DOI: 10.1111/cgf.13581
Pacific Graphics 2018
H. Fu, A. Ghosh, and J. Kopf
(Guest Editors)

*Volume 37 (2018), Number 7*

# GPU-based Polynomial Finite Element Matrix Assembly for Simplex Meshes

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

[1]TU Darmstadt & Fraunhofer IGD, Germany

**Figure 1:** *Left: outer surface of the high-resolution mesh with 1.7 million tetrahedra used in the evaluation of our method. Right: cut through a lower-resolution model to show its inner structure. The models are based on the Airbus flight crew rest compartment (FCRC) bracket, a titanium 3D-printed part developed using simulation and topological optimization (see, e.g., [Kra17]).*

**Abstract**

*In this paper, we present a matrix assembly technique for arbitrary polynomial order finite element simulations on simplex meshes for graphics processing units (GPU). Compared to the current state of the art in GPU-based matrix assembly, we avoid the need for an intermediate sparse matrix and perform assembly directly into the final, GPU-optimized data structure. Thereby, we avoid the resulting* 180% *to* 600% *memory overhead, depending on polynomial order, and associated allocation time, while simplifying the assembly code and using a more compact mesh representation. We compare our method with existing algorithms and demonstrate significant speedups.*

**CCS Concepts**

•***Computing methodologies*** → ***Massively parallel and high-performance simulations;*** *Massively parallel algorithms; Physical simulation; Graphics processors;* •***Mathematics of computing*** → *Combinatoric problems;*

## 1. Introduction

With the continuously high cost of simulating and rendering of computer-generated imagery for movies and television and the recent advances in 3D-printing, physically based simulation and animation are more important than ever. As an example, Fig. 1 shows the geometry of a 3D-printed part developed using simulation and topological optimization, as described by Kranz [Kra17].

One of the most important simulation methods is the finite element method (FEM). Due to the ubiquity of FEM simulations, the constant need for faster simulations, and the price-performance benefits of graphics processing units (GPU), general purpose computing on the GPU (GPGPU) has been applied to them early on (see, e.g., a paper by Liu et al. a year after the first release of CUDA [LJWD08]).

However, the main focus of research on GPU-based FEM has been the efficient solution of the resulting system of equations, not on the assembly of the sparse system matrix, which has come into focus as a significant bottleneck in high performance computing (HPC).

For example, a paper by Guo et al. states that assembly takes 30–40% of total computation time in their use cases [GLG*15]. This overhead is only amplified when a GPU solver is used with a CPU-based assembly approach. Our current solver takes a median time of 14.1 s to set up the initial sparse matrix for the mesh shown on the left in Fig. 1, while solving the corresponding static simulation to convergence only takes 1.5 s. Additionally, mesh loading and CPU-based preprocessing take 15 s and can be GPU accelerated as well (see Section 2.1).

The assembly process can be split into two components:

1. Determination of the sparsity pattern of the system matrix.
2. Summation of the element stiffness matrices into the global system matrix.

During initial setup of a simulation or whenever the mesh changes, for example due to an adaptive method such as the adaptive FEM cloth simulation by Bender and Deul [BD13] or the adaptive, FEM-based brittle fracture simulation by Koschier et al. [KLB14], the sparsity pattern of the matrix must be determined. Cutting or tearing also result in changes to the sparsity pattern. When there are no changes to the topology, only the summation step must be repeated. Three approaches are compared in Sections 4.4 and 5.2. In dynamic simulations using the co-rotational FEM (see, e.g., [MG04]) the summation must be performed once per time step, due to the changes in the rotation matrices. For the nonlinear FEM, multiple updates to the matrix are required while solving the system. The first step is extremely costly when single, linear simulations are performed and when mesh topology is changed frequently.

In computer graphics, simulation meshes are most commonly triangular and tetrahedral meshes, or more generally simplex meshes. Furthermore, tetrahedral mesh generation algorithms, such as the recently published unconditionally robust technique by Hu et al. [HZG*18], are more robust than current hexahedral, or hex-dominant meshing methods.

Next to the element type, choice of element order is important. Linear elements suffer from shear and volume locking, which introduces artificial stiffness into the system (see, e.g., [ISF07]). Furthermore, increasing element order can improve simulation quality at a lower cost than increasing mesh resolution (see, e.g., [WKS*11]).

Therefore, the goal of this work is to answer the following questions:

1. Is it possible to improve GPU system matrix assembly performance and memory use by limiting the input to (higher-order) simplex meshes and making use of the topological properties of simplex meshes?
2. Can assembly be performed efficiently directly into a GPU-optimized sparse matrix data structure?
3. Which summation approach should be chosen for which problem?

In the following, Section 2 outlines the relevant related work. In Section 3, we describe the concepts behind our matrix assembly method. Section 4 provides details on the practical implementation of our approach in CUDA. We list the results of our comparisons with other techniques in Section 5. Lastly, Section 6 sums up this paper and our results, we discuss limitations and we point out potential avenues for future research.

## 2. Related Work

In this section, we outline related work on volumetric mesh data structures, FEM system matrix assembly, and polynomial finite element methods.

### 2.1. Mesh Data Structures

The goal of this work is not to develop a new GPU-based mesh data structure. However, as the choice of mesh data structure significantly influences the implementation of algorithms thereon, we list significant works on GPU-based mesh data structures. Or to paraphrase Fred Brooks, “show me your [data structures], and I won’t usually need your [algorithm]” (cf. [Bro95]).

Very few data structures for (volumetric) mesh representation on the GPU exist that go beyond a simple indexed face table or indexed cell table. However, two recently developed data structures, both based on sparse matrices, influenced this paper significantly.

Zayer et al. presented a GPU-adapted mesh data structure based on a compressed sparse column (CSC) sparse matrix encoding [ZSS17a]. They encode a matrix of local node indices, but omit the value array, instead encoding it into the order of the row indices. For fixed row count matrices, such as the matrix representing the vertices of each cell in each column, they omit the offset array, resulting in an indexed face/cell table. However, the transpose that maps each vertex to the cells it belongs to remains in the partially reduced form. Furthermore, they introduce the concept of “action maps” small matrices that modify the behavior of the sparse matrix-matrix multiplication (SpGEMM) to achieve various effects. Among these is an efficient stiffness matrix assembly method, as detailed in Section 2.2.

A different approach is taken by Mueller-Roemer et al. who encode discrete boundary operators, a concept from algebraic topology (see, e.g., [Hat02]), describing element orientations in ternary compressed sparse row (CSR) matrices [MAS17]. They also omit the value array and encode the sign of the value into the column index. Explicitly encoded zero entries are not allowed. Indirect element relationships are also computed via a modified SpGEMM. This approach is more general, as it does not require defining a fixed element order, thus allowing the representation of arbitrary polyhedra. Coboundary operators describing inverse relationships, as well as the indirect element relationships, are only computed on demand and cached.

### 2.2. System Matrix Assembly

Many works on GPU-based system matrix assembly on the GPU, focus only on the summation step. For example, there is the mesh-coloring-based approach by Komatitsch et al. [KME09]. The method by Weber et al. that uses a CSR-like encoding to store a mapping for each non-zero to the element stiffness matrices that influence it [WBS*13]. They additionally introduce the GPU-optimized Bin-BCSR (binned and blocked CSR) matrix format that

uses the dense $3 \times 3$ blocks in FEM system matrices to improve performance and assembly is performed directly into this structure. Reguly and Giles compare several methods of summation into both CSR and ELLPACK sparse matrices, and a matrix-free approach [RG15].

One of the more comprehensive works, a paper by Cecka et al. [CLD10], compares five different assembly approaches that include determination of the sparsity pattern. However, Zayer et al. have recently published two papers outlining their approach to GPU-based FEM stiffness matrix assembly that achieves even better performance [ZSS17a, ZSS17b]. Their method is based on the highly optimized GPU SpGEMM by Liu and Vinter [LV15]. To achieve high performance, a binning approach is used to choose between one of three different approaches to matrix assembly according to an estimated number of non-zero entries in a row. This leads to a comparatively complex implementation. Furthermore, as the number of non-zero entries is conservatively estimated, a temporary sparse matrix is allocated for assembly before being copied into the final sparse matrix. This leads to a large memory overhead.

In the domain of HPC, very large meshes are used that no longer fit into a single GPU's memory. Therefore the focus in HPC lies primarily on how to best perform the domain decomposition. Thébault et al. [TPD15] present a hybrid approach that combines domain decomposition, divide-and-conquer and mesh coloring for stiffness matrix assembly. Like Zayer et al., we focus on assembling stiffness matrices for small to medium size meshes that fit into GPU memory, as these are more common in simulation for animation and optimization problems. However, combined with a domain decomposition approach, fast assembly on a single compute node is beneficial as well.

### 2.3. Polynomial FEM

In addition to the commonly known linear finite elements, elements with quadratic, cubic, or higher order basis functions can be used (see [ZT00]). As linear elements have one degree of freedom on each vertex, higher order elements provide additional degrees of freedom. While for quadratic basis functions, one additional degree of freedom is added on every edge of the element, cubic elements provide two additional degrees of freedom per edge and one per face. As our matrix assembly method is not only targeted at linear finite elements but at arbitrary polynomial degree FEM, we list several important works here.

The choice of basis functions significantly affects the performance of the solver, as many basis functions require numeric integration to determine the element matrices. Lagrange polynomials are very commonly used and covered in most FEM literature, for example in Zienkiewicz and Taylors book [ZT00]. Similarly common are the 'serendipity' functions, which only add nodes to the edges of the elements. These are also covered in the aforementioned book.

A wide array of other families of basis functions are used, such as the Legendre polynomials and non-uniform rational B-Splines (NURBS) are used as well. Willberg et al. compared several of these polynomials for the use case of simulating Lamb waves [WDV*12].

In simulation for animation, the Bernstein-Bézier polynomials have been used, as the variational integrals can be computed in closed form for constant metric elements (essentially, when undeformed positions are geometrically linear). For example, Weber et al. presented a method for interactive deformation simulation using Bernstein-Bézier polynomials and the co-rotational method [WKS*11]. Furthermore, they observe how the local topology in a tetrahedral mesh can inform the matrix assembly process. However, they do not apply this idea to matrix assembly on the GPU, nor do they generalize it to other element dimension (edge or triangle meshes for example) or other polynomial orders.

Weber et al. later extended their approach to cubic elements and used them for a $p$-multigrid finite element solver that employs a hierarchy not in mesh resolution, but in the polynomial degree $p$ of the elements [WMA*15].

## 3. Concept

In this section, we describe the theoretical concepts behind our matrix assembly method for higher-order simplex meshes. We determine the exact number of nodes that contribute to any individual node's row in the system matrix using only local topological information. No traversal of complex, pointer-based structures or sorting and discarding of duplicate non-zero indices is necessary.

First, let us briefly reiterate how system matrices in the FEM are constructed. As an example, let $\mathbf{K} \in \mathbb{R}^{ne \times ne}$ be the stiffness matrix of a deformation simulation, where $n$ is the number of nodes in the mesh, and $e$ is the embedding or physical dimension, typically 3. The matrix $\mathbf{K}$ is sparse, but consists of locally dense $e \times e$ blocks. Therefore, we can also describe $\mathbf{K}$ as a fourth-order tensor $K_{ijkl} \in \mathbb{R}^{n \times n \times e \times e}$ that is sparse in $ij$. In this notation, we can describe the assembly process as

$$K_{ijkl} = \sum_{T \in \mathbb{T}_{ij}} E_{T,\mathcal{L}_T(i)\mathcal{L}_T(j)kl}, \tag{1}$$

where $\mathbb{T}_{ij} = \{T \mid i \in T \wedge j \in T\}$ is the set of all top-level elements, e.g., tetrahedra, that contain both nodes $i$ and $j$. $\mathcal{L}_T$ is a map from global indices in $[1,n]$ to element-local indices in $[1,m]$, where $m$ is the number of nodes in an element. Finally, $E_{T,ijkl} \in \mathbb{R}^{m \times m \times e \times e}$ is the dense element stiffness tensor of element $T$ (or element stiffness matrix $\mathbf{E}_T \in \mathbb{R}^{me \times me}$). We do not cover the construction of the element stiffness matrices $\mathbf{E}_T$, as it depends on many factors such as the choice of material model and basis functions, for example the commonly used Lagrange polynomials (see, e.g., [ZT00]) or Bernstein-Bézier polynomials (see, e.g., [WMA*15]), and is independent of the assembly process.

As mentioned in the introduction, we limit our approach to simplex meshes to be able to determine the exact number of non-zero entries per row $i$. Simplices are the generalization of triangles and tetrahedra to an arbitrary number of dimensions $k \geqslant 0$. A simplex mesh is a homogeneous or pure simplical complex (cf. [Hat02]), i.e., every $k$-simplex with $k < d$ is part of a $(k+1)$-simplex and therefore of the top-level $d$-simplices. This complex is embedded into an $e$-dimensional space, where $e \geqslant d$.

Every $k$-simplex has $k+1$ vertices or nodes. For order-$p$ ele-

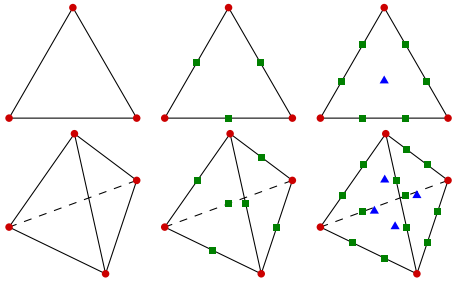


**Figure 2:** *Upper row: Tri3, Tri6, and Tri10 surface elements corresponding to $k = 2$ with $p = 1$, 2, or 3, respectively. Lower row: Tet4, Tet10, and Tet20 elements corresponding to $k = 3$ with $p = 1$, 2, or 3, respectively. Vertex nodes are shown as red circles, edge nodes as green squares, and face nodes as blue triangles.*

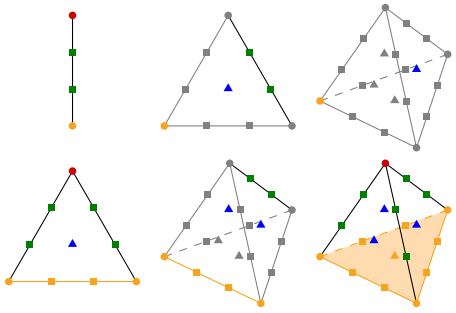


**Figure 3:** *From top-left to bottom-right: edge using a vertex, face using a vertex, cell using a vertex, face using an edge, cell using an edge, cell using a face. The current nodes, edges, and faces are marked in orange. Nodes and edges that are already part of other sub-simplices are grayed out. The nodes with unchanged color (red, green, or blue) are those that introduce new non-zero entries.*

ments, where $p \geqslant 1$, additional nodes are required. An order-$p$ $k$-simplex has

$$\frac{(p+1)^{\overline{k}}}{k!} = \binom{p+k}{k} \tag{2}$$

nodes, where $x^{\overline{n}} = \prod_{k=0}^{n-1}(x+k)$ denotes the rising factorial. For varying $p$, these series of numbers of nodes correspond to the triangular or tetrahedral numbers, depending on $k$, shifted by one. Figure 2 shows complete polynomial triangular and tetrahedral elements of orders one through three. We do not discuss incomplete higher-order elements, such as the cubic Tri9 or Tet16 elements that omit face points (see, e.g., [CGN18]), but adaptation of our method to such elements is straightforward. Furthermore, our focus is on continuous, conforming discretizations which share nodes between cells. In discontinous Galerkin (dG) methods, nodes are unique to each top level element. Therefore, the resulting matrices have a simple, block-sparse structure and are generally easier to assemble. Assembly of dG system matrices is covered in the appendix of Di Pietro and Ern's book on dG methods [DE12].

As shown in Fig. 2, nodes are either only part of a top-level $d$-simplex element or one or more of its $k$-simplex subelements. We call nodes on the interior of a $k$-simplex $k$-simplex nodes, i.e., nodes on a vertex are vertex nodes, nodes on an edge but not on a vertex are edge nodes, and so on. In the following equations, we define the binomial coefficient as

$$\binom{n}{k} = \begin{cases} \dfrac{n!}{k!(n-k)!} & \text{if } 0 \leqslant k \leqslant n \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

for notational simplicity. The number of $k$-simplex nodes on a $k$-simplex is

$$\frac{(p-k)^{\overline{k}}}{k!} = \binom{p-1}{k}. \tag{4}$$

This follows directly from the equivalence to the triangular or tetrahedral figurate numbers, as one "layer" of nodes is removed from all $k+1$ sides of the simplex, as can be seen in Fig. 2 when comparing the number of green edge nodes on an edge or the blue face nodes on a face to all nodes on that edge or face.

To determine the number of other nodes that affect a node, we must examine the local neighborhood of each node. The number of non-zero entries for a node is the number of nodes in all top-level $d$-simplices (cells) that contain the node. However, we cannot simply multiply the result of Eq. (2) with the number of cells that use the node, as most vertices are part of more than one cell. This only provides an upper bound.

For $d = 3$, vertex nodes are part of one 0-simplex, the vertex itself and sets of 1-simplices (edges), 2-simplices (faces), and 3-simplices (cells). Edge nodes are used by the corresponding edge, and sets of faces, and cells. Face nodes lie on the face, and a set of cells. Finally, cell nodes only belong to one cell. The six cases of elements used by other simplices are shown in Fig. 3.

To exactly determine the number of nodes that affect a $k$-simplex node, we use a bottom-up approach. First, we add the number of nodes on the originating $k$-simplex itself, which can be determined according to Eq. (2). Then for every $(k+1)$-simplex that contains the $k$-simplex, we add the number of nodes on it, except for those nodes that are on the initial $k$-simplex. This continues with all $(k+2)$-simplices that contain the node, but now all nodes on $(k+1)$-simplices that use the node of choice must be ignored, as illustrated by the gray nodes in Fig. 3. As in the case of Eq. (4), the number of nodes added by each $l$-simplex with $l \geqslant k$ follows directly from the triangular/tetrahedral numbers, as exactly $l-k$ "layers" of nodes are removed, as can be seen in Fig. 3 by comparing the numbers of red, green and blue nodes to the total nodes on each element:

$$\frac{(p+1-(l-k))^{\overline{l}}}{l!} = \binom{p+k}{l}. \tag{5}$$

Using Eq. (5), the number of nodes affecting a $k$-simplex node $i$ can be computed using

$$n_{k,i} = \sum_{l=k}^{d} \binom{p+k}{l} \left| S_i^{kl} \right|, \tag{6}$$

where $S_i^{kl}$ is the set of $l$-simplices that are or contain the $k$-simplex of node $i$. This allows us to compute the number of non-zeros in

any row of the matrix exactly using minimal topological information. In fact, only the sizes of the sets $S_i^{kl}$ are required, not their contents. Furthermore, the number is equal for all $k$-simplex nodes on a specific $k$-simplex. Additional figures showing examples of Eq. (6) are given in the supplemental material.

## 4. Implementation

In this section, we explain how we used the concepts developed in Section 3 to implement a highly efficient GPU-based system matrix assembly method for arbitrary-order polynomial simplex meshes.

As the performance characteristics of GPUs differ from those of a CPU, we briefly list important considerations and terminology. For more details, refer to the Nvidia CUDA Programming Guide [NVI18].

- CUDA follows a single program multiple data (SPMD) programming model. A *kernel* is executed by a number of threads in parallel. However, the threads are not entirely independent. *Warps*, groups of 32 threads, are scheduled together and closely resemble single instruction multiple data (SIMD) on CPUs. When multiple divergent code paths are taken by threads in a warp, all paths are executed sequentially with masking.
- Threads are additionally grouped into *blocks*. These blocks are assigned to one of several streaming multiprocessors (SMs), the closest equivalent to a core in a multicore CPU.
- Resources such as registers and *shared memory* are limited on the SMs and shared by all active blocks on that SM. Shared memory is as fast as the L1-cache of the SM, but programmable and limited to a small number of kilobytes per block.
- Memory bandwidths, especially on GPUs with high bandwidth memory (HBM), are very high. However, latencies are also large. Ideally, more than one block can be active on an SM for *latency hiding*. Additionally, the high bandwidth is achieved via a very wide memory bus. Therefore, *coalescing* is important, i.e., whenever threads read or write to memory, they should do so in a contiguous block. So if thread 0 reads 4 bytes at address 0x00, thread 1 should read 4 bytes at address 0x04, etc. Supported widths for coalescing are 4, 8, or 16 bytes.
- If more registers than available are needed or when the index into an array cannot be determined by the compiler, register spilling into *local memory* occurs. Local memory is a private, per-thread area in *global memory* (GPU memory) that differs in that stores are cached in the L1-cache. While performance is better than regular global memory access, it is significantly worse than when operations can be performed in registers.

### 4.1. Mesh Data Structure

The mesh data structure we use is based on a combination of the concepts presented by Zayer et al. and Mueller-Roemer et al. [ZSS17a, MAS17]. As we do not require arbitrary polyhedra, but a homogeneous simplical complex, we use a simple array of indexed faces/cells as Zayer et al. do. The orientation of the top-level cells is encoded in the order of the indices in the $n \times (d+1)$ indexed face/cell array. As we only implemented our method for tetrahedral meshes, we call this array `cellVertices`. We store the vertex indices in row-major order, where each cell represents a row, as the four indices fit into a single 16 byte memory transfer. For triangle meshes, storage in column-major order may be preferable to achieve coalescing. Alternatively, a warp-transposed loads can be used that transpose the data in shared memory, this primitive is provided by libraries such as CUB [Mer18]. For the inverse mapping, we apply a matrix transpose method as suggested by Mueller-Roemer et al. As `vertexCells` is irregular, it has an associated CSR/CSC offset array `vertexCellOffsets`. No fixed order is maintained in the transposed matrix, as it would be meaningless, therefore it degenerates to a binary CSR structure, i.e., a CSR matrix without a value array in which presence of an entry describes a 1-value, and 0-values are never explicitly encoded.

For linear FEM, the only additional information required is the $n \times e$ array of positions. Although the embedding dimension $e = 3$ in our case, we do not store `positions` in transposed form for coalescing, as access patterns are mostly random, therefore preventing coalescing. Therefore, we prefer to maintain locality instead. For higher-order simulations, we do not expand `cellVertices` to $n \times \binom{p+d}{d}$ entries. Instead we add additional `cellEdges`/`edgeCells` and `cellFaces`/`faceCell` arrays/-matrices as required. As the number of $k$-simplex nodes on any $k$-simplex is constant and can be determined using Eq. (4), node indices are implicitly derived from the simplex indices. For $p < 3$ the storage requirements are equivalent to explicitly storing node indices per cell as in Zayer et al.'s approach. However, for $p \geqslant 3$ the storage requirements are smaller than explicitly storing all node indices, and independent of polynomial order.

As we use constant metric polynomial tetrahedra, the size of the initial position array `positions` remains unchanged. Furthermore, all additional arrays/matrices can also be computed from `cellVertices` on demand instead of being loaded from a file.

To compute the numbers of non-zero entries in each row of the resulting matrix according to Eq. (6), further information is required. In particular, the number of edges per vertex for $p = 1$ and additionally the number of faces per vertex and faces and cells per edge for $p = 2$ etc. Instead of computing the irregular, sparse matrices `vertexEdges`, `vertexFaces`, and `edgeFaces` (`edgeCells` is computed via transpose), we only determine the regular `edgeVertices`, `faceVertices`, and `faceEdges` arrays. These are easily derived from the `cellVertices`, `cellEdges`, and `cellFaces` arrays by filling a larger array with the sorted tuples of the corresponding indices per edge or face of the cell, then sorting that array of tuples and removing duplicate entries. Sorting and removal of duplicates are standard parallel primitives and efficiently implemented in libraries such as Thrust or CUB [BH15, Mer18]. To determine the counts for Eq. (6), we use atomic additions of the corresponding binomial coefficient into the `columnCounts` array. This atomic count (without scaling) would be necessary for the transposed version anyway, and we do not actually need to know which edges reference a vertex, for example.

### 4.2. Bin-BCSR*

As mentioned in the introduction, we aim to perform our matrix assembly directly into a GPU-optimized sparse matrix structure. As the Bin-BCSR matrix structure by Weber et al. [WBS*13] is also optimized for FEM, we chose it as a basis for our implementation.

**Listing 1:** *Dynamic scheduling for Bin-BCSR. Bin size is chosen to match warp size. Before running,* `counter` *is initialized to 0.*

```
auto tid = threadIdx.x;
auto lid = tid & (WARP_SIZE - 1);

auto bins = rows / WARP_SIZE + (rows % WARP_SIZE != 0);

int bin;
if(lid == 0)
  bin = atomicAdd(counter, 1);
bin = __shfl_sync(0xffffffff, bin, 0, WARP_SIZE);

while(bin < bins)
{
  auto idx = WARP_SIZE * bin + lid;

  if(idx < rows)
  {
    // parallel loop body
  }

  if(lid == 0)
    bin = atomicAdd(counter, 1);
  bin = __shfl_sync(0xffffffff, bin, 0, WARP_SIZE);
}
```

Bin-BCSR groups sets of rows into equally sized bins. Within each bin, the maximum number of non-zero columns is determined. As in the ELLPACK-R format (see [VOFG10]), which does the same for all rows at once, other rows are padded with zeros to match that length and stored in transposed form to achieve coalescing. As done for each row in the commonly used CSR format, offsets are used to mark the position and the size of each bin. The blockwise dense nature of the matrix is exploited by omitting all but the first column index within a row of each dense $e \times e$ block.

However, we made several minor modifications that we collectively refer to as Bin-BCSR*:

1. Instead of only using the implied $e \times e$ block structure along columns, we use it along rows as well and store column indices referring to nodes instead of individual degrees of freedom. This simplifies the assembly, as column indices are not repeated for groups of $e$ rows.
2. Consequently, we store all of the $K_{ij00}$ entries in the block, followed by the $K_{ij01}$ entries, etc. to maintain coalescing. As each thread now processes one $i$ index of $K_{ijkl}$ instead of an $e * i + k$ index into $\mathbf{K}$.
   One major advantage of processing rows of $e \times e$ blocks per thread is that fewer random loads are performed, as otherwise groups of $e$ threads would load the same values.
   Consequently, the separately stored diagonal also stores an $e \times e$ block diagonal.
3. Instead of a bin size of 16 rows, we use a bin size of 32 rows, matching the size of a warp. This allows us to efficiently schedule warps dynamically as shown in Listing 1. This minimizes the effect of different length bins within a block.

Examples of the original Bin-BCSR and improved Bin-BCSR* data structures are shown in Fig. 4. While the example only has padding in the Bin-BCSR* data structure as the bin size matches the block size for Bin-BCSR, the use of 16 row bins and $3 \times 3$ blocks in real-world use of Bin-BCSR also leads to a significant amount of padding. The increased amount of padding due to a larger number of rows per bin in Bin-BCSR* is mostly mitigated by the reduction in size of the column and offset arrays.

Due to the efficient sparse matrix-vector product (SpMV) possible with Bin-BCSR and Bin-BCSR*, these layouts are well-suited for use with iterative solvers such as the conjugate gradient method. The separate storage of the diagonal allows for efficient implementation of (Block-)Jacobi preconditioners, which are highly parallelizable and therefore useful for GPU implementations (see, e.g., [ADFQ17]).

### 4.3. Sparsity Pattern

To determine the sparsity pattern, we first determine `columnCounts` as described in Section 4.1. We then determine the sizes of the bins by iterating over `columnCounts` in parallel and determining the maximum column count per bin, which can be implemented efficiently using warp shuffle instructions (instructions that exchange data between threads in a warp). Afterwards, we compute the cumulative sum of these maxima (minus one multiplied by bin/warp size) using a parallel prefix scan, another standard parallel primitive that is implemented in Thrust, CUB, and other libraries [BH15, Mer18]. These are the `binOffsets`. Finally, the total number of entries can be read back from the end of `binOffsets`. This is then used to allocate the array of non-zero entry values (with $e \times e$ block entries) and column arrays. Additionally the main diagonal is allocated, but this can be done earlier or in parallel, completing the allocation of the final Bin-BCSR* matrix. The overhead w.r.t. a (Block-)CSR matrix due to binning is approximately 30–40% in our experiments (more for the smallest two meshes).

After allocating the matrix structure, the column indices must be determined. We use a dynamically scheduled kernel based on Listing 1. For each row of the block matrix, we first determine which type of $k$-simplex the row/node belongs to, e.g., for $p = 2$ it is a vertex node if `idx` $<$ `numVertices` and an edge node otherwise. Depending on the node type, we iterate over `vertexCells` or `edgeCells` for that particular index. Every node in that cell (determined via the `cell`$\star$ arrays) is then added into a sorted array of column indices by first performing a binary search, then shifting back existing entries if a new value is inserted. Entries on the main diagonal are skipped, due to the separate storage in Bin-BCSR*.

To do so efficiently, we reserve 63 indices per thread in shared memory for the column array. Each block has 64 threads or two warps, so this corresponds to 15.75 KiB of shared memory per block. While $\leqslant 63$ columns are needed, all insertions are performed in shared memory. Upon insertion of the $64^{\text{th}}$ element or after all cells have been processed for that row, the columns are copied into the Bin-BCSR* structure. Further insertions, if any, are performed in global memory. In our experiments, only a negligible fraction of nodes had more than 63 neighbors.

### 4.4. Summation

Having determined the sparsity pattern, the last remaining step is the calculation and summation of the element stiffness matrices. At

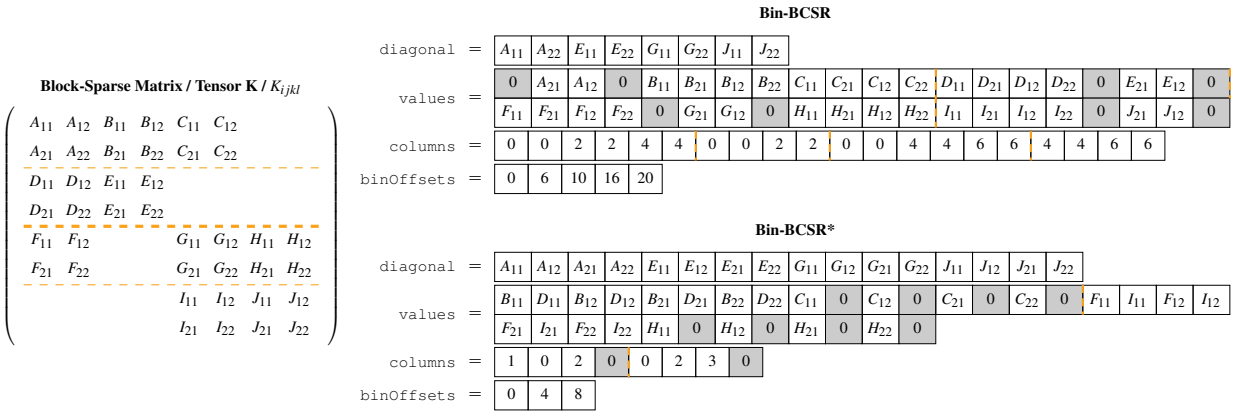



**Figure 4:** *Illustration of the differences between Bin-BCSR and Bin-BCSR* for an embedding dimension $e = 2$ / block size $2 \times 2$ and a bin size of 2. The dashed lines in the matrix mark which rows are combined into a bin with Bin-BCSR. The thick dashed line marks which rows are combined into a bin with Bin-BCSR*. The borders between bins are also indicated in the value and column arrays of the data structures. Padding and explicitly encoded 0-values are shown with a gray background. The column indices in the Bin-BCSR* structure are $j$-indices into the sparse tensor $K_{ijkl}$, while the column indices in the Bin-BCSR structure denote columns of the sparse matrix* **K**.

this point, the choice of material model, strain, and basis functions becomes relevant.

Due to the simplicity of implementation, we use a linear, homogenous, isotropic material model, described using Lamé parameters and the stress-strain relationship

$$\sigma = 2\mu\varepsilon + \lambda \mathrm{tr}(\varepsilon)\mathbf{I}, \tag{7}$$

where $\varepsilon$ is the linear strain tensor, $\lambda$ and $\mu$ are Lamé's first and second parameters, respectively, and $\sigma$ is the resulting stress.

Furthermore, we use the Bernstein-Bézier basis functions, due to the closed form integration. As shown by Weber et al. [WMA*15], computing the stiffness matrices becomes a simple weighted sum of outer products $\mathbf{b}_i \mathbf{b}_j^T$, where

$$\mathbf{b}_i = \left( \frac{\partial \xi_i}{\partial x}, \frac{\partial \xi_i}{\partial y}, \frac{\partial \xi_i}{\partial z} \right)^T \tag{8}$$

are the gradients of the barycentric coordinates $\xi_i$ associated with the local vertex nodes $i$.

We implemented three approaches to summation for comparison. The first approach uses a per node map of nodes to cells, i.e., the same `*Cells` matrices used during pattern computation. The kernel is also very similar and uses the same shared memory cache, except all column indices are either loaded at the beginning if the row has $\leqslant 63$ non-zero entries, or not at all. These loads are coalesced due to the transposed storage of column indices in Bin-BCSR*. For each node in each neighboring cell a binary lookup is performed to find the correct position, and the corresponding element stiffness matrix is loaded from global memory and added in global memory to the corresponding entry. The diagonal is calculated entirely in registers.

The inline approach is exactly the same, except that the element stiffness matrices are computed within the summation kernel. To reduce branch divergence and kernel size, the cell's vertex indices are reordered on the fly to move the current node to the beginning, while maintaining orientation by ensuring an even number of index swaps. As reordering becomes more complicated for higher orders, and register pressure was already very high, we only implemented this method for $p = 1$.

The final approach using per entry maps requires a large amount of additional data. We create an $n \times m$ sparse matrix with $k$ non-zero column-only entries, where $n$ is the number of non-zero entries in the system matrix, $m$ is the number of $e \times e$ element stiffness sub-matrices and $k = \binom{p+d}{d}^2 \cdot |\mathbb{T}|$. This matrix stores the list of all element stiffness matrices that must be added for each entry. As each $e \times e$ entry can be processed in sequence, summation can be performed in registers, which should lead to performance improvements.

## 5. Results

Throughout all measurements, we used a 64-bit Windows 10 computer with an Intel Core i7-6700K CPU (4 cores, 4 GHz base clock), 16 GiB of DDR4-2133 main memory (34 GB/s) and an Nvidia Quadro GP-100 GPU (3584 CUDA cores, 1.3 GHz base clock) with 16 GiB of HBM2 GPU memory (717 GB/s). All experiments were repeated 1000 times and results are given as the median value.

For the comparisons, we created 11 tetrahedral meshes with varying resolution of the model shown in Fig. 1. Table 1 lists the sizes of all meshes in vertices, edges, faces, and tetrahedra. Additionally, it lists the exact number of non-zero entries and the upper bound as calculated when using Zayer et al.'s approach based on Liu and Vinter's GPU spGEMM [LV15, ZSS17a, ZSS17b]. Not only is the allocation of a temporary sparse matrix required, but it is 4.2–6.0× the size of the final matrix for $p = 1$ (an overhead of up to 600%), 2.1–2.5× the size for $p = 2$ (an overhead of up to 250%), and 1.6–1.8× the size for $p = 3$ (an overhead of up to 180%).

**Table 1:** *List of the sizes of the meshes used in the evaluation in numbers of vertices, edges, faces and cells. Additionally, the exact number of non-zero entries, the upper bounds resulting from Zayer et al.'s method, and the ratio between the two are given for the polynomial orders $p = 1$, 2, and 3.*

| Vertices | Edges | Faces | Cells | Order 1 | | | Order 2 | | | Order 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Exact | Bound | Ratio | Exact | Bound | Ratio | Exact | Bound | Ratio |
| 2.9 k | 13.5 k | 18.3 k | 7.7 k | 29.8 k | 124.0 k | 4.2 | 363.9 k | 774.9 k | 2.1 | 1.9 M | 3.1 M | 1.6 |
| 11.7 k | 64.2 k | 96.7 k | 44.2 k | 140.0 k | 707.7 k | 5.1 | 1.9 M | 4.4 M | 2.3 | 10.3 M | 17.7 M | 1.7 |
| 21.7 k | 125.5 k | 194.3 k | 90.6 k | 272.8 k | 1.4 M | 5.3 | 3.8 M | 9.1 M | 2.4 | 20.8 M | 36.2 M | 1.7 |
| 30.8 k | 181.4 k | 283.9 k | 133.3 k | 393.7 k | 2.1 M | 5.4 | 5.5 M | 13.3 M | 2.4 | 30.4 M | 53.3 M | 1.8 |
| 47.1 k | 284.2 k | 450.5 k | 213.4 k | 615.6 k | 3.4 M | 5.5 | 8.7 M | 21.3 M | 2.4 | 48.4 M | 85.4 M | 1.8 |
| 59.7 k | 364.1 k | 580.7 k | 276.3 k | 787.9 k | 4.4 M | 5.6 | 11.2 M | 27.6 M | 2.5 | 62.5 M | 110.5 M | 1.8 |
| 77.6 k | 479.4 k | 769.4 k | 367.6 k | 1.0 M | 5.9 M | 5.7 | 14.9 M | 36.8 M | 2.5 | 82.9 M | 147.0 M | 1.8 |
| 104.3 k | 652.4 k | 1.1 M | 505.8 k | 1.4 M | 8.1 M | 5.7 | 20.4 M | 50.6 M | 2.5 | 113.7 M | 202.3 M | 1.8 |
| 144.8 k | 917.5 k | 1.5 M | 719.1 k | 2.0 M | 11.5 M | 5.8 | 28.8 M | 71.9 M | 2.5 | 161.2 M | 287.6 M | 1.8 |
| 210.5 k | 1.4 M | 2.2 M | 1.1 M | 2.9 M | 17.1 M | 5.9 | 42.7 M | 107.1 M | 2.5 | 239.4 M | 428.5 M | 1.8 |
| 326.2 k | 2.1 M | 3.5 M | 1.7 M | 4.6 M | 27.2 M | 6.0 | 67.4 M | 170.3 M | 2.5 | 329.3 M | 681.1 M | 1.8 |

### 5.1. Assembly

First, we compare the runtimes of the complete matrix assembly, including determination of the sparse pattern and element stiffness calculation and summation, with Zayer et al. For this comparison, we use the summation method with per simplex maps, as they do not require any information beyond the `vertexCells` and `edgeCells` sparse maps which are also required for the determination of column indices in each row. Furthermore, unlike the inlined method, which requires the same information, it is implemented for $p = 1$, 2, and 3. We discuss the three summation methods in detail in Section 5.2.

Figure 5 shows the times taken by the assembly process, including allocation and calculation (included in "Summation" which is discussed in more detail in Section 5.2) of the element stiffness matrices and the allocation of the resulting matrix, for $p = 1$, 2, and 3. The results of the individual measurements are also given in Table 2. The numbers for Zayer et al.'s method are taken from Figure 5 of their paper [ZSS17b] and also given below:

| **Tetrahedra** | 250 k | 500 k | 800 k | 1 M |
|---|---|---|---|---|
| **Order 1** | 82.3 | 185 | 287 | 402 |
| **Order 2** | 623 | 1642 | 2732 | 3584 |

In particular, we used the timings given for SuiteSparse Sparse2 (see [Dav18]) in their paper, divided by the speedup given for their method on the GPU without reordering, as their numbers do not include the time taken for reordering or any other preprocessing steps.

A direct comparison is difficult, as Zayer et al. used a different set of hardware using an Nvidia Tesla K40m (2880 CUDA cores, 745 MHz base clock) with 12 GiB of GDDR5 memory (288 GB/s). Therefore, we do not list any exact speedups. However, they use purely tetrahedral meshes for their evaluation as well, and the sizes of the meshes are on the same order of magnitude. The memory bandwidth of our GPU is 2.5× higher than the one used in their experiments. The theoretical peak single precision compute performance of the Quadro GP100 (10.3 TFLOPS) is similarly 2.4× larger than for the Tesla K40m (4.29 TFLOPS). Even when a factor of 2.5× is taken into account, a significant speedup remains.

Furthermore, the allocation of the final sparse matrix takes slightly more than 20% of the assembly time, both for $p = 1$ and 2, and only slightly less for $p = 3$. Since the allocation time is linear in the number of bytes allocated, as can be seen in the runtime measurements, introducing an additional 6×, 2.5×, or 1.8× oversized allocation for a temporary matrix would increase the runtime by more than 120%, 50%, or 32%, respectively, due to allocation alone.

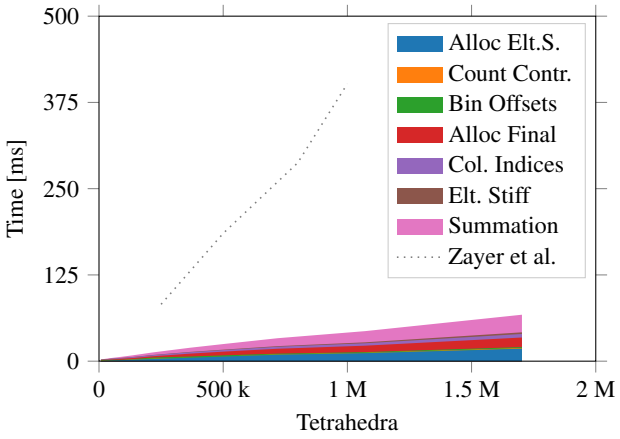
In summary, by restricting our assembly method to simplex meshes, we were able to remove the 600%, 250%, or 180% memory overhead for $p = 1$, 2, or 3, respectively, caused by the temporary sparse matrix. Furthermore, we were able to achieve a significant speedup compared to the state of the art.

### 5.2. Summation

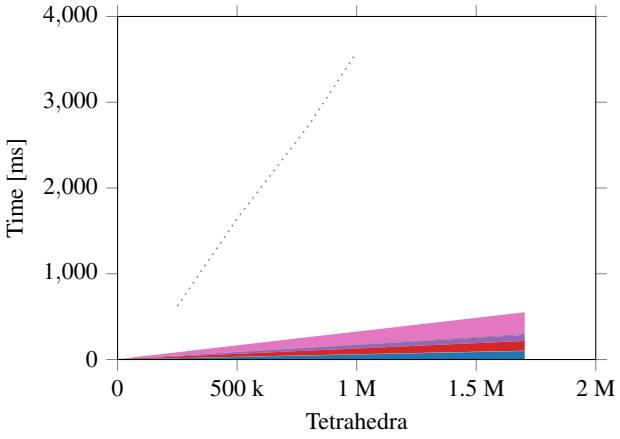
In Section 4, we described three different approaches to summation that differ significantly in the amount of memory they require. Figure 6 compares the runtimes of the three summation approaches, including calculation of the element stiffness matrices. The measurements are also given in Table 3.

The method using per entry maps is 1.8–2.1× faster for $p = 1$ and 2.4–2.6× faster for $p = 2$ than when using per simplex maps (`vertexCells` and `edgeCells`). However, it requires a significantly larger amount of memory and fails for the two largest meshes when $p = 2$. For $p = 3$, only the per simplex approach has been implemented and is therefore not included in this section. The kernel with inlined computation of the element stiffness matrices (and per simplex maps), which was only implemented for $p = 1$, requires the least amount of memory and does not require allocation of the $e^2 \cdot \frac{n \cdot (n+1)}{2} \cdot |\mathbb{T}|$ floats for the element stiffness matrices, where $n$ is the number of nodes per $d$-simplex (tetrahedron), and $\mathbb{T}$ is the set of all top level $d$-simplices. However, it is slightly slower than the per simplex map implementation due to the high register pressure and increased compute load.

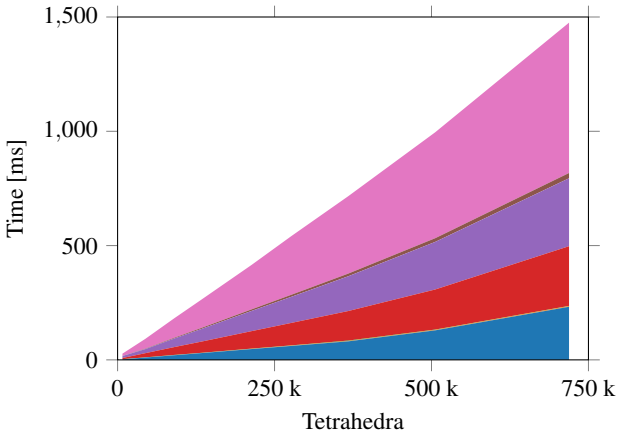
Which summation method should be chosen strongly depends on the use case. The method with per simplex maps is the easiest to implement, as it requires no preprocessing beyond what is required for assembly itself, making it a good choice for simulations with




**(a)** $p = 1$

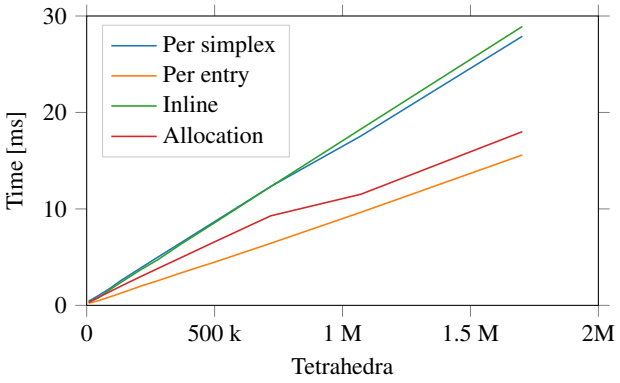



**(b)** $p = 2$

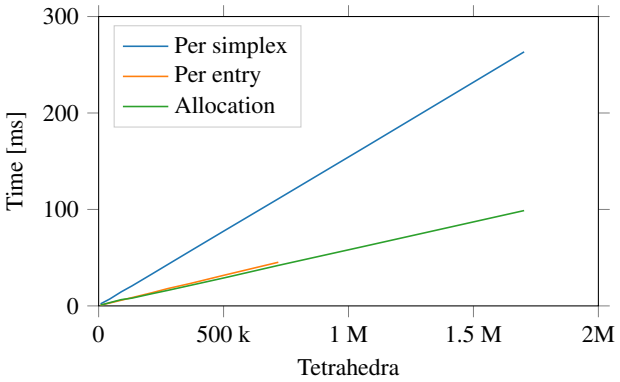



**(c)** $p = 3$

**Figure 5:** *Runtime of the entire assembly process over number of tetrahedra for $p = 1$, 2, and 3. Our method is shown split into the individual steps, where the colored areas represent the runtime of each step. The total height corresponds to the total runtime. The numbers for Zayer et al.'s method are taken from Figure 5 in their paper [ZSS17b].*




**(a)** $p = 1$




**(b)** $p = 2$

**Figure 6:** *Comparison of the runtimes of the three summation approaches for $p = 1$ and the two map-based approaches for $p = 2$. The methods with per node or per entry summation maps additionally require allocation of the element stiffness matrix array as a one-time cost (shown separately). For $p = 2$, the per entry map method failed for the two largest meshes due to allocation failures.*

adaptive meshes. Furthermore, it separates assembly and material model completely and makes implementation of complex material models possible.

The method with inline computation of the element stiffness matrices requires the same per simplex maps. However, even though many memory reads are avoided, it is slightly slower than the per simplex method. Furthermore, a very large number of registers are required, even for homogeneous, isotropic materials and linear basis functions. For more complex material models or higher-order basis functions, register spilling is effectively unavoidable, leading to even slower calculations. Despite these limitations, using the inline method may be worthwhile for static, linear simulations, as no time is needed to allocate the element stiffness matrix arrays, a smaller amount of memory is required, and the system is typically only assembled once.

Finally, the per entry map approach is up to 2.6× faster than the approach with per simplex maps. Like the per simplex method, it separates assembly and material model completely. Therefore,

it should be the first choice for dynamic simulations using co-rotational strains as well as static or dynamic simulations using nonlinear material models or nonlinear strains, since both require frequent updates of the system matrix without changing the sparsity pattern. The large speedup is to be expected, as summation of the $e \times e$ blocks of the matrix can be performed in registers, unlike the other methods where summation takes place in memory. However, the memory overhead is significant and additional preprocessing is required.

## 6. Conclusion

In conclusion, we have shown that by restricting our method to simplex meshes (triangular and tetrahedral meshes), we are able to derive combinatorial equations that allow for exact allocation of the resulting sparse matrix using minimal topological information.

This allows us to perform assembly directly into a GPU-optimized sparse matrix structure based on Bin-BCSR by Weber et al. [WBS*13], while avoiding the 180% to 600% memory overhead, depending on polynomial order, and allocation time for the temporary matrix inherent to the current state of the art approach by Zayer et al. [ZSS17a, ZSS17b]. Due to the reduced memory requirements, we can simulate significantly larger meshes.

Furthermore, we achieve a significant speedup compared to the state of the art. At the same time, our approach is easier to implement, as no sophisticated binning approach to choose between several algorithms, such as the one introduced by Liu and Vinter [LV15] and used by Zayer et al. [ZSS17a, ZSS17b], is necessary to achieve these speedups.

Additionally, we presented a modification to Bin-BCSR, Bin-BCSR*, that uses the $e \times e$ block structure of the matrix along both dimensions to further reduce memory size and improve locality. By matching the bin size to the GPU's warp size, a simple, per-warp dynamic scheduling approach (see Section 4) can be used while processing the Bin-BCSR* matrix. This scheduling approach is used during matrix assembly and can be used to improve the speed of the sparse matrix-vector product (SpMV) as well.

Finally, we compared three approaches to the summation step that is required whenever material parameters change, on every frame in co-rotational FEM, or multiple times per step in nonlinear FEM. We provide the reader with a basis for deciding which method to choose, depending on the use case, in Section 5.2.

### 6.1. Limitations

As a direct result of the restriction to simplex meshes, mixed-element and hexahedral meshes can not be used with our method. For quadrilateral or hexahedral meshes, similar equations to the ones we presented can be derived. However, such meshes are frequently also structured. In such cases, the multidiagonal structure makes assembly trivial. For mixed-element meshes, a more general method, such as the one presented by Zayer et al. [ZSS17a, ZSS17b], should be used.

Furthermore, while our method should be well suited to adaptive meshes ($h$-adaptivity) due to its speed and memory efficiency, adaptive polynomial degrees per element ($p$-adaptivity) can not be used. An overview of $h$, $p$, and $h$-$p$ FEM methods can be found in Babuška and Guo's paper [BG92]. To our knowledge, $p$-adaptive FEM is currently fairly uncommon in computer graphics, but is covered in detail in Grinspun's PhD thesis [Gri03].

### 6.2. Future Work

With respect to future work, there are several ways in which our method could be extended or applied to other problems.

In Section 3, we noted that higher-order elements that omit cell nodes (for $p \geqslant 4$) or cell and face nodes (for $p \geqslant 3$) are also used in practice, next to the complete higher-order elements. Extension to such elements should be straightforward. For purely quadrilateral or hexahedral meshes, or their $n$-dimensional extension, similar equations can be derived as well. However, a different approach, such as extracting regular parts of the mesh and treating those as regular grids, is likely to be even more efficient.

Additionally, while we avoided a binning approach to maintain a simple implementation, further performance gains may be possible, as registers and shared memory could be used more effectively. However, potential gains are limited by the allocation times which currently amount to up to 47% of the assembly time.

For adaptive simulations, local updates of the matrix may be faster than recreating the entire matrix. However, our approach is orthogonal to local updates, as local updates can benefit from exact allocation as well. Combining the two is a potential area for future research.

An interesting future area of application for our approach is co-dimensional simulation on simplical complexes, as pioneered by Zhu et al. [ZQC*14, ZLQF15]. Although we only consider homogeneous simplical complexes in this paper, Eq. (6) applies to inhomogeneous simplical complexes that have a mix of lines, edges, triangles, and tetrahedra as top level elements as well. Several steps of the assembly process would require significant modifications. For example, in a co-dimensional setting triangles have element matrices as well that must be treated differently.

## Acknowledgments

This work was supported by the European Union project CloudiFacturing which is co-funded by the Horizon 2020 Framework of the European Union under Grant No. 768892, and the Fraunhofer-internal program "Market-oriented strategic preliminary research" under Grant No. MAVO 828424. We thank Airbus Operations GmbH for the permission to use the FCRC bracket model in our evaluation.

## References

[ADFQ17] ANZT H., DONGARRA J., FLEGAR G., QUINTANA-ORTÍ E. S.: Batched Gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores* (2017), PMAM '17, pp. 1–10. doi:10.1145/3026937.3026940. 6

**Table 2:** *Times shown in Fig. 5 for the individual stages of the assembly process with $p = 1$, 2, and 3, as well as the total time taken. All times are in milliseconds.*

**(a)** $p = 1$

| Tetrahedra | 7.7 k | 44.2 k | 90.6 k | 133.3 k | 213.4 k | 276.3 k | 367.6 k | 505.8 k | 719.1 k | 1.1 M | 1.7 M |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Alloc Elt.S.** | 0.285 | 0.812 | 1.41 | 1.99 | 3.01 | 3.8 | 4.93 | 6.66 | 9.28 | 11.5 | 18 |
| **Count Contr.** | 0.644 | 0.475 | 0.0894 | 0.491 | 0.479 | 0.481 | 0.135 | 0.15 | 0.544 | 0.555 | 1.01 |
| **Bin Offsets** | 0.961 | 0.696 | 0.515 | 0.458 | 1.63 | 1.64 | 1.69 | 1.7 | 1.08 | 1.06 | 1.43 |
| **Alloc Final** | 0.0169 | 0.466 | 1.62 | 1.57 | 1.94 | 2.68 | 4.19 | 5.41 | 7.1 | 9.32 | 13.8 |
| **Col. Indices** | 0.172 | 0.921 | 0.977 | 1.28 | 1.53 | 1.82 | 2.08 | 2.33 | 2.95 | 3.53 | 5.13 |
| **Elt. Stiff** | 0.0649 | 0.105 | 0.142 | 0.191 | 0.266 | 0.33 | 0.426 | 0.585 | 0.837 | 1.34 | 2.4 |
| **Summation** | 0.365 | 0.886 | 1.61 | 2.38 | 3.65 | 4.64 | 6.06 | 8.19 | 11.5 | 16.2 | 25.5 |
| **Total** | 2.51 | 4.36 | 6.36 | 8.36 | 12.5 | 15.4 | 19.5 | 25 | 33.3 | 43.5 | 67.3 |

**(b)** $p = 2$

| Tetrahedra | 7.7 k | 44.2 k | 90.6 k | 133.3 k | 213.4 k | 276.3 k | 367.6 k | 505.8 k | 719.1 k | 1.1 M | 1.7 M |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Alloc Elt.S.** | 0.772 | 3.36 | 6.49 | 8.04 | 12.6 | 16.1 | 21.3 | 29.3 | 42 | 62.2 | 98.8 |
| **Count Contr.** | 0.0731 | 0.12 | 0.151 | 0.162 | 0.925 | 1.02 | 1.01 | 1.44 | 1.78 | 2.32 | 3.44 |
| **Bin Offsets** | 0.0933 | 0.678 | 0.743 | 0.47 | 0.713 | 0.691 | 0.957 | 1.04 | 1.12 | 0.886 | 0.158 |
| **Alloc Final** | 1.23 | 4.86 | 8.84 | 10.7 | 16.3 | 20.4 | 26.6 | 35.6 | 49.5 | 72.3 | 114 |
| **Col. Indices** | 1.51 | 2.51 | 4.81 | 6.01 | 9.39 | 11.9 | 16.2 | 21.7 | 30.6 | 45.2 | 71.3 |
| **Elt. Stiff** | 0.132 | 0.313 | 0.511 | 0.686 | 1.11 | 1.42 | 1.9 | 2.61 | 3.63 | 5.49 | 8.87 |
| **Summation** | 1.98 | 6.78 | 14.1 | 20 | 32 | 41.4 | 55.1 | 75.7 | 107 | 160 | 254 |
| **Total** | 5.79 | 18.6 | 35.6 | 46.1 | 73 | 93 | 123 | 167 | 236 | 348 | 551 |

**(c)** $p = 3$

| Tetrahedra | 7.7 k | 44.2 k | 90.6 k | 133.3 k | 213.4 k | 276.3 k | 367.6 k | 505.8 k | 719.1 k |
|---|---|---|---|---|---|---|---|---|---|
| **Alloc Elt.S.** | 2.45 | 10.2 | 20.2 | 29.6 | 47.3 | 60.9 | 81.1 | 129 | 232 |
| **Count Contr.** | 0.166 | 0.181 | 0.579 | 0.581 | 1.28 | 1.53 | 1.76 | 2.3 | 2.96 |
| **Bin Offsets** | 0.574 | 0.557 | 0.949 | 0.604 | 0.738 | 0.813 | 1.68 | 1.13 | 0.93 |
| **Alloc Final** | 4.91 | 18.3 | 34.3 | 49 | 76.4 | 98.1 | 129 | 175 | 261 |
| **Col. Indices** | 8.82 | 17.9 | 37.8 | 55.3 | 88.5 | 114 | 152 | 208 | 297 |
| **Elt. Stiff** | 0.394 | 1.74 | 3.12 | 4.55 | 6.76 | 8.97 | 11.6 | 16.2 | 23 |
| **Summation** | 9.92 | 43.1 | 85.5 | 124 | 194 | 256 | 339 | 463 | 658 |
| **Total** | 27.2 | 92 | 183 | 263 | 415 | 541 | 716 | 996 | 1480 |

**Table 3:** *Runtime measurements of the three summation methods as well as the time for element stiffness matrix array allocation shown in Fig. 6. All times are given in milliseconds.*

| Tetrahedra | Order 1 | | | | Order 2 | | |
|---|---|---|---|---|---|---|---|
| | Simplex | Entry | Inline | Alloc | Simplex | Entry | Alloc |
| 7.7 k | 0.43 | 0.226 | 0.387 | 0.285 | 2.11 | 0.82 | 0.772 |
| 44.2 k | 0.991 | 0.473 | 0.761 | 0.812 | 7.1 | 3 | 3.36 |
| 90.6 k | 1.75 | 0.876 | 1.61 | 1.41 | 14.6 | 5.88 | 6.49 |
| 133.3 k | 2.57 | 1.26 | 2.37 | 1.99 | 20.7 | 8.56 | 8.04 |
| 213.4 k | 3.91 | 2.01 | 3.73 | 3.01 | 33.1 | 13.7 | 12.6 |
| 276.3 k | 4.97 | 2.54 | 4.7 | 3.8 | 42.8 | 17.9 | 16.1 |
| 367.6 k | 6.48 | 3.36 | 6.33 | 4.93 | 57 | 23.3 | 21.3 |
| 505.8 k | 8.77 | 4.53 | 8.66 | 6.66 | 78.3 | 32 | 29.3 |
| 719.1 k | 12.3 | 6.43 | 12.3 | 9.28 | 111 | 45.2 | 42 |
| 1.1 M | 17.6 | 9.65 | 18.3 | 11.5 | 165 | — | 62.2 |
| 1.7 M | 27.9 | 15.6 | 28.9 | 18 | 263 | — | 98.8 |

[BD13] BENDER J., DEUL C.: Adaptive cloth simulation using corotational finite elements. *Computers & Graphics 37*, 7 (2013), 820–829. doi:10.1016/j.cag.2013.04.008. 2

[BG92] BABUŠKA I., GUO B. Q.: The *h*, *p* and *h-p* version of the finite element method: Basis theory and applications. *Advances in Engineering Software 15*, 3–4 (1992), 159–174. doi:10.1016/0965-9978(92)90097-Y. 10

[BH15] BELL N., HOBEROCK J.: Thrust 1.8.1, Mar. 2015. URL: https://thrust.github.io/. 5, 6

[Bro95] BROOKS JR. F. P.: *The Mythical Man-month*, anniversary ed. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-83595-9. 2

[CGN18] CGNS STEERING COMMITTEE: CGNS standard interface data structures 3.3 (rev 2), 2018. Retrieved 2018-06-08. URL: https://cgns.github.io/CGNS_docs_current/sids/sids.pdf. 4

[CLD10] CECKA C., LEW A. J., DARVE E.: Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering 85*, 5 (2010), 640–669. doi:10.1002/nme.2989. 3

[Dav18] DAVIS T.: SuiteSparse: A suite of sparse matrix packages, 2018. Retrieved 2018-06-08. URL: http://faculty.cse.tamu.edu/davis/suitesparse.html. 8

[DE12] DI PIETRO D. A., ERN A.: *Mathematical Aspects of Discontinuous Galerkin Methods*. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-22980-0. 4

[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. doi:10.1016/j.compfluid.2014.09.007. 2

[Gri03] GRINSPUN E.: *The Basis Refinement Method*. PhD thesis, California Institute of Technology, 2003. URL: https://thesis.library.caltech.edu/2324/. 10

[Hat02] HATCHER A.: *Algebraic Topology*. Cambridge University Pr., 2002. ISBN 0-521-79540-0. 2, 3

[HZG*18] HU Y., ZHOU Q., GAO X., JACOBSON A., ZORIN D., PANOZZO D.: Tetrahedral meshing in the wild. *ACM Transactions on Graphics 37*, 4 (2018), 60:1–60:14. doi:10.1145/3197517.3201353. 2

[ISF07] IRVING G., SCHROEDER C., FEDKIW R.: Volume conserving finite element simulations of deformable models. *ACM Transactions on Graphics 26*, 3 (2007), 13:1–13:6. doi:10.1145/1276377.1276394. 2

[KLB14] KOSCHIER D., LIPPONER S., BENDER J.: Adaptive tetrahedral meshes for brittle fracture simulation. In *Proceedings of the 2014 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2014), SCA '14, pp. 57–66. 2

[KME09] KOMATITSCH D., MICHÉA D., ERLEBACHER G.: Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing 69*, 5 (2009), 451–460. doi:10.1016/j.jpdc.2009.01.006. 2

[Kra17] KRANZ J.: *Methodik und Richtlinien für die Konstruktion von laseradditiv gefertigten Leichtbaustrukturen*. Light Engineering für die Praxis. Springer Vieweg, 2017. doi:10.1007/978-3-662-55339-8. 1

[LJWD08] LIU Y., JIAO S., WU W., DE S.: GPU accelerated fast FEM deformation simulation. In *2008 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)* (2008), pp. 606–609. doi:10.1109/APCCAS.2008.4746096. 1

[LV15] LIU W., VINTER B.: A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. *Journal of Parallel and Distributed Computing 85* (2015), 47–61. doi:10.1016/j.jpdc.2015.06.010. 3, 7, 10

[MAS17] MUELLER-ROEMER J. S., ALTENHOFEN C., STORK A.: Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. *Computer Graphics Forum 36*, 5 (2017), 59–69. doi:10.1111/cgf.13245. 2, 5

[Mer18] MERILL D.: CUB 1.8.0, Feb. 2018. URL: https://nvlabs.github.io/cub/. 5, 6

[MG04] MÜLLER M., GROSS M.: Interactive virtual materials. In *Proceedings of Graphics Interface 2004* (2004), GI '04, pp. 239–246. 2

[NVI18] NVIDIA CORPORATION: CUDA C programming guide, May 2018. Version 9.2. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. 5

[RG15] REGULY I. Z., GILES M. B.: Finite element algorithms and data structures on graphical processing units. *International Journal of Parallel Programming 43*, 2 (2015), 203–239. doi:10.1007/s10766-013-0301-6. 3

[TPD15] THÉBAULT L., PETIT E., DINH Q.: Scalable and efficient implementation of 3D unstructured meshes computation: A case study on matrix assembly. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2015), PPoPP 2015, pp. 120–129. doi:10.1145/2688500.2688517. 3

[VOFG10] VÁZQUEZ F., ORTEGA G., FERNÁNDEZ J. J., GARZÓN E. M.: Improving the performance of the sparse matrix vector product with GPUs. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology* (Bradford, United Kingdom, 2010), CIT '10, pp. 1146–1151. doi:10.1109/CIT.2010.208. 6

[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. 2, 5, 10

[WDV*12] WILLBERG C., DUCZEK S., VIVAR PEREZ J. M., SCHMICKER D., GABBERT U.: Comparison of different higher order finite element schemes for the simulation of Lamb waves. *Computer Methods in Applied Mechanics and Engineering 241–244* (2012), 246–261. doi:10.1016/j.cma.2012.06.011. 3

[WKS*11] WEBER D., KALBE T., STORK A., FELLNER D., GOESELE M.: Interactive deformable models with quadratic bases in Bernstein-Bézier-form. *The Visual Computer 27*, 6 (2011), 473–483. doi:10.1007/s00371-011-0579-6. 2, 3

[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, 7

[ZLQF15] ZHU B., LEE M., QUIGLEY E., FEDKIW R.: Codimensional non-Newtonian fluids. *ACM Transactions on Graphics 34*, 4 (2015), 115:1–115:9. doi:10.1145/2766981. 10

[ZQC*14] ZHU B., QUIGLEY E., CONG M., SOLOMON J., FEDKIW R.: Codimensional surface tension flow on simplicial complexes. *ACM Transactions on Graphics 33*, 4 (2014), 111:1–111:11. doi:10.1145/2601097.2601201. 10

[ZSS17a] 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. 2, 3, 5, 7, 10

[ZSS17b] ZAYER R., STEINBERGER M., SEIDEL H.: Sparse matrix assembly on the GPU through multiplication patterns. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)* (2017), pp. 1–8. doi:10.1109/HPEC.2017.8091057. 3, 7, 8, 9, 10

[ZT00] ZIENKIEWICZ O. C., TAYLOR R. L.: *Finite Element Method: Volume 1*, fifth ed. Butterworth-Heinemann, 2000. ISBN 0-750-65049-4. 3