DOI: 10.1111/cgf.13245

Eurographics Symposium on Geometry Processing 2017
J. A. Bærentzen and K. Hildebrandt
(Guest Editors)

Volume 36 (2017), Number 5

Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs

J. S. Mueller-Roemer[1], C. Altenhofen[1] and A. Stork[1]

[1]TU Darmstadt & Fraunhofer IGD, Germany

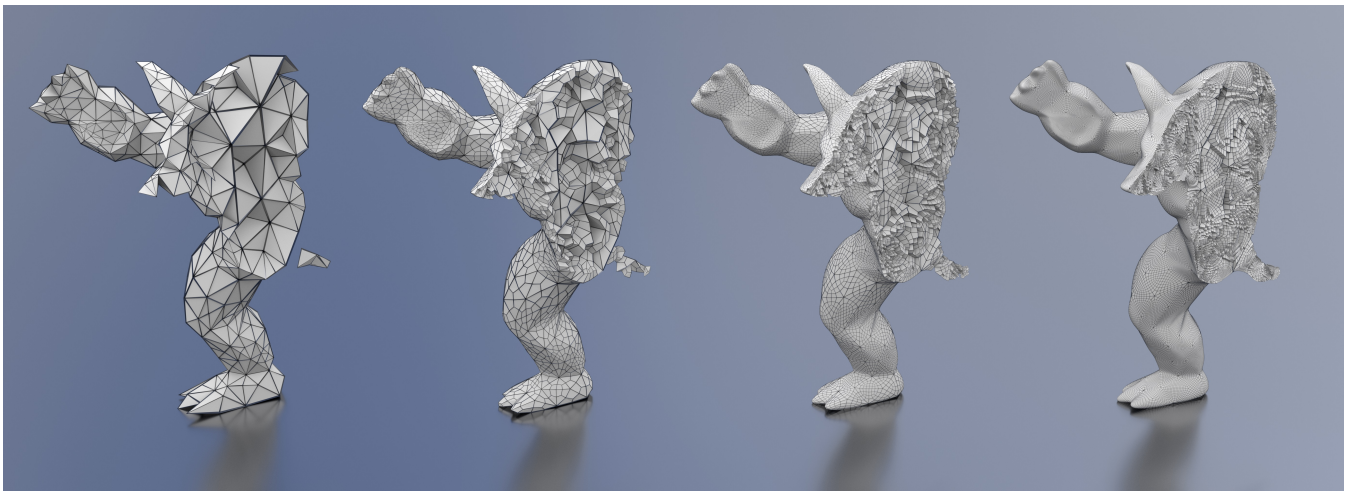


Figure 1: *Sliced view of the armadillo model at subdivision levels 0 to 3, from left to right, showing the inner structure of the mesh.*

Abstract

In this paper, we present a novel volumetric mesh representation suited for parallel computing on modern GPU architectures. The data structure is based on a compact, ternary sparse matrix storage of boundary operators. Boundary operators correspond to the first-order top-down relations of k-faces to their $(k-1)$-face facets. The compact, ternary matrix storage format is based on compressed sparse row matrices with signed indices and allows for efficient parallel computation of indirect and bottom-up relations. This representation is then used in the implementation of several parallel volumetric mesh algorithms including Laplacian smoothing and volumetric Catmull-Clark subdivision. We compare these algorithms with their counterparts based on OpenVolumeMesh and achieve speedups from $3\times$ to $531\times$, for sufficiently large meshes, while reducing memory consumption by up to 36%.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations

1. Introduction

Numerical simulation plays a large role both in the modern engineering design process and in computer animation. Many of the simulation algorithms used are based on discrete, volumetric meshes. While general purpose computing on graphics processing units (GPGPU) for simulations themselves is widespread, volumetric mesh processing mostly remains confined to serial processing on the system processor (CPU), especially when topological changes are involved. At the same time, changes to the input mesh frequently require recomputation of several matrices and other data

structures, such as spatial acceleration structures, required on the GPU. Computing these on the CPU and then transferring them to the GPU involves high synchronization and bus transfer costs and leads to significant slowdowns.

When working with meshes, e.g., for finite element simulation, volumetric subdivision algorithms are often used to globally or locally refine a simulation mesh. However, subdivision schemes for surfaces and volumes alike require a large amount of information about neighborhood relations. Typically, the relationships between cells, faces, edges, and vertices are required. Computing these for large meshes is often too time consuming to do on the fly, whereas storing them all drastically increases memory consumption.

In this work, we propose a data structure for volumetric meshes based on a compact, ternary sparse matrix representation of boundary operators. The proposed data structure is capable of representing both manifold and non-manifold meshes as well as meshes with mixed element types. We examine the suitability of this structure for mesh processing on the GPU. To that end, we compare several mesh processing algorithms implemented in CUDA using our data structure and compare them with their CPU counterparts implemented using Kremer et al.'s OpenVolumeMesh library [KBK13]. As our approach focuses on efficient calculation and storage of neighborhood relations between mesh elements, we also implemented and analyzed the volumetric Catmull-Clark subdivision scheme as presented by Joy and MacCracken [JM96].

The rest of this paper is organized as follows: in Section 2, we provide an overview of terminology and related work. Section 3 describes the basic, theoretical concepts underlying our data structure. The practical implementation of the data structure and the algorithms thereon are presented in Section 4. Section 5 lists the results of our comparisons with OpenVolumeMesh. Finally, we summarize the paper and the results, and highlight potential directions for further research in Section 6.

2. Background and Related Work

In this section, we provide an overview of terminology and related work in the areas of array-based / index-based volumetric mesh data structures, mesh processing on GPUs and volumetric subdivision.

2.1. Array-based volumetric mesh data structures

Being based on boundary operators stored as sparse matrices to represent a mesh's topology, our storage format leads to contiguously laid out arrays in memory and addressing being done via column indices and offsets. Therefore, it is more closely related to array-based (sometimes also called index-based) volumetric mesh data structures, as opposed to pointer-based data structures such as most implementations of the half-edge / doubly connected edge list (DCEL) data structures introduced for surface meshes by Muller and Preparata [MP78] or the more recent linear cell complex data structure used in the CGAL library [Dam16].

Aside from the use of indices into arrays instead of pointers, most pointer-based data structures represent entities (edges, half-edges, faces, etc.) as individual objects while array-based data structures spread their attributes over one or more arrays and may not store intermediate entities such as half-edges and half-faces explicitly at all. Due to the allocation of various, potentially large, objects, pointer-based data structures frequently suffer significant memory fragmentation leading to unnecessarily high cache pressure. While advanced allocation techniques can reduce memory fragmentation to some degree, array-based data structures remain better suited for parallel and vectorized implementations, due to the separate, contiguous storage of individual entity properties. Furthermore, the number of bits used for indices and offsets can be chosen or adapted freely to save memory and reduce cache pressure, while the size of pointers is dictated by the system and is currently on the order of 8 bytes (64 bits) on most systems.

A variety of data structures for simplical complexes and tetrahedral meshes have been proposed by authors such as De Floriani et al. and Lage et al. [DH03, DGH04, LLLV05]. However, these data structures are limited to one cell type (tetrahedra) by definition, limiting their usefulness. In particular, volumetric Catmull-Clark subdivision is not possible. While it leads to hexahedral-dominant meshes, it can also lead to non-hexahedral cells, depending on the input mesh.

An overview of array-based mesh data structures by Alumbaugh and Jiao [AJ05] introduces an array-based variant of the half-edge data structure (array HEDS), and its generalization to volumetric meshes, the array-based half-face data structure (array HFDS). Like other data structures derived from DCEL, it only supports manifold meshes. And while it is very compact, it introduces the concept of anchored half-faces, which require a fixed local numbering of vertex indices per cell and cell face. Therefore, the set of allowed cell types (polyhedra) must be known beforehand.

More recently, array HFDS was extended to support mixed-dimensional and non-manifold meshes (AHF) by Dyedov et al. [DRE*15]. While the restriction to manifold meshes is lifted, a priori knowledge of the set of allowed cell types is still required. Although Catmull-Clark subdivision of surface meshes always leads to uniform quadrilateral meshes, volumetric subdivision cell types depend on the number of incident faces and edges to a vertex within a cell. For example, subdividing a four-sided pyramid leads to four hexahedra at the base and an octahedron with ten vertices and (non-planar) quadrilateral faces at the top, as the top vertex has four incident edges and faces, as shown in Fig. 2. Therefore, this limitation of AHF can be difficult to work around.

A different extension of DCEL to half-faces called OpenVolumeMesh was proposed by Kremer et al. [KBK13]. By using arrays of handles instead of doubly linked lists and using a full hierarchy of relationships, they are able to represent general polytopal complexes, both without requiring a priori knowledge of all cell types and without requiring a mesh to be manifold. While their representation is less compact than array HFDS or AHF, we consider it to be the main competitor to our data structure due to its similarities and generality.

A linear algebraic mesh representation that bears some similarity to our approach is presented in a technical note by DiCarlo et al. [DPS14]. They store characteristic matrices that map each k-face (see Section 3) to the unordered vertices as a binary compressed sparse row (CSR) matrix. While boundary operators can be derived

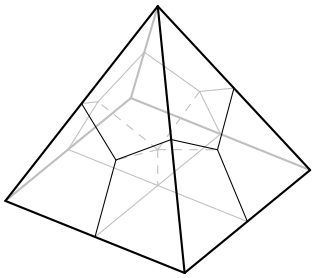


Figure 2: *Topology of a pyramid after Catmull-Clark subdivision without smoothing. The new cell connected to the top vertex is not hexahedral, but an octahedron with ten vertices and non-planar quadrilateral faces.*

efficiently, they can only be derived in an unoriented form, limiting the scheme's suitability to applications that do not require facet orientations.

2.2. Mesh processing on GPUs

Despite numerical simulation on GPUs being a widespread and well researched topic, GPGPU volumetric mesh processing is far less common. Surface mesh processing in the form of parallel mesh simplification has been examined by various authors, such as the works by DeCoro and Tatarchuk, Papageorgiou and Platis, and Odaker et al. [DT07, PP15, OKV15].

For volumetric meshes, a hybrid CPU-GPU algorithm for mesh optimization has been developed by D'Amato and Vénere [DV13]. While many operations are performed in parallel on the GPU, topological changes are done on the CPU. They achieve speedups between 2.8× and 6.6× compared to the sequential version.

A different approach is used in Cheng et al.'s paper on parallel optimization of volumetric meshes on heterogeneous systems [CSY*15]. To improve parallelizability, they avoid topological changes entirely and only modify vertex positions. This improves GPU performance significantly, as GPUs are sensitive to irregular workloads within thread blocks (groups of threads that are scheduled together) and divergent control flow within small thread groups (32-thread warps on NVIDIA GPUs), as GPUs internally use a model akin to masked SIMD (Single Instruction Multiple Data) operations to implement the SPMD (Single Program Multiple Data) programming model of OpenCL and CUDA. The speedups achieved compared to the serial version range between 14× and 21× when using only the GPU. They also attempt processing interior vertices on the GPU and exterior vertices on the CPU, both due to their lower number and the higher control flow divergence, but only achieve speedups between 11× and 16× compared to the sequential version due to the CPU-GPU communication overhead.

Very recently, during the initial review of this paper, Zayer et al. published a sparse matrix-based mesh representation for GPU mesh processing [ZSS17]. They use a compressed sparse column (CSC) encoding of a face table, mapping each face (or cell) to its ordered vertices. Additionally, they introduce the concept of action maps to modify the sparse matrix-vector and sparse matrix-matrix operations to achieve various mesh operations. However, these require all faces or cells to be of the same type, and does not allow for general polytopal meshes.

2.3. Volumetric Subdivision

Subdivision surfaces are widely used in computer graphics and computer animation to create smooth models with a relatively low number of degrees of freedom. A relatively coarse control mesh is refined iteratively by inserting new vertices, edges, and faces with every subdivision step. When repeating this subdivision process an infinite number of time, the geometry converges to the so-called limit surface that can also be calculated analytically for some subdivision schemes. Certain schemes require purely triangular control meshes, such as the subdivision scheme presented by Loop [Loo87]. Others operate on quad-based meshes or are able to handle arbitrary polygons like the Catmull-Clark surface subdivision scheme [CC78].

The concept of volumetric subdivision extends the idea of subdivision surfaces by an additional dimension. Volumetric control meshes are used and in addition to vertices, edges, and faces, new cells are created when subdividing. Similar to surface subdivision schemes for triangle meshes, some volumetric subdivision schemes operate on tetrahedral control meshes, such as the methods by Chang et al. [CMQ03] or Schaefer et al. [SHW04]. These are often used for global or local refinement of simulation meshes for FEM simulation, as shown by Burkahrt et al. [BHU10a]. In this paper, we utilize the Catmull-Clark subdivision solids described by Joy and MacCracken [JM96] to showcase our approach. As an extension to Catmull-Clark subdivision surfaces, the volumetric Catmull-Clark subdivision scheme operates on control meshes with arbitrary polyhedra.

In a recent conference paper, Altenhofen et al. presented an approach to generate tetrahedral simulation meshes directly from volumetric Catmull-Clark models [ASSF17]. As Altenhofen et al. already use a GPU-based FEM solver (originally presented by Weber et al. [WMA*15]), their approach would benefit significantly from performing subdivision and mesh operations directly on the GPU.

3. Concept

In this section, we describe the concepts used in our data structure for efficient volumetric mesh representation on GPUs. We represent a volumetric mesh using the discrete boundary operators ∂_k that describe the top down relationships from each k-face to its oriented $(k-1)$-face facets. While the principles outlined in this section are applicable to n-D meshes embedded in m-D space, the current implementation is limited to 3-D meshes in $\mathbb{R}^3$. To clarify the terminology of k-faces, a volumetric mesh in $\mathbb{R}^3$ consists of vertices (0-faces), edges (1-faces), faces (2-faces), and cells (3-faces).

The boundary operators ∂_k are linear operators that correspond to sparse, ternary matrices, i.e., all entries are values in $\{-1,0,1\}$ and most entries are zero. To make use of the large body of research on sparse matrix operations on GPUs, we use a ternary compressed sparse row (TCSR) representation for ∂_2 and ∂_3. We only store non-zero entries, and encode the sign of the non-zero entries in the sign

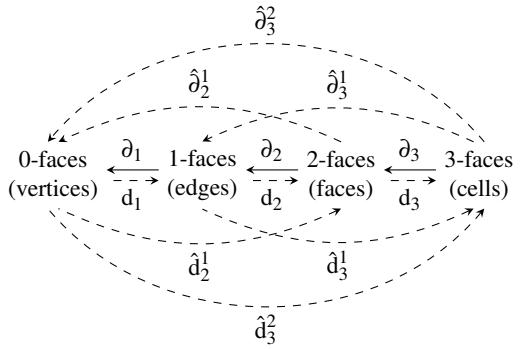


Figure 3: *This diagram shows how the different (co)boundary and chained operators describe the relations between k-faces. Only operators shown as solid lines are always stored. Dashed operators are computed and cached on demand.*

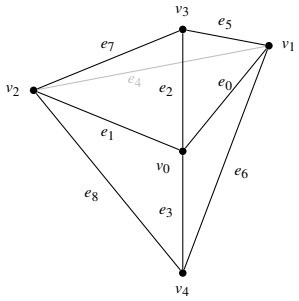
bit of the corresponding column index. ∂_1 is stored as a vector of ordered pairs instead.

When bottom-up relationships are required, these can be computed efficiently on demand from the boundary operators ∂_k by computing their transpose ∂_k^T. Therefore, efficient CSR transpose algorithms can be reused. These transposed boundary operators correspond to the boundary operators of the dual mesh or the coboundary operators $\mathrm{d}_k = \partial_k^T$. While ∂_1 is stored as a vector of ordered pairs, the dual edges $\mathrm{d}_3 = \partial_3^T$ are not, as faces on the outer boundary of a mesh only have one neighboring cell. Additionally, $\mathrm{d}_1 = \partial_1^T$ requires special treatment to convert from ordered pairs to TCSR.

Indirect relationships such as the list of all vertices belonging to a specific face can be computed by chaining boundary operators. Direct chaining/multiplication of two consecutive boundary operators always results in a zero matrix $\partial_k \partial_{k-1} = \mathbf{0}$, as each well-formed k-face must be 2-manifold and each $(k-2)$-face is therefore used twice, once in its positive and once in its negative orientation. For example, each edge (1-face) within a cell (3-face) is used exactly twice, in opposing directions, by two adjacent faces (2-faces). By first taking the elementwise absolute value $\hat{\partial}_{k,ij} = |\partial_{k,ij}|$ and then using $\max(\cdot,\cdot)$ as an additive field operator, the indirect relationships $\hat{\partial}_k^n = \prod_{i=0}^{n} \hat{\partial}_{k-i}$ can be computed using sparse general matrix-matrix products (SpGEMM).

An overview of the described operators is shown in Fig. 3. An example mesh, along with the associated boundary operator matrices ∂_k, is given in Fig. 4. While the direct, top-down boundary operators ∂_k are always stored, all other operators / relations d_k, $\hat{\partial}_k^n$, and $\hat{\mathrm{d}}_k^n$ are computed and cached on demand.

Many of these concepts such as boundary and coboundary operators are shared with discrete differential geometry (DDG) / discrete exterior calculus (DEC) (see, e.g., Desbrun et al.'s workshop paper on discrete differential forms [DKT06]) and originate in algebraic topology (see, e.g., Hatcher's book on algebraic topology [Hat02]). While most treatises on DEC limit themselves to simplical complexes (meshes consisting only of simplices), our representation based on boundary operators stored in TCSR matrices is not limited to simplical complexes and can represent meshes with arbitrary polyhedral cells. Storage of non-manifold meshes is possible as well. Due to the common concepts, our storage format could be useful in the implementation of DEC simulation codes.



$$\partial_3 = \begin{array}{c} \begin{matrix} f_0 & f_1 & f_2 & f_3 & f_4 & f_5 & f_6 \end{matrix} \\ \begin{pmatrix} 1 & -1 & 0 & 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} c_0 \\ c_1 \end{matrix}$$

$$\partial_2 = \begin{array}{c} \begin{matrix} e_0 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{matrix} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{matrix}$$

$$\partial_1 = \begin{array}{c} \begin{matrix} v_0 & v_1 & v_2 & v_3 & v_4 \end{matrix} \\ \begin{pmatrix} -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \end{pmatrix} \end{array} \begin{matrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{matrix}$$

Figure 4: *Example mesh with two tetrahedra c_0 and c_1 connected by a common face f_0, along with the resulting boundary operators. Boundary operators ∂_k describe which $(k-1)$-face facets (columns) form a k-face (rows) and their orientation (signs). Negative values indicate reversed orientation or the starting vertex in the case of edges.*

In this paper, we focus on bulk operations (processing multiple elements, instead of individual elements). This is necessary for parallelization and to fully load the GPU, which is not possible when processing individual elements or small numbers of elements. However, such operations could be useful even if they are slower than on CPU, due to the high communication cost over the PCIe bus.

4. Implementation

In this section, we detail the implementation of our data structure and the associated algorithms. As mentioned in previous sections, we use a compact, ternary variant of CSR sparse matrices, TCSR. The basic CSR data structure consists of three 1-D arrays:

1. `columns`[`nnz`], the column indices for each non-zero entry, where `nnz` is the number of non-zero entries is the matrix.
2. `values`[`nnz`], the values of each non-zero entry.
3. `offsets`[`nrows` + 1], the starting indices of each row in the `columns` and `values` arrays, where `nrows` is the number of rows of the matrix. The offset of the following row is one past the end of the current row. An additional entry equal to `nnz` is appended, so that no special case is required for the last row.

As the boundary operators only contain entries in $\{-1,0,1\}$, TCSR discards the `values` array and encodes the sign of the entry in `columns`:

$$\texttt{columns}_{\text{TCSR}}[\texttt{i}] = \begin{cases} -\texttt{columns}[\texttt{i}]-1 & \text{if } \texttt{values}[\texttt{i}] = -1 \\ \texttt{columns}[\texttt{i}] & \text{if } \texttt{values}[\texttt{i}] = 1, \end{cases}$$

where the operation $-i-1$ corresponds to the bitwise negation of the two's-complement integer i. Zero entries are not stored explicitly. Decoding is very simple, as every encoded column entry is negative if the value is -1. On most architectures, an optimizing compiler will replace all conditionals in encoding and decoding by combinations of `sar` (sign-extending, arithmetic shift right) and `xor` (bitwise exclusive or) instructions. This is particularly advantageous on GPUs, as control flow divergence is avoided. On sign-magnitude architectures, a different encoding would be preferable. Our current implementation uses 32-bit integers for all offsets and encoded values.

As mentioned in Section 3, all operators and indirect relationships except ∂_1 are stored in the TCSR format. The edges described by ∂_1 are stored as ordered pairs of vertices, as the number of vertices per edge is always two. While edges could be stored in TCSR form as well, using ordered pairs removes the need for an offset array, saving memory. Furthermore, no encoding/decoding of the vertex's sign and index is necessary, and it is known a priori which of the vertices is the starting vertex. In summary, our data structure consists of the following arrays:

1. `cellFaces`[`ncf`], the encoded column/value pairs of ∂_3, where `ncf` is the number of non-zero entries of ∂_3.
2. `cellFaceOffsets`[`ncells` + 1], the row offsets of ∂_3, where `ncells` is the number of rows of ∂_3 and equal to the number of cells in the mesh.
3. `faceEdges`[`nfe`], the encoded column/value pairs of ∂_2, where `nfe` is the number of non-zero entries of ∂_2.
4. `faceEdgeOffsets`[`nfaces` + 1], the row offsets of ∂_2, where `nfaces` is the number of rows of ∂_2 and equal to the number of faces in the mesh.
5. `edgeVertices`[`nev` = 2 · `nedges`], ∂_1 stored as ordered pairs of vertex indices, where `nev` is the number of non-zero entries of ∂_1 and always equal to twice the number of edges `nedges`. No TCSR encoding is used.
6. `positions`[`nvertices`][3], the positions of the vertices in $\mathbb{R}^3$, where `nvertices` is the number of vertices in the mesh.

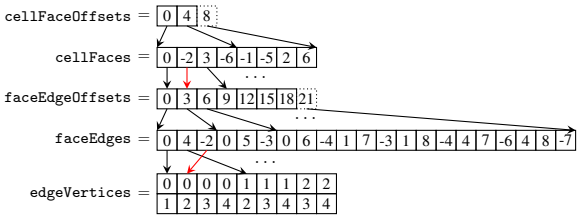


Figure 5: *In-memory representation of the boundary operators of the mesh in Fig. 4 using our TCSR encoding. Red arrows indicate that the associated facet is used in its inverted orientation. The offsets point to positions between entries, as each pair indicates a range. The borders of the last offset entries are dotted, as they do not belong to a cell or face.*

7. The inverse and indirect relations d_k, $\hat{\partial}^n_k$, and $\hat{\mathrm{d}}^n_k$ are all stored in the TCSR format, but only allocated and computed on demand. If computed, they are cached in our data structure for reuse until invalidated by mutable access to any operator ∂_k on which they depend, or explicitly purged by the user.

The in-memory representation of the example mesh's boundary operators in Fig. 4 is illustrated in Fig. 5. While the basic data structure does not contain additional properties beyond vertex positions, such properties can be added to any operator by using supplementary value arrays for the various TCSR matrices or for the desired entities.

4.1. Coboundary Operators and Basic Queries

To compute the coboundary operator matrices from the boundary operators, the boundary matrices must be transposed. Transposition of CSR matrices is equivalent to conversion to the compressed sparse column (CSC) format, and reinterpreting columns as rows and vice versa. The algorithm TCSRTOTCSC used to transpose the matrices follows the count-scan-fill pattern that underlies many of the procedures outlined in this paper and is given in Algorithm 1.

The ATOMICINC subprocedure used is an atomic increment operation that returns the old value before addition. As many separate counters are used, contention is minimal. INCLUSIVESCAN is a parallel, inclusive cumulative sum, i.e., the sequence $[3,1,0,2]$ is transformed into $[3,4,4,6]$ in parallel, as implemented in libraries such as Thrust [BH15]. The ENCODETCV operation is the encoding described in the previous section, and DECODETCV is its inverse. At the end of TCSRTOTCSC, all rows are sorted (in parallel) by SORTROWS according to their decoded column index to achieve a deterministic order. In our current implementation, each thread of the SORTROWS kernel performs a sequential sort of a single row. In summary, the number of outputs is first *counted* (while storing local offsets, if necessary), then an inclusive *scan* is performed to compute the offsets in the result from the counts. Finally, the output is *filled* with the desired values, for which the offsets are now known, allowing the operation to be performed in parallel. The transposition of edges uses a modified version of TCSRTOTCSC, where the even indices correspond to positive values and odd in-

Algorithm 1 Pseudocode for the ternary CSR to ternary CSC transposition procedure. The individual subprocedures are explained in the text.

```
1:  procedure TCSRToTCSC(M)
2:      counts ← 0                                   ▷ M.ncols zeros
3:      for o ← [0..M.nnz) do                        ▷ In parallel
4:          c, v ← DecodeTCV(M.columns[o])
5:          entryOffsets[o] ← AtomicInc(counts[c])
6:      end for
7:      R.offsets ← [0] + InclusiveScan(counts)
8:      for r ← [0..M.nrows) do                      ▷ In parallel
9:          for o ← [M.offsets[r]..M.offsets[r+1]) do
10:             c, v ← DecodeTCV(M.columns[o])
11:             e ← EncodeTCV(r, v)
12:             o ← R.offsets[c] + entryOffsets[o]
13:             R.columns[o] ← e
14:         end for
15:     end for
16:     R ← SortRows(R)                              ▷ Optional, for determinism
17:     return R
18: end procedure
```

dices to negative values. The offsets of the input are known as well, as each row has exactly two offsets.

As mentioned in Section 3, the computation of indirect indices corresponds to an SpGEMM operation with modified operations. While the GraphBLAS forum, as founded by Mattson et al. [MBB*13], is building a library of graph operations in the language of linear algebra, additional assumptions can be made for well formed meshes. Therefore, we use specialized algorithms instead of a modified SpGEMM procedure.

$\hat{\partial}_2^1$ (`faceVertices`) can be computed without previously counting the vertices of each face, as each face forms a closed loop. For $\hat{\partial}_3^1$ (`cellEdges`), the number of unique edges per cell can be computed as the sum of edges of each cell divided by two, as each edge is used once in each direction. Following the count-scan-fill pattern and after performing an inclusive scan on these counts to compute the offsets, the output columns can be allocated according to the last offset entry. Finally, a last pass over each cell's face is performed and positively oriented edge uses (`cellFaceOrientation` · `faceEdgeOrientation` $= 1$) are stored in the column array. Due to the double indirection in $\hat{\partial}_3^2$, orientation can no longer be used to discard multiple references. Therefore, the number of edges per cell is computed as above without division as each edge has two vertices. After the scan and fill operations, the per-cell vertex arrays are sorted in memory and duplicate entries are moved to the back and set to a negative number while the new sizes are written to a second count array. Finally the counts are accumulated again and the vertex array is compacted to remove negative entries using Thrust. As the aforementioned assumptions only hold for the primal mesh, the indirect bottom-up relationships are computed from their top-down duals $\hat{\mathrm{d}}_k^n = \hat{\partial}_k^{n,T}$.

4.2. Boundary Extraction and Laplacian Smoothing

Using the d_3 coboundary operator that maps faces to cells, the set of boundary faces is trivial to compute. In a manifold mesh, each boundary face has exactly one cell that uses it, therefore only the distance of the offsets within d_3 is required to create a boolean mask of boundary faces. To find the orientation of the boundary face, the orientation of the cell is read from d_3 and inverted. For non-manifold meshes, a different approach is required, as a face may have no cells that use it causing both half-faces to be part of the boundary set. Additionally, a single orientation may be used more than once causing a boundary face to have two or more cells using it. However, even in the non-manifold case, all necessary information is present in d_3. To mark boundary edges and vertices, an additional pass over d_2 and $\hat{\mathrm{d}}_2^1$ is required, respectively. With the boolean markers given, extracting boundary elements becomes a simple compaction.

Laplacian smoothing of inner vertices also requires the set of boundary vertices, as these should remain unaffected. This mask is computed once, then smoothing is performed iteratively. Each smoothing iteration sets a vertex's position to the average of the positions of its neighbors. To perform this smoothing in parallel, separate input and output arrays of positions are used that are swapped between iterations using an efficient pointer exchange. The set of neighbors is determined by iterating over the corresponding entries in ∂_1 and d_1, which is also computed once beforehand. Which entry of ∂_1 has to be read is determined by the sign of the entry in d_1.

4.3. Catmull-Clark Subdivision

While boundary extraction and Laplacian smoothing only use but do not affect or create topological information, volumetric Catmull-Clark subdivision requires the creation of new vertices, edges, faces, and cells. The volumetric Catmull-Clark subdvision rules are defined in a technical report by Joy and MacCracken [JM96]. A compact summary can be found in a paper by Burkhart et al. [BHU10b].

In the first step, the number of new entities is determined to allocate the output mesh's arrays:

$$|V_{CC}| = |V| + |E| + |F| + |C| \tag{1}$$

is the new number of vertices and equals the sum of the old vertex, edge, face, and cell counts. Equivalently, these are the number of nonzeros in the identity matrices mapping each entity to itself. Each existing vertex is maintained, while each edge, face, and cell are mapped to a new point at the center of the corresponding entity before smoothing.

$$|E_{CC}| = \mathrm{nnz}(\partial_1) + \mathrm{nnz}(\partial_2) + \mathrm{nnz}(\partial_3) \tag{2}$$

is the new number of edges and equals the total number of nonzeros (nnz) in all boundary operators. Referring back to Fig. 2, it can be observed that each edge is split, i.e., there is one "edge-edge" for each vertex of each edge. Additionally, for each face in the mesh, a "face-edge" is added for each edge of each face, connecting the new edge vertices to the new face vertex. Finally, a "cell-edge" is added for each face of each cell, connecting the new face vertices

to the new cell vertex.

$$|F_{CC}| = \mathrm{nnz}(\hat{\partial}_2^1) + \mathrm{nnz}(\hat{\partial}_3^1) \tag{3}$$

is the new number of faces and adds another level of indirection. Each face is split along the new "face-edges" connecting these with the "edge-edges" of edges that share the face's vertex. Similarly, "cell-faces" are created for the "cell-edges" and "face-edges" that share the cell's edge.

$$|C_{CC}| = \mathrm{nnz}(\hat{\partial}_3^2) \tag{4}$$

is the new number of cells. One for each vertex within each cell. Each "cell-cell" is enclosed by the "face-faces" corresponding to that vertex and the "cell-faces" connecting these back to the cell point. In summary, all element counts can be determined directly from the TCSR matrices, and the offsets of these matrices can directly be used as element offsets in the new lists of elements. However, all top-down relationships must be computed before subdivision.

The positions of the new "cell-vertices" are computed as

$$\mathbf{p}_{CC,C} = \hat{\partial}_3^2 \mathbf{p} \cdot \mathrm{diag}(\hat{\partial}_3^2 \mathbf{1})^{-1} = \mathrm{AVG}_{\hat{\partial}_3^2}(\mathbf{p}), \tag{5}$$

where $\mathbf{p}$ is the vector of 3-vectors of input vertex positions, $\mathbf{1}$ is a vector of scalar 1-values, and $\mathrm{diag}(\mathbf{x})$ creates a square matrix from a vector $\mathbf{x}$ of diagonal entries. This corresponds to the cell centers, defined as a simple average of the positions of all vertices used by each cell. For the other positions, additional helper positions

$$\hat{\mathbf{p}}_F = \mathrm{AVG}_{\hat{\partial}_2^1}(\mathbf{p}) \tag{6}$$

and

$$\hat{\mathbf{p}}_E = \mathrm{AVG}_{\hat{\partial}_1}(\mathbf{p}) \tag{7}$$

are computed. The matrix multiplication and division by the number of entries in Eqs. (5) to (7) are fused in a single GPU kernel.

The final "face-vertex" positions also require the calculation of the boundary as described in Section 4.2. While the topological operations require all top-down relationships, the computation of the final positions requires all bottom-up relationships.

$$\mathbf{p}_{CC,F} = \begin{cases} \hat{\mathbf{p}}_F & \text{if boundary} \\ \dfrac{\hat{\mathbf{p}}_F + \mathrm{AVG}_{\hat{\partial}_3}(\mathbf{p}_{CC,C})}{2} & \text{otherwise,} \end{cases} \tag{8}$$

which corresponds to the centers of the original faces for faces on the boundary (as in Catmull-Clark subdivision surfaces) and to a weighted average of the face center and the centers of the two neighboring cells for non-boundary faces. The individual faces are processed using one thread per face. The "edge-vertex" and new vertex positions are computed in a similar manner as weighted averages of their centers and the center points of all connected higher-level entities (i.e., edges, faces and cells).

After computing all positions, the new mesh's topology must be constructed. This is again done for each set of entities separately. First, edges are created with the orientations shown in Fig. 6. Edges are split according to the following rule:

$$e : (v_1, v_2) \rightarrow \begin{matrix} 2e : (v_1, e + |V|) \\ 2e+1 : (e + |V|, v_2) \end{matrix}, \tag{9}$$

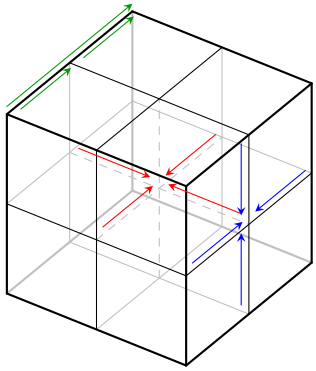


Figure 6: *Catmull-Clark edge orientations. "Edge-edges" (green) use the orientation of the original edge. "Face-edges" (blue) and "cell-edges" (red) point towards the corresponding face and edge points, respectively.*

where e is the index of the edge connecting the vertices with the indices v_1 and v_2. Face edges are created as follows:

$$f : (o_1 : e_1, \ldots, o_n : e_n) \rightarrow \\ o_i + \mathrm{nnz}(\partial_1) : (e_i + |V|, f + |V| + |E|)|_{i=1}^{n}, \tag{10}$$

where f is the index of the face consisting of the edges $e_1, \ldots, e_n$ with the entries at the offsets $o_1, \ldots, o_n$. Cell edges are created analogously.

As for the edges, the size of each face is known a priori. After Catmull-Clark subdivision every face is a quadrilateral. For the "face-faces", the orientation of the original face is kept, as shown in Fig. 7. The orientation of the "cell-faces" however cannot be derived from the original mesh directly. In order to orient them consistently, we decided to define the "cell-face" in a way that its normal vector is oriented in the global direction of the original edge whose edge point is used in the current "cell-face". This is done using only the topological information, in particular the orientation in which the two faces that reference it use the edge, and not the positions of the mesh.

While the number of resulting cells can be determined directly, including their index, each "cell-cell" has a varying number of indices (Figure 2 shows an example with a non-hexahedral cell). Therefore, the count-scan-fill pattern is used again. To count the number of "cell-faces" per "cell-cell", we iterate over the entries of $\hat{\partial}_3^2$ for each cell in parallel and count the number of faces within each cell that use the vertex. This number is additionally multiplied by two to account for the "edge-faces" belonging to the edges between the faces.

After the scan, the output mesh's `cellFaces` array can be filled. As during counting, we iterate over $\hat{\partial}_3^2$ and find the cell's faces and edges that use the vertex. The "face-faces" are used in the same orientation as the corresponding faces. The orientation of "cell-faces" are determined by comparing the starting vertex with the current vertex index.

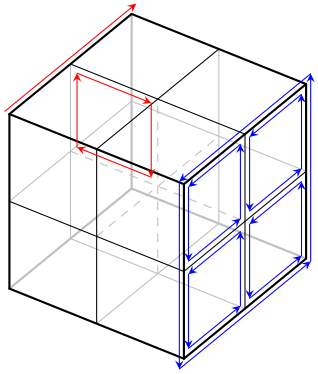


Figure 7: *Catmull-Clark face orientations. "Face-faces" (blue) are oriented in the same direction as the global orientation of the originating face. "Cell-faces" (red) are oriented according to the global orientation of the originating edge.*

5. Results

All benchmarks were compiled using Microsoft Visual Studio 2015 Update 3 and NVIDIA CUDA 8.0. The measurements were made on a desktop PC with Windows 10 64-bit, an Intel Core i7-6700 CPU, 32 GB of DDR4-2133 main memory and a GeForce GTX 1080 GPU with 8 GB of GDDR5X GPU memory. All benchmarks have been repeated 20 times and averaged. No outliers were observed. Dynamic updating of bottom-up connectivity was disabled in all OpenVolumeMesh benchmarks, as recommended by Kremer et al. [KBK13]. In addition to the GPU version of our data structure and algorithms described in Section 4, a CPU version that replaces all parallel algorithms by their direct, serial equivalents has been implemented and evaluated.

Table 1 (level 0) lists the sizes of the three tetrahedral meshes used in our evaluation. Table 2 lists the memory requirements of all meshes given in Table 1. While similar information is stored, our TCSR-based representation is more compact than OpenVolumeMesh and uses between 27% and 36% less memory. Figure 8 shows a sliced view of each model to give the reader a better impression of the resolutions of the individual meshes.

In the following, the results are organized in subsections in the same order as in Section 4. The results given in Sections 5.1 and 5.2 all refer to the input meshes (level 0 in Table 1).

5.1. Coboundary Operators and Basic Queries

To compare the computation times of the coboundary operators d_k, we first copied the OpenVolumeMesh mesh to a new mesh instance with disabled bottom-up indices. We then measured the time taken to enable each individual bottom-up index. For our mesh data structure, we equivalently measured the times taken to transpose the boundary operators ∂_k. For the indirect relationships $\hat{\partial}_k^n$, we compared computing them using our data structure to iterating over the mesh using the corresponding iterator in OpenVolumeMesh while only incrementing a counter.

The results of this comparison are shown in Table 3. While the speeds for the smallest armadillo model are comparable between OVM and our GPU implementation, for all but face transposition, face vertex iteration, and cell transposition, where we measured speedups of 16×, 18×, and 89×, respectively. This could be caused by a bug or inefficient implementation in OpenVolumeMesh. Only the transpose of the edges of the smallest model shows a slight slowdown of 0.86×. For the large dragon model, significant speedups between 2.9× and 531× are achieved in all cases. Comparing OVM with our CPU implementation, good speedups up to 149× are achieved in all cases, even for small models with speedups of up to 98×. The only exception is the computation of cell edges, with a slight slowdown between 0.62× and 0.74×. As expected, the GPU version is much faster than the other variants for sufficiently large models.

Table 1: *Mesh sizes for the armadillo, dragon and bunny models on different subdivision levels. The initial meshes are tetrahedral meshes, after subdivision they are hexahedral meshes.*

Mesh	Lvl.	Vertices	Edges	Faces	Cells
Armadillo	0	1.01k	5.08k	7.23k	3.16k
	1	16.4k	44.4k	40.6k	12.6k
	2	114k	327k	314k	101k
	3	857k	2.52M	2.47M	809k
	4	6.65M	19.8M	19.6M	6.47M
Bunny	0	10.7k	64.7k	104k	50k
	1	229k	641k	612k	200k
	2	1.68M	4.93M	4.85M	1.60M
	3	13.1M	38.9M	38.6M	12.8M
Dragon	0	232k	1.21M	1.80M	825k
	1	4.06M	11.1M	10.3M	3.30M

Table 2: *Calculated mesh sizes in bytes of the armadillo, bunny and dragon models using OpenVolumeMesh (OVM) or our data structure. All suffixes in this table are binary suffixes ($2^{10}, 2^{20}, \ldots$).*

Mesh	Lvl.	OVM	Ours
Armadillo	0	348k	226k
	1	2.25M	1.64M
	2	17.3M	12.5M
	3	135M	97.7M
	4	1.05G	773M
Bunny	0	4.92M	3.16M
	1	33.8M	24.5M
	2	266M	192M
	3	2.06G	1.49G
Dragon	0	85.1M	55.0M
	1	573M	417M

5.2. Boundary Extraction and Laplacian Smoothing

In the case of boundary extraction, we compared building the list of boundary face indices to using the provided boundary face iterator in OpenVolumeMesh while only incrementing a counter. Laplacian smoothing in OpenVolumeMesh was implemented by repeatedly iterating over all vertices, checking if they are on the boundary,

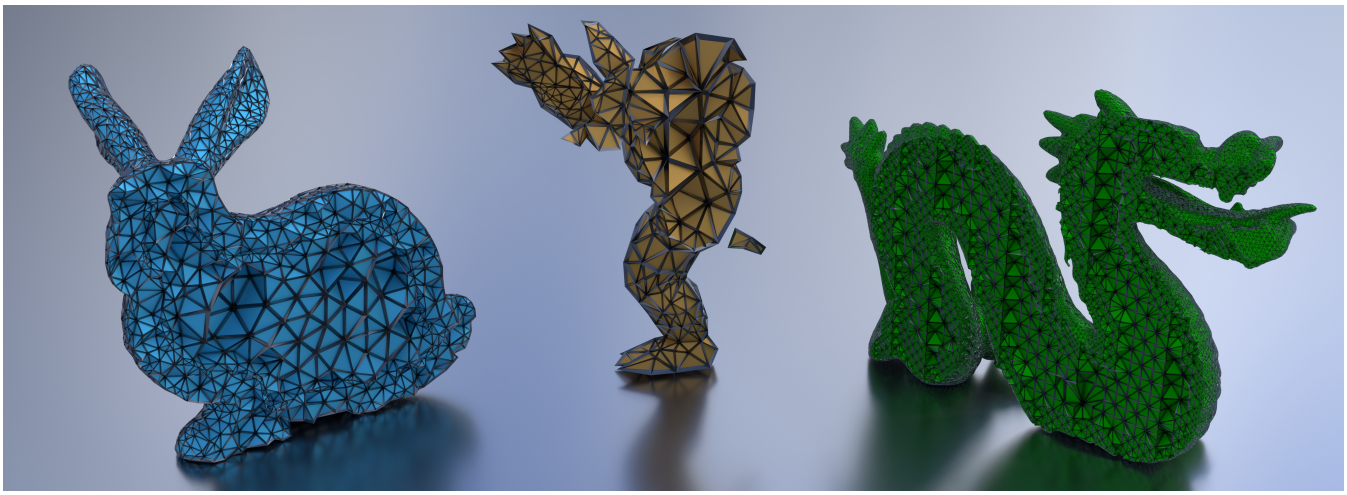


Figure 8: *Sliced views of the three models used in the comparison give an impression of the initial mesh resolutions. The armadillo has the lowest resolution, followed by the bunny, while the dragon has the highest resolution of the three.*

Table 3: *Benchmark results for the calculation of inverse relations as well as indirect relations. For OpenVolumeMesh (OVM), we used the iterators provided by the library. All times are given in milliseconds.*

Operation	Method	Mesh		
		Armadillo	Bunny	Dragon
Transpose Cells	CPU	0.20	2.49	114
	GPU	0.22	0.70	14.0
	OVM	19.6	372	6788
Transpose Faces	CPU	0.26	3.07	65.6
	GPU	0.20	1.17	7.02
	OVM	3.19	56.3	1424
Transpose Edges	CPU	0.16	1.71	35.9
	GPU	0.72	0.58	4.11
	OVM	0.62	5.73	184
Cell Edges	CPU	3.08	46.7	935
	GPU	0.98	22.2	237
	OVM	1.91	34.1	691
Cell Vertices	CPU	3.74	57.2	1236
	GPU	2.72	25.1	261
	OVM	3.90	65.5	1369
Face Vertices	CPU	0.05	0.85	17.3
	GPU	0.12	0.44	1.88
	OVM	2.08	26.3	558

and averaging their neighbor positions (again using the provided iterator).

The results of these comparisons are given in Table 4. Laplacian smoothing is much faster in our GPU implementation and shows a speedup of 5.3× to 289× for the smallest and largest models, respectively. Our CPU implementation achieves a speedup between 35× and 48×, and is not significantly affected by mesh size.

Table 4: *Benchmark results for the extraction of the outer surface as well as Laplacian smoothing of inner vertices. All times are given in milliseconds.*

Operation	Method	Mesh		
		Armadillo	Bunny	Dragon
Surface Extraction	CPU	0.03	0.28	5.13
	GPU	1.13	1.15	2.83
	OVM	0.15	1.21	22.7
Laplacian Smoothing	CPU	0.54	21.4	860
	GPU	3.58	4.48	116
	OVM	19.0	1026	33474

The GPU speedup for boundary computation is 8× for the largest model, but exhibits a slowdown of 0.14× on the smallest model. However, this is not surprising as the boundary consists of a very small number of triangles on the smallest model and is insufficient to load the GPU fully. The speedup of the CPU implementation is again mostly independent of mesh size and between 4.3× and 5×.

5.3. Catmull-Clark Subdivision

For volumetric Catmull-Clark subdivision, a direct comparison between subdivision using our data structure and subdivision using OpenVolumeMesh and its iterators has been performed. As OpenVolumeMesh was significantly slower, only the first subdivision level was computed. For our data structure, we also measure the time taken to compute the full set of indirect and bottom-up relationships separately. The sizes of the subdivided meshes are given in Table 1. A visualization of the subdivision of the armadillo model is shown in Fig. 1.

The results of this comparison are given in Table 5. Using our GPU version, the speedups for the first subdivision level range from 45× to 166×, and even the large dragon model takes less than a

Table 5: *Benchmark results for volumetric subdivision of the three meshes. For OpenVolumeMesh (OVM), only one level of subdivision has been computed. For our approach, we also measured the time required for pre-calculating the indirect relations needed for the subdivision process. All times are given in milliseconds.*

Subdiv. Level	Operation	Mesh		
		Armadillo	Bunny	Dragon
1	CPU Pre.	7.50	120	2692
	CPU Total	13.4	159	3844
	GPU Pre.	6.33	57.3	553
	GPU Total	8.71	66.6	754
	OVM	392	7058	125625
2	CPU Pre.	33.9	611	—
	CPU Total	60.7	1010	—
	GPU Pre.	13.8	173	—
	GPU Total	18.3	236	—
3	CPU Pre.	294	4847	—
	CPU Total	487	8013	—
	GPU Pre.	93.1	1036	—
	GPU Total	121	1401	—
4	CPU Pre.	2333	—	—
	CPU Total	3868	—	—
	GPU Pre.	537	—	—
	GPU Total	717	—	—

second for subdivision. The CPU version achieves speedups between 47× and 59×. The additional subdivision levels (limited by the two-second GPU kernel timeout on Windows), show promising results as well.

6. Conclusion and Future Work

In summary, we have described a novel data structure for volumetric mesh representation based on compact ternary sparse matrices storing boundary operators. Furthermore, we have examined the suitability of this data structure for mesh processing on GPUs by implementing computation of indirect and reverse relationships, Laplacian smoothing of inner vertices, and volumetric Catmull-Clark subdivision. These implementations have been compared to their equivalents using OpenVolumeMesh, achieving speedups of up to 3× to 531×, more than two orders of magnitude, for sufficiently large meshes.

While these results are extremely promising, several potential avenues for future research remain. Using modified CSR to represent sparse matrices is very compact, but GPU memory coalescing can affect performance significantly. Due to the extreme irregularity of the boundary operator matrices and particularly the coboundary operator matrices, a ternary version of the Bin-CSR format for highly irregular matrices on GPUs presented by Weber et al. [WBS*13] could provide improved performance. Alternatively, more efficient ways to operate on CSR matrices such as LightSpMV by Liu and Schmidt [LS15] and the efficient SpGEMM for irregular matrices by Liu and Vinter [LV14] could be adapted for processing of TCSR matrices. Besides possibilities to improve performance, another avenue of research are in-place topological changes, such as adding a layer of cells at the outer surface. These would benefit from a matrix format designed for dynamic matrices such as the DCSR format presented by King et al. [KGKM16].

Acknowledgments

This work was supported in parts by the European Union Horizon 2020 project CAxMan (H2020-FoF-2015-680448), the German Federal Ministry for Economic Affairs and Energy (BMWi) under Grant No. 01MT16004A, and the Fraunhofer-internal program “Market-oriented strategic preliminary research” under Grant No. MAVO 828424.

References

[AJ05] ALUMBAUGH T. J., JIAO X.: Compact array-based mesh data structures. In *Proceedings of the 14th International Meshing Roundtable* (2005), Hanks B. W., (Ed.), pp. 485–503. doi:10.1007/3-540-29090-7_29. 2

[ASSF17] ALTENHOFEN C., SCHUWIRTH F., STORK A., FELLNER D.: Implicit mesh generation using volumetric subdivision. In *Workshop on Virtual Reality Interaction and Physical Simulation* (Lyon, France, 2017), Jaillet F., Zara F., (Eds.), VRIPHYS'17. doi:10.2312/vriphys.20171079. 3

[BH15] BELL N., HOBEROCK J.: Thrust 1.8.1, Mar. 2015. URL: https://thrust.github.io/. 5

[BHU10a] BURKHART D., HAMANN B., UMLAUF G.: Adaptive and feature-preserving subdivision for high-quality tetrahedral meshes. *Comput. Graphics Forum 29*, 1 (2010), 117–127. doi:10.1111/j.1467-8659.2009.01581.x. 3

[BHU10b] BURKHART D., HAMANN B., UMLAUF G.: Iso-geometric finite element analysis based on Catmull-Clark subdivision solids. *Comput. Graphics Forum 29*, 5 (2010), 1575–1584. doi:10.1111/j.1467-8659.2010.01766.x. 6

[CC78] CATMULL E., CLARK J.: Recursively generated B-spline surfaces on arbitrary topological meshes. *Comput.-Aided Des. 10*, 6 (1978), 350–355. doi:10.1016/0010-4485(78)90110-0. 3

[CMQ03] CHANG Y., MCDONNELL K. T., QIN H.: An interpolatory subdivision for volumetric models over simplicial complexes. In *2003 Shape Modeling International* (2003), IEEE, pp. 143–152. doi:10.1109/smi.2003.1199610. 3

[CSY*15] CHENG Z., SHAFFER E., YEH R., ZAGARIS G., OLSON L.: Efficient parallel optimization of volume meshes on heterogeneous computing systems. *Engineering with Computers* (2015). doi:10.1007/s00366-014-0393-7. 3

[Dam16] DAMIAND G.: CGAL 4.9 user manual, 2016. ch. Linear Cell Complex. URL: http://doc.cgal.org/latest/Linear_cell_complex/index.html. 2

[DGH04] DE FLORIANI L., GREENFIELDBOYCE D., HUI A.: A data structure for non-manifold simplical *d*-complexes. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (Nice, France, 2004), SGP '04, pp. 83–92. doi:10.1145/1057432.1057444. 2

[DH03] DE FLORIANI L., HUI A.: A scalable data structure for three-dimensional non-manifold objects. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (Aachen, Germany, 2003), SGP '03. 2

[DKT06] DESBRUN M., KANSO E., TONG Y.: Discrete differential forms for computational modeling. In *ACM SIGGRAPH 2006 Courses* (Boston, MA, 2006), pp. 39–54. doi:10.1145/1185657.1185665. 4

[DPS14] DiCarlo A., Paoluzzi A., Shapiro V.: Linear algebraic representation for topological structures. *Computer-Aided Design 46* (2014), 269–274. 2013 SIAM Conference on Geometric and Physical Modeling. doi:10.1016/j.cad.2013.08.044. 2

[DRE*15] Dyedov V., Ray N., Einstein D., Jiao X., Tautges T. J.: AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold meshes. *Engineering with Computers 31* (2015), 389–404. doi:10.1007/s00366-014-0378-6. 2

[DT07] DeCoro C., Tatarchuk N.: Real-time mesh simplification using the GPU. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (Seattle, WA, 2007), I3D '07. doi:10.1145/1230100.1230128. 3

[DV13] D'Amato J. P., Vénere M.: A CPU-GPU framework for optimizing the quality of large meshes. *J. Parallel Distrib. Comput. 73*, 8 (2013), 1127–1134. doi:10.1016/j.jpdc.2013.03.007. 3

[Hat02] Hatcher A.: *Algebraic Topology*. Cambridge University Pr., 2002. 4

[JM96] Joy K. I., MacCracken R.: *The Refinement Rules for Catmull-Clark Solids*. Tech. Rep. CSE-96-1, University of California, Davis, 1996. 2, 3, 6

[KBK13] Kremer M., Bommes D., Kobbelt L.: OpenVolumeMesh: A versatile index-based data structure for 3D polytopal complexes. In *Proceedings of the 21st International Meshing Roundtable* (2013), Jiao X., Weill J., (Eds.), pp. 531–548. doi:10.1007/978-3-642-33573-0_31. 2, 8

[KGKM16] King J., Gilray T., Kirby R. M., Might M.: Dynamic sparse-matrix allocation on GPUs. In *High Performance Computing. ISC High Performance 2016* (Frankfurt, Germany, 2016), Kunkel J. M., Balaji P., Dongarra J., (Eds.), vol. 9697 of *Lecture Notes in Computer Science*, pp. 61–80. doi:10.1007/978-3-319-41321-1_4. 10

[LLLV05] Lage M., Lewiner T., Lopes H., Velho L.: CHF: A scalable topological data structure for tetrahedral meshes. In *XVIII Brazilian Symposium on Computer Graphics and Image Processing* (2005), SIBGRAPI'05. doi:10.1109/sibgrapi.2005.18. 2

[Loo87] Loop C. T.: *Smooth Subdivision Surfaces Based on Triangles*. Master's thesis, University of Utah, 1987. 3

[LS15] Liu Y., Schmidt B.: LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors* (Toronto, ON, Canada, 2015), ASAP, pp. 82–89. doi:10.1109/ASAP.2015.7245713. 10

[LV14] Liu W., Vinter B.: An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Phoenix, AZ, 2014), pp. 370–381. doi:10.1109/IPDPS.2014.47. 10

[MBB*13] Mattson T., Bader D., Berry J., et al.: Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference* (2013). doi:10.1109/hpec.2013.6670338. 6

[MP78] Muller D. E., Preparata F. P.: Finding the intersection of two convex polyhedra. *Theoretical Computer Science 7*, 2 (1978), 217–236. doi:10.1016/0304-3975(78)90051-8. 2

[OKV15] Odaker T., Kranzlmueller D., Volkert J.: GPU-accelerated real-time mesh simplification using parallel half edge collapses. In *Revised Selected Papers of the 10th International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science. MEMICS 2015* (Telǎź, Czech Republic, 2015), Kofroň J., Vojnar T., (Eds.), vol. 9548 of *Lecture Notes in Computer Science*, pp. 107–118. doi:10.1007/978-3-319-29817-7_10. 3

[PP15] Papageorgiou A., Platis N.: Triangular mesh simplification on the GPU. *The Visual Computer 31*, 2 (2015), 235–244. doi:10.1007/s00371-014-1039-x. 3

[SHW04] Schaefer S., Hakenberg J., Warren J.: Smooth subdivision of tetrahedral meshes. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry processing* (Nice, France, 2004), SGP' 04, pp. 147–154. doi:10.1145/1057432.1057452. 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. *Comput. Graphics Forum 32*, 1 (2013), 16–26. doi:10.1111/j.1467-8659.2012.03227.x. 10

[WMA*15] Weber D., Mueller-Roemer J. S., Altenhofen C., Stork A., Fellner D.: Deformation simulation using cubic finite elements and efficient p-multigrid methods. *Computers & Graphics 53*, 2 (2015), 185–195. doi:10.1016/j.cag.2015.06.010. 3

[ZSS17] Zayer R., Steinberger M., Seidel H.: A GPU-adapted structure for unstructured grids. *Computer Graphics Forum 36*, 2 (2017), 495–507. doi:10.1111/cgf.13144. 3