to store the topology (vertex, face and edge connectivity information). Crucial operations such as memory allocation is performed on the CPU in [TWT*16]. In our view, assembly though extensive data structures as commonly done on high performance computing clusters e.g., [TPD15], is not suitable for the GPU as it restricts the range of applications to moderately sized data-sets.

3. The mesh matrix

For a given unstructured grid \mathcal{M} , the corresponding cell or face table reads $\mathcal{F} = \{\mathbf{f}_1, \dots, \mathbf{f}_{n_f}\}$, where n_f is the number of faces. Each polygonal face \mathbf{f}_i groups the vertex indices of its summits as illustrated in Figure 1. For an oriented mesh, the orientation is reflected

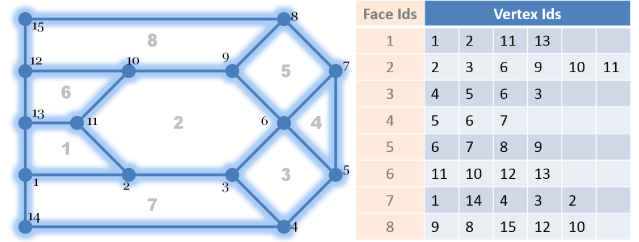


Figure 1: The connectivity of a simple mesh (left) and its face table representing counter-clockwise oriented faces (right).

in the traversal order of the face (up to a cyclic permutation). The vertex coordinates are generally stored in a separate array \mathbf{P} of size $n_v \times 3$, where n_v is the number of vertices.

The face table fully encodes the mesh connectivity but does not explicitly reveal its underlying topological structure. We propose to overlay this representation on a sparse matrix while preserving the prescribed face orientation. We do so by laying out faces along columns; In each column, the location of the face summits are set to their order in the face. This introduces the *mesh matrix* representation. More formally, this is the sparse matrix $\mathbf{M}_{n_f \times n_f}$ defined by the location and values of its nonzero elements:

$$\mathbf{M}(\mathbf{f}_i(k), i) = k; \quad (1)$$

where i spans the faces and k spans the elements of each face \mathbf{f}_i . An example of a simple mesh and its face table is given in Figure 1. The associated mesh matrix representation \mathbf{M} is given by

$$\begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ r_9 \\ r_{10} \\ r_{11} \\ r_{12} \\ r_{13} \\ r_{14} \\ r_{15} \end{matrix} \begin{bmatrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ 1 & & & & & & 1 & \\ 2 & 1 & & & & & 5 & \\ & 2 & 4 & & & & 4 & \\ & & 1 & & & & 3 & \\ & & 2 & 1 & & & & \\ & 3 & 3 & 3 & 1 & & & \\ & & & 2 & 2 & & & \\ & & & & 3 & & 2 & \\ & 4 & & & 4 & & 1 & \\ & 5 & & & & 2 & 5 & \\ 3 & 6 & & & & 1 & & 4 \\ & & & & & 3 & & 4 \\ 4 & & & & & 4 & & \\ & & & & & & 2 & \\ & & & & & & & 3 \end{bmatrix};$$

This representation brings forward the topological structure of the mesh. Faces neighboring a given vertex (vertex fan) line up along its corresponding row. In principle, classical traversal operations can also be performed on the mesh matrix however, this is not the aim of the current contribution. The full face table can be directly recovered from the matrix by simply scanning the columns of the matrix and permuting the nonzeros values to preserve the face orientation without requiring a global or local sorting.

We denote by $\bar{\mathbf{M}}$ the binary mesh matrix representation of \mathcal{M} obtained by setting all nonzero entries of \mathbf{M} to 1. This binary representation, is sometimes referred in graph theory literature as the *face-vertex incidence* matrix. It should not be confused with the more common *edge-vertex incidence* which is described in most introductory graph theory and exterior calculus books. An alternative definition of \mathcal{M} would be to set up its nonzero values so that

they refer to either their direct predecessor or successor in the cell. In view of action maps, our current formal definition is preferable.

4. Storage requirements

Arguably, the most widely used matrix formats are the coordinates format (COO), Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC), see e.g. [DER87]. In coordinates format, a sparse matrix is represented by the triplet $(\mathbf{r}, \mathbf{c}, \mathbf{v})$ which refers to keys for rows, columns, and corresponding values. In practice, CRS/CSC are more adapted for numerical computations. In this paper, we will use the CSC format for explaining algorithmic details. In this format, a matrix is defined by the triplet $\{\text{colptr}, \text{rowind}, \text{values}\}$, where *rowind*, and *values* are the same as *r* and *v*, whereas *colptr* is a compressed form of *c* which marks only the start of columns. Please note that in theory, CRS and CSC are simply transposes of each other.

To illustrate the concept behind our mesh matrix storage reduction, we start from the CSC format of the example in in Figure 1 which translates into the following matrix:

$$\begin{aligned} \text{values} &= [1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 4 \ 1 \ 2 \ 3 \ 1 \ 3 \ 2 \ \dots] \\ \text{rowind} &= [1 \ 2 \ 11 \ 13 \ 2 \ 3 \ 6 \ 9 \ 10 \ 11 \ 3 \ 4 \ 5 \ 6 \ 5 \ 6 \ 7 \ \dots] \\ &\quad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \\ \text{colptr} &= [1 \qquad \qquad 5 \qquad \qquad 11 \qquad \qquad 15 \qquad \dots] \end{aligned}$$

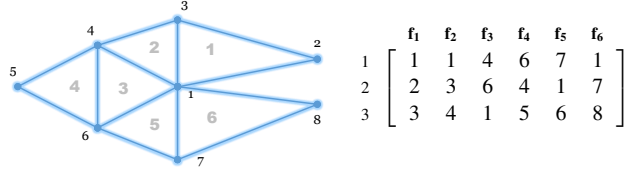
For the sake of argument, assume that the nonzero values are stored in double precision ($s_{\text{double}} = 8\text{byte}$) and the indices as integers ($s_{\text{int}} = 4\text{byte}$). The storage requirement for a general matrix within this format amounts to $Nz \cdot (s_{\text{double}} + s_{\text{int}}) + n_{\text{col}} \cdot s_{\text{int}}$, whereas Nz corresponds to the number of nonzeros and n_{col} is the width of the matrix. For a triangle mesh, the number of nonzeros in the mesh matrix is $3n_f$ and the number of columns is n_f . As the entries in the matrix correspond to indices storing them as integer is preferable to double. The storage cost then is $3 \cdot n_f \cdot (s_{\text{int}} + s_{\text{int}}) + n_f \cdot s_{\text{int}} = 7 \cdot n_f \cdot s_{\text{int}}$, which is 28 bytes per face.

The CSC representation is not unique, in fact, 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. Consider the previous example again. For the first and second column, the information contained in *values* is redundant since it coincides with traversal order when stepping through the column ($[1, 2, 3, 4]$ and $[1, 2, 3, 4, 5, 6]$). However, by changing the order of indices in the third column according to their entries in *values* (as shown below in bold), the entries in *values* become also redundant.

$$\begin{aligned} \text{values} &= [1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ \mathbf{1} \ \mathbf{2} \ \mathbf{3} \ 4 \ 1 \ 2 \ 3 \ \dots] \\ \text{rowind} &= [1 \ 2 \ 11 \ 13 \ 2 \ 3 \ 6 \ 9 \ 10 \ 11 \ \mathbf{4} \ \mathbf{5} \ \mathbf{6} \ 3 \ \mathbf{5} \ \mathbf{7} \ 6 \ \dots] \\ &\quad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \qquad \qquad \uparrow \\ \text{colptr} &= [1 \qquad \qquad 5 \qquad \qquad 11 \qquad \qquad 15 \qquad \dots] \end{aligned}$$

After performing this step for all columns, the role of the *values* becomes obsolete. In fact, the number of entries per column is known from the *colptr* and we need simply to traverse the entries in their order of appearance. The reordering reduces the storage cost for triangle meshes to $4 \cdot n_t \cdot s_{\text{int}}$, which corresponds to 16 bytes per triangle. Thus, in practice, we do not need to store the mesh matrix but only its sparsity pattern, namely *rowind* and *colptr*.

Storage requirements can be further reduced when all faces are of the same type. Consider the simple triangle mesh depicted below.



After re-ordering the indices in *rowind* according the entries in *values* as shown below it becomes clear that also the *colptr* becomes redundant since its entries can be inferred.

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

It ensues that we only need to store the *rowind*, and thus only 12 bytes per triangle. Hence, the storage cost for meshes where all faces are of the same type is the same as for the face table. In case of a general mesh, we require an additional index (4 bytes) per face. However, when using the face table, the number of vertices per face also needs to be stored. Thus, in practice we achieve equal memory requirements as the face table representation. The resulting memory storage reduction can make a substantial difference especially in concurrent infrastructures with limited memory resources, as for instance, the GPU. The advantage over the face table is that we gain the structure of a special sparse matrix on which we can perform linear algebra operations.

5. Linear algebra primitives

In order to take full advantage of the proposed matrix representation we endow it with suitable linear algebra primitives, which allow performing numerical computations on meshes as well as topological modifications (section 8). We pay special attention to avoid intermediate data creation which can severely hamper performance as it generates additional memory access. We capitalize on reusing the sparsity pattern of the mesh matrix in sparse matrix vector operations (*SpMV*) and sparse matrix-matrix multiplication (*SpMM*) and show how it can be applied in practice.

5.1. Action maps on vectors

For *SpMV* we define an action map as a function Q which acts on the nonzero entries of the mesh matrix. Its action can be i) a compact stencil which encodes interaction between face summits, ii) a vector of the same length as the nonzero values of the matrix which associates values with summits, e.g. angles, iii) a combination of both. Without loss of generality, we will illustrate the construction of action maps and their typical applications through a set of simple but expository examples. Assume one wants to compute the barycenter of each triangle using the mesh matrix. A straightforward way to do so, would be to write the barycenters as $\frac{1}{3} \mathbf{M}^T \mathbf{P}$, with \mathbf{P} being the array of vertex positions (the three dense vectors that correspond to the *x*, *y*, and *z* coordinates of the vertices). The problem with such a formulation is that it requires the creation of

new data (the sparse matrix $\bar{\mathbf{M}}$) and the computation of the transpose. Instead, we observe that the sparsity pattern of this intermediate matrix is similar to \mathbf{M} . We propose to use a mapping which acts on the non-zeros values of the matrix during the computation of the product, by replacing the values of the nonzero elements of the matrix \mathbf{M} by those stored in a given vector or obtained by a predefined scheme. This is what we call a mesh action map. The advantage of this formalism is that the intermediate matrix $\bar{\mathbf{M}}$ is actually never constructed explicitly but its values are only inferred from those of \mathbf{M} during multiplication. More precisely, given a sparse matrix with nonzero values V_{old} , we define a mapping Q which acts on the entries of V_{old} during multiplication as follows $\mathbf{M} \xrightarrow{Q:V_{old} \rightarrow V_{new}}$.

Since a transpose matrix times vector can be handled algorithmically as will be shown shortly in **Algorithm 1**, it is not necessary to create or store $\bar{\mathbf{M}}$ or $\bar{\mathbf{M}}^T$.

Revisiting the previous example, barycenters can be obtained by means of the action map which takes $(1, 2, 3)$ to $(1, 1, 1)$ as follows

$$B = \frac{1}{3} \mathbf{M}_{(1,2,3) \rightarrow (1,1,1)}^T \mathbf{P} = \frac{1}{3} \bar{\mathbf{M}}^T \mathbf{P}.$$

Algorithmically, this amounts to modifying the sparse matrix vector multiplication to account for the action map as outlined for the CSC matrix format in **Algorithm 1**.

Algorithm 1 Action mapped sparse matrix vector multiplication

```

1: procedure MAPPED-SPMV
2: input: Matrix in CSC format (rowind, colptr, values), vector x,
   actionMap Q
3: if  $\mathbf{M}\mathbf{v}$ 
4: for  $j = 0$  to  $n_c - 1$ 
5:   for  $k = \text{colptr}[j]$  to  $\text{colptr}[j + 1] - 1$ 
6:      $y[\text{rowind}[k]] += Q(\text{values}[k]) * x[j]$ 
7: if  $\mathbf{M}^T \mathbf{v}$ 
8: for  $j = 0$  to  $n_c - 1$ 
9:   for  $k = \text{colptr}[j]$  to  $\text{colptr}[j + 1] - 1$ 
10:     $y[j] += Q(\text{values}[k]) * x[\text{rowind}[k]]$ 
11: end procedure

```

The algorithm covers the action mapped multiplication of a matrix or its transpose by a vector. The multiplication by an array instead of a vector follows naturally by means of an extra loop.

To further illustrate the use of action maps, consider the case of computing vertex normals of a mesh, whereas the contribution of each face is weighted by its area. The arrays $\mathbf{P}_1\mathbf{P}_2$ and $\mathbf{P}_1\mathbf{P}_3$ over all triangles can be obtained by applying the operations

$$\mathbf{P}_1\mathbf{P}_2 = \mathbf{M}_{(1,2,3) \rightarrow (-1,1,0)}^T \mathbf{P}, \text{ and } \mathbf{P}_1\mathbf{P}_3 = \mathbf{M}_{(1,2,3) \rightarrow (-1,0,1)}^T \mathbf{P}. \quad (2)$$

The un-normalized vertex normals can be obtained by

$$\mathbf{N}_v = \mathbf{M}_{(1,2,3) \rightarrow (1,1,1)} \mathbf{N}_f = \bar{\mathbf{M}} \mathbf{N}_f; \quad (3)$$

where, the array \mathbf{N}_f holds face normals obtained by a row-wise vector product of $\mathbf{P}_1\mathbf{P}_2$ and $\mathbf{P}_1\mathbf{P}_3$.

If explicit weights should be associated with faces which are stored in a vector w of length n_f , action maps can be used to apply those weights. Weighted averaging can then be carried out

using the actions map Q which acts on face indices as follows $Q : (1, 2, 3)_{\mathbf{f}_i} \rightarrow (w_i, w_i, w_i)$ for all i . In this scenario, the use of action maps can be likened to a sparse matrix vector multiplication where the vector holding the nonzero values is replaced by the weights vector. A numerical example using the weighting scheme in [Max99] is substantiated in the results section 9.

Using our technique and implementation, a user can write the above example in about five lines of code. The only data structure that is used is the mesh matrix representation \mathbf{M} and none of the intermediate matrices are explicitly created or stored. Furthermore, with our parallel GPU implementation, high performance is guaranteed. If the above example was implemented without action maps but traditional matrix algebra, the intermediate matrices would need to be generated manually by means of iterating over the entries and using conditionals to write the required values to memory. This would not only require additional storage, but also increase memory transfers and branching operations which are especially costly within a GPU implementation. If a traditional mesh data structure was used, a user would usually iterate over the entire mesh, explicitly compute the normal for each face and add its contribution to all face vertices. In case of this simple example this might be a reasonable approach as long as one is not thriving for a parallel implementation. As faces contribute to multiple vertices, threads would have to synchronize while adding the contributions of their face normal, which not only complicates the code but also hurts performance.

To underline the advantage of using action maps, let us re-examine alternative options in details. For a standard CPU implementation using the face table, the steps can be summarized as follows: compute face normals, set vertex normals to zero, run over faces and add face normal to all involved vertex normals, normalize the resulting vectors. When attempting to perform the same steps on the GPU, one is faced with two options: i) start one thread for each face and use atomic adds to sum up the contributions (scatter), then normalize. ii) start one thread for each vertex normal, run over all faces surrounding this vertex and sum up face normals (gather), then normalize. In general, the scatter option (i) is slow due to access conflicts/atomics. The gather option (ii) is more favorable but requires an additional data structure for accessing faces around a given vertex. Clearly, this can be done without action maps as shown in equation 3 by explicitly creating the matrix $\bar{\mathbf{M}}$. With action maps, we avoid such intermediate data thus saving on storage (size of M) and memory transfer (two times size of M).

5.2. Action maps on generalized sparse matrix-matrix product

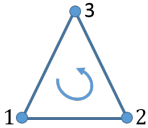
The concept of action maps can even be taken further and applied to sparse matrix-matrix multiplication. To fix the ideas, let us examine the matrix $\mathbf{S}_v = \bar{\mathbf{M}}\bar{\mathbf{M}}^T$ in the light of the example from Figure 1. The resulting matrix is shown below. The nonzero entries of \mathbf{S}_v represent the number of faces common to any given two vertices. The diagonal entries count the number of faces common to a given vertex. Please note, that for general polygonal meshes, two vertices will flag a 1 even if they are not directly connected by an edge, they need only to belong to the same face. In the particular case of triangle meshes, when two vertices share a face, they are connected by an edge. We can regard the matrix \mathbf{S}_v as a generalized vertex-

vertex adjacency. A similar face-face adjacency matrix is detailed in Appendix A.

$$\overline{\mathbf{M}}\overline{\mathbf{M}}^T = \begin{matrix} & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 & c_{10} & c_{11} & c_{12} & c_{13} & c_{14} & c_{15} \\ \begin{matrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ r_9 \\ r_{10} \\ r_{11} \\ r_{12} \\ r_{13} \\ r_{14} \\ r_{15} \end{matrix} & \left[\begin{array}{cccccccccccccccc} 2 & 2 & 1 & 1 & & & & & & & & 1 & & 1 & 1 & \\ 2 & 3 & 2 & 1 & 1 & & & & & & 1 & 1 & 2 & & 1 & 1 \\ 1 & 2 & 3 & 2 & 1 & 2 & & & & 1 & 1 & 1 & & & & 1 \\ 1 & 1 & 2 & 2 & 1 & 1 & & & & & & & & & & 1 \\ & & 1 & 1 & 2 & 2 & 1 & & & & & & & & & \\ & 1 & 2 & 1 & 2 & 4 & 2 & 1 & 2 & 1 & 1 & & & & & \\ & & & 1 & 2 & 2 & 1 & 1 & & & & & & & & \\ & & & & 1 & 1 & 2 & 2 & 1 & & & & & & & 1 \\ & 1 & 1 & & 2 & 1 & 2 & 3 & 2 & 1 & 1 & & & & & 1 \\ & 1 & 1 & & 1 & 1 & 2 & 3 & 2 & 2 & 1 & & & & & 1 \\ 1 & 2 & 1 & & 1 & & 1 & 2 & 3 & 1 & 2 & & & & & \\ & & & & & & 1 & 1 & 2 & 1 & 2 & 1 & & & & 1 \\ 1 & 1 & & & & & & & 1 & 2 & 1 & 2 & & & & \\ 1 & 1 & 1 & 1 & & & & & & & & & & & 1 & \\ & & & & & & & & 1 & 1 & 1 & & 1 & & & 1 \end{array} \right] \end{matrix}$$

Our objective is to facilitate the assembly of commonly used matrices, as, e.g., the one above. Assume a user requires to construct the adjacency matrix of an oriented mesh by looking at it as a directed graph. Starting with the matrix $\overline{\mathbf{M}}\overline{\mathbf{M}}^T$ one can see that it is clearly symmetric and does not reflect the orientation of edges. In order to capture this information, so as to obtain the adjacency matrix, we propose to alter the matrix multiplication by means of suitable action maps.

Consider the case of a triangle mesh. The sparsity pattern of the matrix product $\overline{\mathbf{M}}\overline{\mathbf{M}}^T$ results from the collisions of nonzero entries of $\overline{\mathbf{M}}$ and its transpose. The outcome of these collisions can be encoded by 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 to encode triangle orientation

$$Q_3 = \begin{matrix} & 1 & 2 & 3 \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{array} \right]; \quad (4)$$


When performing an action mapped matrix-matrix product, the entries of the first mesh matrix are used for indexing along the first dimension of the action map, the second matrix indexes along the second dimension, e.g., the collision of a 1 in the first matrix and a 3 in the second matrix corresponds to the third element in the first row of the action map. A pseudo code illustrating our idea is given in **Algorithm 2**; note that the implementation of SpMM itself is much more involved and requires additional steps such as memory allocation, see, e.g., [Dav06, Dav] for an ample description. Since Q_3 and its powers $Q_3^0 = \mathbf{I}_{3 \times 3}$ and Q_3^2 form a basis for all 3×3 circulant matrices, we can encode a variety of interactions within a face by means of simple action maps. For instance, for a triangular mesh, the uniform Laplacian (graph Laplacian) $\mathbf{D} - (\mathbf{A} \vee \mathbf{A}^T)$, where \vee refers to the logical OR, and \mathbf{D} to the diagonal degree matrix, can be obtained through the action map $Q = Q_3^0 - (Q_3^2 + Q_3)$. If a mesh has a boundary, the boundary adjacency can be captured by $Q_3 - Q_3^2$, traversal of positive entries will yield counterclockwise oriented boundary loops, negative ones yield clockwise traversal.

More generally, given a mesh where all faces are consistently counterclockwise oriented (n-gones) of the same kind. We define the action map Q_n associated with the matrix product of mesh ma-

Algorithm 2 Action mapped sparse matrix-matrix multiplication

```

1: procedure MAPPED-SPMM C=AB
2: input: Matrices A, B, and action map Q
3: for j ← 1 to n
4:   for k where B(k, j) ≠ 0
5:     for i where A(i, k) ≠ 0
6:       C(i, j) ← C(i, j) + Q(A(i, k), B(k, j))
7: end procedure

```

trix M and its transpose, as the cyclic permutation

$$Q_n = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ 0 & 0 & \dots & 0 & 1 \\ 1 & 0 & \dots & 0 & 0 \end{pmatrix} \quad (5)$$

All circulant interactions within a face are captured by linear combinations of Q_n and its powers. It is interesting to note that when the global mesh orientation is changed, the corresponding adjacency (transposed adjacency) can be obtained by the transpose of Q which corresponds to $Q^{(n-1)}$.

Consider the case of a quad mesh (such meshes can result from Catmull-Clark subdivision [CC78]). If we wish to capture relations only between diagonally opposed vertices within a faces, we can simply augment the matrix multiplication by the action map Q_4^2 . This would flag ones for diagonally opposed vertices. For a mesh with arbitrary oriented faces, the results above hold by operating on the sub-matrix meshes corresponding to set of faces of similar nature and then summing the results up.

We can already see that the combination of action maps and matrix-matrix multiplication offers an alternative way to build some well known matrices without having to go through a conventional sparse matrix construction. We will see how this extends to the more general assembly in finite elements.

6. Fast sparse matrix assembly

In most finite element formulations, a crucial but necessary step is the assembly of the system matrix, which amount to stiffness, mass, and/or a linear combination of both. Large sparse matrices are formed by adding several smaller contributions in a random order before the final nonzero values are known. This procedure is laborious and can be a bottleneck for instance in nonlinear or iterative setups where these matrices need to be updated.

An elemental contribution is a small matrix whose size is defined by the degrees of freedom of the problem (e.g., integration type). For instance, in the case of the constant strain triangle (CST) commonly known in graphics as the cotangent Laplacian, this will be a 3×3 matrix per triangle face (which corresponds to the cotangent Laplacian discretization). By setting action maps in terms of elemental contributions, the global matrix can be similarly obtained as above. For the sake of simplicity, consider the case of the CST.

The corresponding elemental contribution is given by

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

It can be seen that the pattern of this matrix already resembles that of $Q = Q_3^0 - (Q_3^2 + Q_3)$ from the previous sections. In order to use these entries along with the action map Q as in **Algorithm 2**, we recall that the vector holding the nonzeros entries of the mesh matrix M and the vector holding the cotangents of face angles are ordered in the same way. Therefore, upon multiplication the entry from the action map can be simply multiplied by the corresponding cotangent value using the index obtained from the mesh matrix.

For tetrahedral elements, the linear tetrahedron is given by:

$$\mathbf{k}_t = \frac{1}{6} \begin{pmatrix} \Sigma_1 & -l_{34} \cot \theta_{34} & -l_{24} \cot \theta_{24} & -l_{23} \cot \theta_{23} \\ & \Sigma_2 & -l_{14} \cot \theta_{14} & -l_{13} \cot \theta_{13} \\ & \text{sym.} & \Sigma_3 & -l_{12} \cot \theta_{12} \\ & & & \Sigma_4 \end{pmatrix};$$

where Σ_i infers the sum of non-diagonal entries in row i multiplied by -1 .

We can regard the tetrahedron as a combination of doubly oriented edges or oriented faces and we can associate the following map $Q_{tet} = Q_4^4 - (Q_4^3 + Q_4^2 + Q_4)$, i.e.,

$$Q_{tet} = \begin{pmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{pmatrix}. \quad (6)$$

The use of action maps in this case requires storing only 6 entries per tetrahedron contribution instead of 16.

Throughout the use of action maps the assembly process is streamlined by avoiding conditional statements, which is beneficial for concurrent programming as branching is avoided. Furthermore, the reduced memory requirements, makes the approach suitable for platforms with limited memory resources such as graphics hardware. Our construction extends to arbitrary finite elements such as shells, higher order tetrahedral and hexahedral elements (More degrees of freedom and integration points). Given the scope and nature of these scenarios, a detailed description and analysis will be given elsewhere.

7. Parallel GPU implementation

While usability and expressiveness is a core concept in our approach, the potential for an efficient implementation is equally important. Sparse matrix operations can be parallelized and have been brought to the GPU before. While we could use a standard sparse matrix library, like cuSparse [Dem12], CUSP [BDO12], or bhSparse [LV15] for most matrix algebra operations, an integration of our compressed format and action mapped multiplications forms a technical issue, as these libraries are either closed source (cuSparse), or optimized for a different format (CUSP uses COO), or only support non-transposed matrix vector operations (bhSparse).

Additionally, these libraries are built for general matrices and therefore do not take advantage of the special structure of the mesh matrix. Thus, we have implemented our own library for simple matrix algebra operations around mesh matrices and action maps in CUDA. For non-action mapped and general matrix operations we use cuSparse, which can be directly called with pointers to low level data structures and thus allows for data sharing with our implementation.

As representation for the mesh matrix, we directly use the format as described in Section 3, omitting the values and reordering *rowind* according to the vertex indices. Note that these reordering precludes the use of the *rowind* array with cuSparse, CUSP, and bhSparse as they expect the data to be ordered. For meshes with faces of the same type, we also omit the *colptr*. For best performance, we provide specialized kernels of all operations for triangle and quad meshes, allowing for a full optimization of the involved operations, like, e.g. loop unrolling. Specialized kernels for meshes with larger polygon counts can also be constructed easily, however, we also provide a generalized form of each operation which takes the number of vertices per face as input argument.

Mapped Matrix Vector The mapped matrix vector multiplication can be split into two cases: $\mathbf{M}\mathbf{v}$ and $\mathbf{M}^\top\mathbf{v}$. In both cases we follow the implementation shown in **Algorithm 1**, and parallelize over the matrix columns. This results in a parallelization over the output for $\mathbf{M}^\top\mathbf{v}$ and a parallelization over the elements of \mathbf{v} for $\mathbf{M}\mathbf{v}$.

In case of $\mathbf{M}^\top\mathbf{v}$, every thread is working on a separate output element and no inter thread communication is required. Furthermore, we can keep the temporary output result in a local register and only write the result after completing the loop. Furthermore, the number of elements processed per threads corresponds to the vertices of the face. If the mesh has a similar number of vertices across all faces, all threads will perform a similar number of operations and thus load balancing will implicitly be good. The entries of the mesh matrix are accessed once. The *colptr* is read once at the head of the for loop (line 9) and each entry in *rowid* is read throughout the loop body (line 10). However, adjacent values are read by the same thread across iterations of the loop. Thus, performance can be increased when caching these values in level-1 cache. While we could use a texture to perform this caching, we use the low-level *ldg* instruction provided in CUDA as there is no need for texture interpolation or data conversion and the same cache will be used. The vector \mathbf{v} and the map itself will potentially be accessed multiple times by different threads. Thus, we also enforce caching of this data in level-1 cache. Note that the implementation is rather simple, but due to the homogeneous load and efficient use of level-1 cache, we are competitive to other state-of-the-art, GPU sparse matrix algorithms.

In case of $\mathbf{M}\mathbf{v}$, every thread works on a different element of \mathbf{v} , but there is an overlap between the threads when writing the output. This case is usually about $10\times$ slower than the first case when using state-of-the-art GPU sparse matrix implementations, like cuSparse. Current practice suggests that in the general case, the detour of parallelization over the outputs and searching through the compressed column data to find those entries in the matrix that add to the thread's output element achieves the best performance. However, we follow another strategy and stick to **Algorithm 1**. To perform

the inter thread communication, we use the build-in atomic-add operations for float data, and a try-set-loop using atomic-compare-and-swap instructions for double. We again make sure all input data is cached in level-1. According to our experiments this strategy achieves superior results for mesh matrix multiplications. We attribute this fact to the relatively low number of collisions, which are bounded by the valence of the respective vertices. Additionally, as the result of the atomic-add instruction is not used by the kernel, the instruction compiles down to an atomic reduction in machine code, which is efficient on current GPU hardware. A more detailed description and performance analysis of our approach on standard numerical computing benchmarks can be consulted in [SDZS16].

Mapped Matrix-Matrix A mapped matrix-matrix multiplication is more complicated than the matrix vector product. The biggest challenges are (i) that the structure of the resulting matrix depends on the input matrices, (ii) that the organization of the entries in the resulting matrix requires communication between threads, and (iii) that the number of operations carried out by individual threads may vary strongly. To provide an efficient implementation we take advantage of the algorithmic description of bhSparse [LV15]. We tackle the aforementioned issues in a four stage approach: In the first stage, we compute an upper bound for the number for nonzeros in each column of the result matrix, which allows for allocating sufficient storage. To compute this estimate, we use one thread for each column of the first matrix, which iterates over the columns of the second matrix. If there is a match between the column index of the first matrix and the row index of the second matrix, an entry in the resulting matrix will exist and we increase the memory estimate. The second stage sorts the output matrix columns into bins, based on the expected number of entries they will contain, which allows choosing the best fitting strategy for the expected workload. The third stage applies the heap method [GPJS99] for columns with few entries, the ESC method [BDO12] for medium sized columns, and the merge method [GHS*15] for columns with many entries. The first two methods can be carried out using efficient on-chip shared memory, while the latter requires multiple kernel launches and memory allocations. The final stage rearranges the results of the previous stag 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. Again, we make sure that the action map entries are loaded via the level-1 cache. If a mapped matrix-matrix multiplication is carried out multiple times, like, e.g., the assembly during each iteration of a solver, we can reuse information from previous iterations. Specifically, we can perform binning only once and reuse the bins. In case the mesh changes only slightly from one iteration to the next, we only need to re-bin for those columns that were affected by the mesh operation.

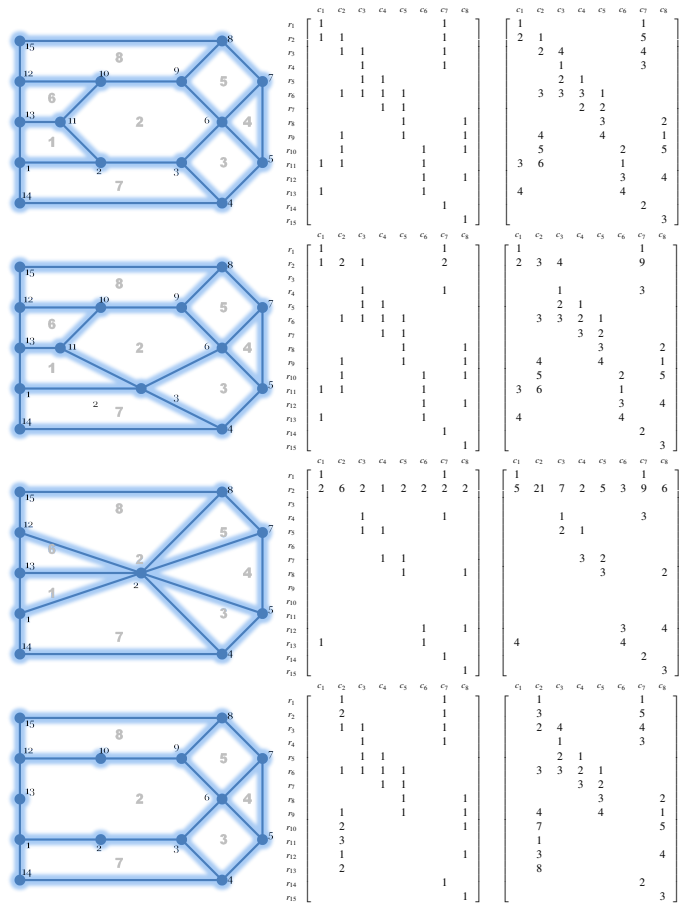
8. Topological operations

Many mesh processing applications require introducing modifications on the connectivity as for instance in mesh decimation and simplification, e.g. [GH97]. These applications proceed by performing a series of local operations such as edge collapse and face decimation. We will show how such operations can be performed on the mesh matrix representation by means of matrix multiplication. At this stage, we are not concerned with the geometric validity of the merging from a modeling viewpoint, but we demonstrate the algebraic operation capable of performing such task.

Edge collapse removes an edge from a mesh merging its end vertices. Given a mesh defined by its mesh matrix \mathbf{M} . Its topology after collapsing an edge i, j (or merging vertices i and j) is captured by the matrix $\bar{\mathbf{N}} = \mathbf{K}\bar{\mathbf{M}}$. Where \mathbf{K} is the sparse identity matrix \mathbf{I}_{n_v} , with $K(j, j) = 0$ and $K(i, j) = 1$;

Consider the example in Table 1-top: Assume a user would like to merge vertices 2 and 3, let K be the matrix defined as identity with $K(3,3) = 0$ and $K(2,3) = 1$. $K\bar{\mathbf{M}}$ is shown in the second row,

Table 1: Illustration of mesh simplification operations on an input mesh (top-left). Their effect on the mesh matrix and its binary form are shown. From second row to bottom, edge collapse, face collapse, vertex removal, see text for details.



middle in Table 1; Notice that the third row has disappeared. The faces that are affected by the merging are f_2 , f_3 and f_7 and their new summits are given by the nonzero elements of the matrix. Furthermore, when dealing with the ordered matrix, only the merged vertices order is affected. The right ordering can be obtained by simple shifts.

Face collapse amounts to the merging of all face vertices into a single vertex and it can be achieved similarly. Given a mesh defined by its mesh matrix \mathbf{M} . Its topology after collapsing a face \mathbf{f}_k into one of its vertices i is captured by the matrix $\bar{\mathbf{N}} = \mathbf{K}\mathbf{M}$. Where \mathbf{K} is sparse identity matrix \mathbf{I}_{n_v} with $K(j, j) = 0$ and $K(i, j) = 1$, for all $j \in \mathbf{f}_k, j \neq i$; This scenario is illustrated in the third row of Table 1 for the collapse of face 2 into vertex 2.

Vertex removal directly leads to face merging. The algebraic operations can be summarized in a similar manner, but this time the action is on the faces. The merging of faces (generally due to the removal of an edge or a vertex) can be formulated algebraically as: let $f_j, j = 1, \dots, n_f$ be the faces which will be merged into face f_i . let \mathbf{K} be a sparse matrix equal to identity of size $n_f \times n_f$. Let we set, $K(j, j) = 0$ and $K(i, j) = 1$; The topology of the new mesh is reflected in $\bar{\mathbf{N}} = \mathbf{M}\mathbf{K}^T$; This scenario is illustrated in the last row of Table 1 which features the merging of faces 1 and 6 into face 2.

Although matrix based mesh simplification is very efficient when performing a batch of operations simultaneously, its use in sequential algorithms, e.g. as proposed by Garland and Heckbert [GH97] is not recommended as it is technically expensive to perform matrix multiplication for the sake of a single simplification step. Instead, this can be done efficiently by first extracting the relevant faces (columns), performing a localized multiplication and then replacing the results back. Figure 2 illustrates the simplification of the mesh of a typical model driven by our matrix based simplification operations which reflects the original approach [GH97]. An animation of the evolution of the matrix and the mesh is shown in the accompanying media.

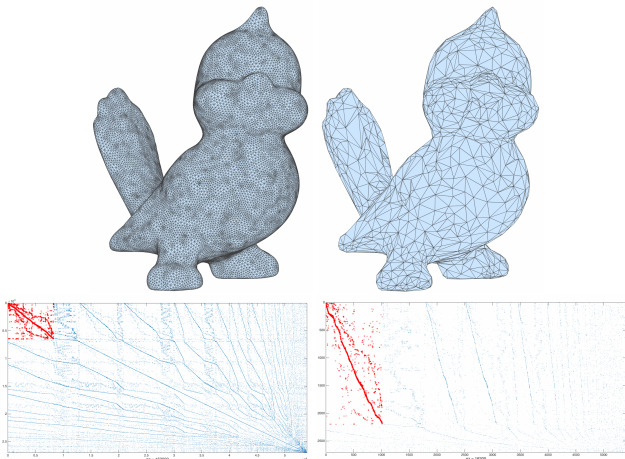


Figure 2: Decimation of the Tweety model (top-left). The overall matrix structure is preserved throughout simplification.

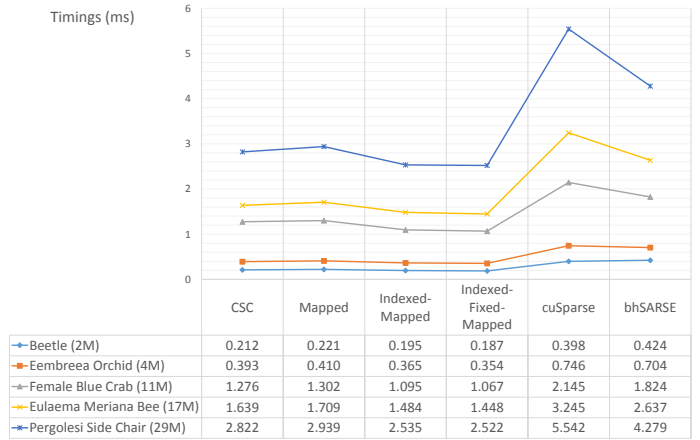
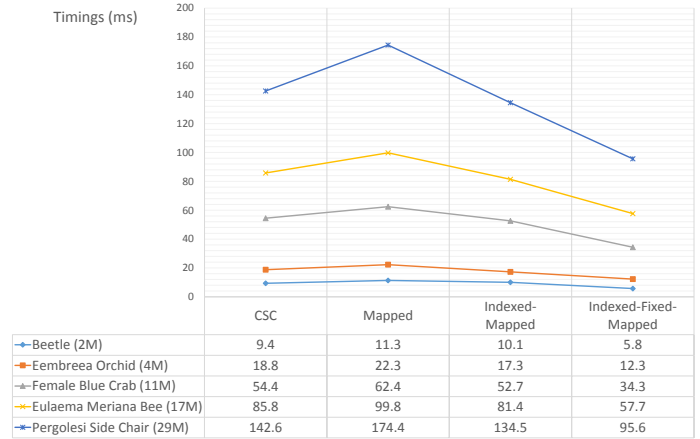


Figure 3: Impact of sparse matrix storage reduction and the use of action maps on matrix vector multiplication $\mathbf{M}^T \mathbf{v}$ on CPU (top) and on the GPU bottom (timings in milliseconds). Note that $\mathbf{M}^T \mathbf{v}$ implies summation on the compressed direction. For reference, we report the timings for cuSparse [NV15] and BhSparse [LV15].

9. Experimental results

Throughout our experiments, we used the following hardware configuration: an Intel Xeon E5-2637 v3 CPU running at 3.50GHz, 32GB of memory and an NVIDIA Geforce 980Ti with 2816 compute cores and 6GB of memory running at 1GHz.

Matrix-vector multiplication

To evaluate the performance of our sparse matrix-vector multiplication, we compare our reference CSC implementation to CSC combined with action maps in the storage formats described in sections 4 and 7, namely CSC, rowind+colptr and rowind. Note that CSC corresponds to a simple matrix multiplication where the values of the mesh matrix have already been replaced with the values otherwise inserted by the action map, i.e., we do not capture the overhead of manually changing the matrix in this case. CSC+maps corresponds to storing the mesh matrix explicitly in CSC format and using an action map to look up the value during multiplication. rowind+colptr+maps recovers the entries of the mesh matrix

from the rowind array and performs the lookup in the action map. rowind+maps is only applicable, if all faces of the mesh have an equal number of vertices and thus the colptr can be omitted. For our serial implementation, Figure 3-top shows the effect of these storage reduction techniques on the CPU for $M^T v$. The pattern is consistent across our testing data which comprises meshes with triangle counts ranging from 2M to 29M triangles.

For our experiments on graphics hardware, shown in Figure 3-bottom, we provide the performance of cuSparse [NVI15] and BhSparse [LV15] for reference, again assuming the mesh matrix has already been replaced with the values required for the operation. As expected, using action maps without storage optimizations slightly decreases performance in comparison to a plain matrix multiplication, as it adds an additional lookup. However, the overhead is usually below 20% on the CPU and below 5% on the GPU. Furthermore, the explicit generation of the required matrix would take considerably longer and additional memory would be required to store the matrix, before the slightly more efficient traditional matrix multiplication could be carried out. Applying the storage reductions and specialized implementations consistently improves performance. Surprisingly, our implementation (even in its plain form) outperformed the highly tuned cuSparse and BhSparse implementations. We can only attribute that to the fact that our implementation is not targeted for the general case where a high number of load imbalances occur, but rather to mesh matrices which show a more consistent structure. We conclude that our specialized algebra primitives are better suited for computations on meshes than general purpose primitives.

A key feature of our sparse matrix implementation is the steady performance when computing Mv on graphics hardware, i.e., when access conflicts complicate the computations. Typical results of our approach are compared to cuSparse [NVI15] in Table 2.

Table 2: Computation of $M^T v$ (no access conflicts between threads) and Mv (access conflicts need to be resolved) of our approach with action maps and a plain matrix vector multiplication in cuSparse. Timings in milliseconds.

	$M^T v$		Mv	
	cuSparse	Ours	cuSparse	Ours
Embreea Orchid (4M Δ)	0.74	0.35	8.97	0.27
Pergolesi side chair (29M Δ)	5.54	2.52	63.56	1.96

While a plain matrix multiplication in cuSparse is about two times slower than our mapped multiplication when no access conflicts occur, our implementation is about 30 \times faster in the case when access conflicts need to be resolved. It seems that our implementation using atomic operations is very well suited for mesh matrix operations. Please note that for this scenario, we do not report CPU times since the transpose is implicitly taken care of in the serial algorithmic formulation as noted earlier.

Matrix assembly: A well established serial method for matrix assembly is the *Sparse* function in Matlab [GMS92], which directly assembles the coordinate triplets into a sparse matrix. A variant known as *Sparse2* which capitalizes on a different sorting scheme has been proposed in the SuiteSparse package [Dav]. Most recently,

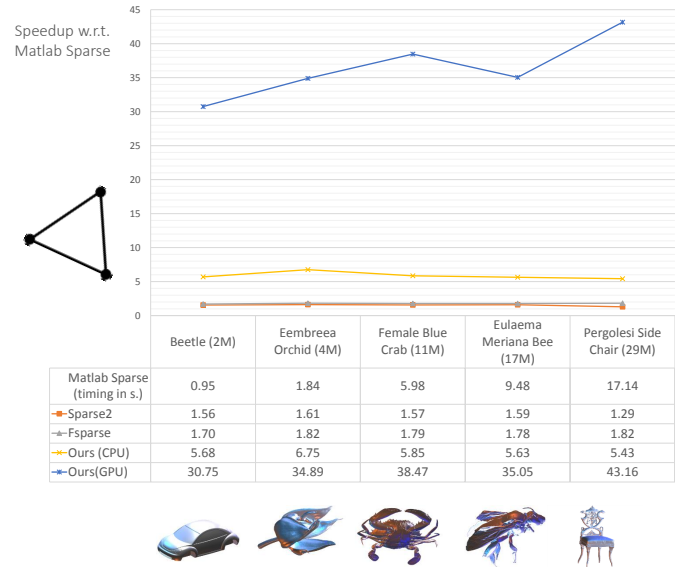


Figure 4: Assembly of linear triangle based stiffness (Laplacian). Timings for Matlab Sparse are reported in seconds. For Sparse2, Fsparse, our serial and GPU implementations we report the relative speedup to Matlab Sparse.

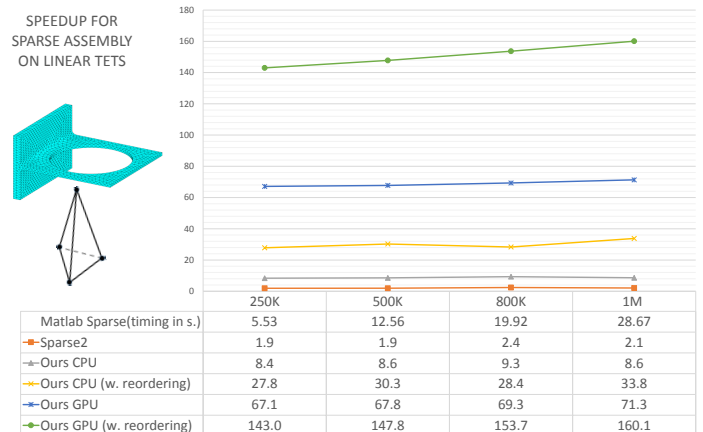


Figure 5: Assembly of the linear tetrahedron based stiffness (Laplacian). Timings for Matlab Sparse are reported in seconds. For Sparse2, our serial and GPU implementations we report the relative speedup to Matlab Sparse. Note the effect of reordering.

an assembly method which takes advantage of the multi-core structure of modern CPUs was reported [EL14] under the name *Fsparse*. We tested our approach for matrix assembly against these state of the art methods. Figure 4 shows the speedup of Sparse2, Fsparse and our approach against the Matlab sparse function. In this case, the stiffness based on the linear triangle is assembled over different meshes.

Figure 5 shows the speedup of Sparse2, our serial and parallel implementations with and without reordering (Fsparse is not reported as it crashed for large tet counts). In this scenario, the stiff-

ness matrix based on the linear tetrahedron is assembled over the mesh of a mechanical part with varying tetrahedron count. The data suggests that as connectivity patterns get more complex as for instance in the case of tetrahedral meshes, the gains achieved by our lean formulation of the problem become more pronounced reaching speedups of up to $150\times$ compared to the well established general purpose Matlab sparse function. The use of memory friendly layouts through re-ordering is highlighted by our array based approach. The results shown in figure 5 rely on the reverse Cuthill-McKee (RCM) ordering method [GL81], which attempts to reduce matrix bandwidth. It can be regarded as an iterative variant of the basic breadth first search (BFS).

Matrix assembly is central to a wide range of applications such as parametrization, smoothing, deformation, and simulation. The speedup obtained by our formulation and especially the simplicity by which our abstraction extends to the parallel setting has wide reaching effects and can help improve the overall performance across those applications.

Applications: To substantiate the flexibility and versatility of our approach, we tested on a set of basic but representative algorithms commonly used in practice. In all experiments, the timings reflect only the operation performed on the mesh. Loading and data structure creation (for methods that use half edge or other structures) is not included. For our GPU measurements, please note that all of all our algorithmic steps are performed on the GPU, so there is no intermediate transfers from main memory.

We implemented the normals estimation of [Max99] using our abstraction and we compare its performance to an existing library [Rus15]. Results are summarized in Table 3.

Table 3: Vertex normals computation using the weights of N. Max [Max99], timings in seconds.

	Vertex Normals		
	Trimesh2	ours(CPU)	ours(GPU)
Embreea Orchid ($4M\Delta$)	.61	.32	0.0078
Earhart Flight Suit ($21.5M\Delta$)	3.31	1.93	0.046
Pergolesi side chair ($29M\Delta$)	4.45	2.81	0.073

As a second example, we slice a mesh by a plane. This induces a change of the mesh topology as well as the creation of different types of faces. For a triangle mesh this yields triangles and quads along the regions being cut. The results of our serial and parallel implementations are summarized in Table 4.

Table 4: Mesh slicing by a plane, timings in seconds

	Mesh slicing	
	Ours (CPU)	Ours (GPU)
Embreea Orchid ($4M\Delta$)	0.17	0.011
Earhart Flight Suit ($21.5M\Delta$)	1.13	0.065
Pergolesi side chair ($29M\Delta$)	1.68	0.125

A more elaborate type of problems are subdivision schemes. The generalized Catmull-Clark subdivision [CC78] offers an interesting challenge for specialized triangle mesh based on classical data

Table 5: Catmull-Clark subdivision based on the mesh matrix, timings for 1 step in seconds.

	Catmull-Clark subdivision			
	Meshlab	OpenSubdiv	Ours CPU	Ours GPU
Embreea Orchid $4M\Delta$	8.24	6.77	2.93	0.25
Earhart Flight Suit $21.5M\Delta$	44.58	54.82	27.29	2.01
Pergolesi side chair $29M\Delta$	252.68	90.75	48.66	4.09

structures since the mesh nature changes to quads after the first round. Our performance results are summarized below and compared to the reference implementation in [CCC*08] and to the state of the art implementation of OpenSubdiv [Pix]. Please note that for OpenSubdiv, major steps are performed on the CPU, namely topology refinement and stencil table creation. Only the evaluation of stencils is performed on the GPU. We can readily observe that there is a noticeable deterioration in the performance of Meshlab on the Pergolesi side chair mesh. This reflects that the implementation is affected by swapping as it consumes more than the available 32GB of memory. In contrast, our lean mesh representation maintains a steady performance even on graphics hardware.

10. Discussion and conclusion

A data structure is not a goal per se, but rather a representation which facilitates performing desired numerical tasks on the underlying data. In this spirit, we described a mesh representation based on sparse matrices, and demonstrated how the ensuing storage requirements can be effectively reduced. Furthermore, we developed algebraic tools to allow our representation to blend seamlessly into the mesh processing pipeline. Special attention is paid to avoiding intermediate data creation (communication) which can severely hamper performance. Note that ideas like action maps can be easily implemented on top of existing dedicated linear algebra packages such as, e.g., Eigen [GJ*10] thanks to expression template. Within our formalism, algorithms that generally require considerable amounts of code and programming effort can be formulated in a clear and concise linear algebra syntax. This improves code readability and reduces code bloat bringing us closer to the spirit of the “ten digits, five seconds, and just one page” jingle [Tre05].

Acknowledgements

Mesh data sets are courtesy of the Smithsonian Institution. We thank Helge Rhodin for his feedback on the manuscript, and Nadia Robertini for her most encouraging comments on an early presentation of this material. This research was partially supported by the Max Planck Center for Visual Computing and Communication.

References

- [AMR88] ABRAHAM R., MARSDEN J. E., RATIU R.: *Manifolds, Tensor Analysis, and Applications*. Springer-Verlag New York, Inc., 1988. 1
- [ASW06] ASPNÄS M., SIGNELL A., WESTERHOLM J.: Efficient assembly of sparse matrices using hashing. In *PARA'06* (2006), Springer-Verlag, pp. 900–907. 3
- [Bau72] BAUMGART B. G.: *Winged edge polyhedron representation*. Tech. rep., Stanford, CA, USA, 1972. 1, 2

- [BDO12] BELL N., DALTON S., OLSON L. N.: Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152. 3, 7, 8
- [BG09] BELL N., GARLAND M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), ACM, pp. 1–11. 2, 3
- [Bos98] BOSSAVIT A.: *Computational electromagnetism*. Electromagnetism. Academic Press Inc., San Diego, CA, 1998. Variational formulations, complementarity, edge elements. 2
- [BSBK02] BOTSCH M., STEINBERG S., BISCHOFF S., KOBBELT L.: Openmesh – a generic and efficient polygon mesh data structure. In *OpenSG Symposium* (2002). 2
- [CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design* 10, 6 (1978), 350 – 355. 6, 11
- [CCC*08] CIGNONI P., CALLIERI M., CORSINI M., DELLEPIANE M., GANOVELLI F., RANZUGLIA G.: MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference* (2008). 11
- [CDG*08] CHANG F., DEAN J., GHEMAWAT S., HSIEH W. C., WALLACH D. A., BURROWS M., CHANDRA T., FIKES A., GRUBER R. E.: Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.* 26, 2 (June 2008), 4:1–4:26. 2
- [Cho97] CHOW M. M.: Optimized geometry compression for real-time rendering. In *Proceedings of the 8th Conference on Visualization '97* (1997), IEEE Computer Society Press, pp. 347–354. 2
- [CKS98] CAMPAGNA S., KOBBELT L., SEIDEL H.-P.: Directed edges—a scalable representation for triangle meshes. *J. Graph. Tools* 3, 4 (1998), 1–11. 2
- [Com16] COMSOL: *Multiphysics Reference Guide*, 1996–2016. 3
- [CRW05] CASTILLO P., RIEBEN R., WHITE D.: Femster: An object-oriented class library of high-order discrete differential forms. *ACM Trans. Math. Softw.* 31, 4 (Dec. 2005), 425–457. 2
- [Dav] DAVIS T.: SuiteSparse: A suite of sparse matrix packages. <http://www.cise.ufl.edu/davis/>. 3, 6, 10
- [Dav06] DAVIS T. A.: *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. 2, 3, 6
- [Dee95] DEERING M.: Geometry compression. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques* (1995), SIGGRAPH '95, ACM, pp. 13–20. 2
- [Dem12] DEMOUTH J.: Sparse matrix-matrix multiplication on the gpu. In *Proceedings of the GPU Technology Conference* (2012). 3, 7
- [DER87] DUFF I. S., ERISMAN A. M., REID J. K.: *Direct Methods for Sparse Matrices*. Monographs on Numerical Analysis. Oxford University Press, USA, 1987. 4
- [DKT06] DESBRUN M., KANSO E., TONG Y.: Discrete differential forms for computational modeling. In *ACM SIGGRAPH Courses* (2006), ACM, pp. 39–54. 2
- [DMPS07] DICARLO A., MILICCHIO F., PAOLUZZI A., SHAPIRO V.: Solid and physical modeling with chain complexes. In *SPM '07: Proceedings of the 2007 ACM symposium on Solid and physical modeling* (2007), ACM, pp. 73–84. 2
- [DMZ*16] DEVITO Z., MARA M., ZOLLHÖFER M., BERNSTEIN G. L., RAGAN-KELLEY J., THEOBALT C., HANRAHAN P., FISHER M., NIESSNER M.: Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *CoRR abs/1604.06525* (2016). 3
- [Edm60] EDMONDS J.: A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society* 7 (1960). 2
- [EL14] ENGBLOM S., LUKARSKI D.: Fast matlab compatible sparse assembly on multicore computers. *CoRR abs/1406.1066* (2014). 3, 10
- [GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *SIGGRAPH '97* (1997), pp. 209–216. 8, 9
- [GHS*15] GREMSE F., HOFTER A., SCHWEN L. O., KIESSLING F., NAUMANN U.: Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71. 3, 8
- [GJ*10] GUENNEBAUD G., JACOB B., ET AL.: Eigen v3. <http://eigen.tuxfamily.org>, 2010. 11
- [GL81] GEORGE A., LIU J. W.: *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall, 1981. 11
- [GLG*15] GUO X., LANGE M., GORMAN G., MITCHELL L., WEILAND M.: Developing a scalable hybrid mpi/openmp unstructured finite element model. *Computers & Fluids* 110 (2015), 227 – 234. 2
- [GMS92] GILBERT J. R., MOLER C., SCHREIBER R.: Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications* 13, 1 (1992), 333–356. 2, 3, 10
- [GPJS99] GILBERT J. R., PUGH JR W. W., SHPEISMAN T.: Ordered sparse accumulator and its use in efficient sparse matrix computation, Nov. 9 1999. US Patent 5,983,230. 8
- [GS85] GUIBAS L., STOLFI J.: Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.* 4, 2 (Apr. 1985), 74–123. 2
- [GY03] GU X., YAU S.-T.: Global conformal surface parameterization. In *SGP '03* (2003), pp. 127–137. 2
- [Har67] HARARY F.: Graphs and matrices. *SIAM Review* 9, 1 (1967), pp. 83–90. 2
- [HLSO12] HECHT F., LEE Y. J., SHEWCHUK J. R., O'BRIEN J. F.: Updated sparse cholesky factors for corotational elastodynamics. *ACM Trans. Graph.* 31, 5 (Sept. 2012), 123:1–123:13. 2
- [Hop99] HOPPE H.: Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99* (1999), pp. 269–276. 2
- [ip17] IN PREPARATION: Meshblas. *in preparation* (2017). 2
- [Jan] JANSSON N.: Optimizing sparse matrix assembly in finite element solvers with one-sided communication. In *High Performance Computing for Computational Science - VECPAR 2012*, Springer, pp. 128–139. 3
- [JDB*15] JEHL M., DEDNER A., BETCKE T., ARISTOVICH K., KLÖFKORN R., HOLDER D.: A fast parallel solver for the forward problem in electrical impedance tomography. *IEEE Transactions on Biomedical Engineering* 62, 1 (Jan 2015), 126–137. 2
- [JHN11] JANSSON N., HOFFMAN J., NAZAROV M.: Adaptive simulation of turbulent flow past a full car model. In *SC 2011* (Nov 2011), pp. 1–8. 2
- [Ket98] KETTNER L.: Designing a data structure for polyhedral surfaces. In *SCG '98* (1998), ACM, pp. 146–154. 2
- [Kir47] KIRCHHOFF G.: über die auflösung der gleichungen, auf welche man bei der untersuchungen der linearen verteilung galvanischer ströme geführt wird.. *Ann. Phys. Chem.* 72 (1847), 497–508. Translated by J. B. O'Toole in I.R.E. Trans. Circuit Theory, CT-5 (1958) 4. 2
- [Lie94] LIENHARDT P.: N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Int. J. Comput. Geometry Appl.* 4, 3 (1994), 275–324. 2
- [LV15] LIU W., VINTER B.: A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *J. Parallel Distrib. Comput.* 85, C (Nov. 2015), 47–61. 3, 7, 8, 9, 10
- [Män89] MÄNTYLÄ M.: *Advanced Topics in Solid Modeling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 49–74. 1, 2
- [Max99] MAX N.: Weights for computing vertex normals from facet normals. *J. Graph. Tools* 4, 2 (1999), 1–6. 5, 11
- [MBB*13] MATTSO T., BADER D., BERRY J., BULUC A., DONGARRA J., FALOUTSOS C., FEO J., GILBERT J., GONZALEZ J., HENDRICKSON B., KEPNER J., LEISERSON C., LUMSDAINE A., PADUA

D., POOLE S., REINHARDT S., STONEBRAKER M., WALLACH S., YOO A.: Standards for graph algorithm primitives. In *IEEE High Performance Extreme Computing Conference (HPEC)* (2013), pp. 1–2. 2

[MLDH15] MAGLO A., LAVOUÉ G., DUPONT F., HUDELLOT C.: 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Comput. Surv.* 47, 3 (Feb. 2015), 44:1–44:41. 2

[MTW73] MISNER C., THORNE K., WHEELER J.: *Gravitation*. W.H. Freeman and Company, 1973. 3

[NVI15] NVIDIA: *The API reference guide for cuSPARSE, the CUDA sparse matrix library.*, v7.5 ed. NVIDIA, September 2015. 2, 9, 10

[Pix] PIXAR: Opensubdiv. <http://graphics.pixar.com/opensubdiv>. 11

[PO09] PARKER E. G., O'BRIEN J. F.: Real-time deformation and fracture in a game environment. In *Proc. SCA '09* (2009), ACM, pp. 165–175. 2

[PT95] PDE TOOLBOX THE MATHWORKS I.: *MATLAB and Partial Differential Equation Toolbox*. Natick, Massachusetts, United States, 1995. 1, 3

[RG15] REGULY I. Z., GILES M. B.: Finite element algorithms and data structures on graphical processing units. *Int. J. Parallel Program.* 43, 2 (Apr. 2015), 203–239. 2

[Rus15] RUSINKIEWICZ S.: The trimesh2 library-version 2.12. <http://gfx.cs.princeton.edu/proj/trimesh2/>, 2015. 11

[SDZS16] STEINBERGER M., DERLER A., ZAYER R., SEIDEL H. P.: How naive is naive spmv on the gpu? In *IEEE High Performance Extreme Computing Conference (HPEC)* (2016), pp. 1–8. 8

[Tay70] TAYLOR R. L.: FEAP - finite element analysis program, version 8.4, 2013, 1970. 1

[TMDK15] TENG Y., MEYER M., DEROSE T., KIM T.: Subspace condensation: Full space adaptivity for subspace deformations. *ACM Trans. Graph.* 34, 4 (July 2015), 76:1–76:9. 3

[TPD15] THÉBAULT L., PETIT E., DINH Q.: Scalable and efficient implementation of 3d unstructured meshes computation: A case study on matrix assembly. *SIGPLAN Not.* 50, 8 (Jan. 2015), 120–129. 3

[Tre05] TREFETHEN L. N.: *TEN DIGIT ALGORITHMS*. Tech. rep., Oxford University, 2005. 11

[TWT*16] TANG M., WANG H., TANG L., TONG R., MANOCHA D.: CAMA: Contact-aware matrix assembly with unified collision handling for GPU-based cloth simulation. *Computer Graphics Forum (Proceedings of Eurographics)* 35, 2 (2016), 511–521. 3

[WBS*13] WEBER D., BENDER J., SCHNOES M., STORK A., FELLNER D.: Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications. *Computer Graphics Forum* 32, 1 (2013), 16–26. 3

[Wor81] WORLTON J.: The philosophy behind the machines. *Computer World* (Nov.9 1981). 2

[YT12] YOSHIZAWA H., TAKAHASHI D.: Automatic tuning of sparse matrix-vector multiplication for crs format on GPUs. In *IEEE 15th International Conference on Computational Science and Engineering (CSE)* (2012), pp. 130–136. 2

[Zay07] ZAYER R.: *Numerical and Variational Aspects of Mesh Parameterization and Editing*. Doctoral dissertation, Universität des Saarlandes, September 2007. 3

Appendix A: Mesh matrix relations

We can define a generalized face-face adjacency matrix as $\mathbf{S}_f = \overline{\mathbf{M}}^\top \overline{\mathbf{M}}$; the nonzero entries of \mathbf{S}_f represent the number of shared vertices between any two given faces. The matrix \mathbf{S}_f is interesting in many ways. Let us examine it in the light of our first example from section 3. The corresponding matrix is given below on the right side. We can see that the diagonal entries count the

number of vertices of a face, whereas the non-diagonal entries reveal the nature of adjacency between faces. When two faces share an edge, the corresponding value is 2, whereas it is 1 when only a single vertex is shared, as for instance, faces 2 and 4. This information reveals which faces lay on the mesh boundary. On a given row, when the number of off-diagonal entries flagging a 2 matches the diagonal entry, this means all edges are double and thus the face has no boundary edges. When they differ, the difference is the number of face edges on the boundary. The conventional adjacency matrix \mathbf{A}_d of the dual mesh is related to the matrix \mathbf{S}_f as $\mathbf{A}_d = \{\mathbf{S}_f == 2\}$; The matrix \mathbf{A}_d amounts to restricting the face-face connectivity to shared edges as explained above. Thus it yields exactly the dual of the original mesh. This is illustrated in the example in figure 6 and its corresponding dual matrix \mathbf{A}_d . A more elaborate example is shown in figure 7..

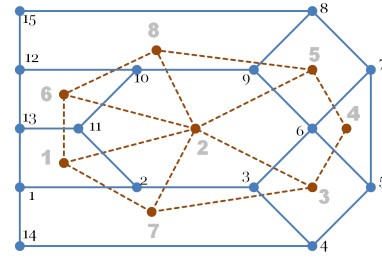


Figure 6: Example mesh and its dual.

$$\begin{array}{c}
 \begin{matrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ r_1 & \begin{bmatrix} 4 & 2 & & & & 2 & 2 & \\ 2 & 6 & 2 & 1 & 2 & 2 & 2 & 2 \\ 2 & 4 & 2 & 1 & & & 2 & \\ 1 & 2 & 3 & 2 & & & & \\ 2 & 1 & 2 & 4 & & & 2 & \\ 2 & 2 & & & 4 & & 2 & \\ 2 & 2 & 2 & & & & 5 & \\ 2 & & & & & & 2 & 2 & 5 \end{bmatrix} \\ \mathbf{S}_f \end{matrix} & , & \begin{matrix} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ r_1 & \begin{bmatrix} 1 & & & & & 1 & 1 & \\ 1 & 1 & & & & 1 & 1 & 1 & 1 \\ 1 & & 1 & & & & 1 & \\ & & 1 & & 1 & & & \\ 1 & & 1 & & & & & 1 \\ 1 & 1 & & & & & & 1 \\ 1 & 1 & 1 & & & & & \\ 1 & & & & 1 & 1 & & \end{bmatrix} \\ \mathbf{A}_d \end{matrix}
 \end{array}$$

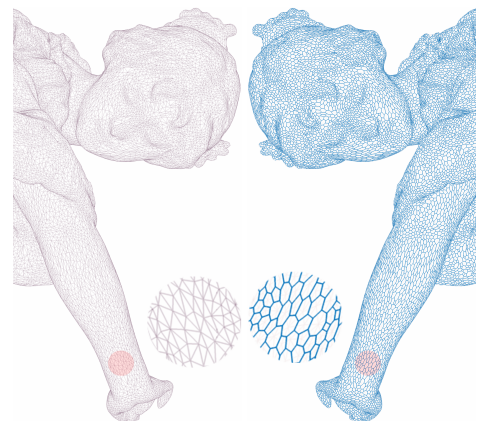


Figure 7: A mesh (right) and its dual (left) as obtained from \mathbf{A}_d . Here, figure of a dancer from 1910, courtesy of the Smithsonian.