RXMesh: A GPU Mesh Data Structure

AHMED H. MAHMOUD, University of California, Davis, USA and Autodesk Research, Canada
SERBAN D. PORUMBESCU, University of California, Davis, USA
JOHN D. OWENS, University of California, Davis, USA

We propose a new static high-performance mesh data structure for triangle surface meshes on the GPU. Our data structure is carefully designed for parallel execution while capturing mesh locality and confining data access, as much as possible, within the GPU's fast "shared memory." We achieve this by subdividing the mesh into *patches* and representing these patches compactly using a matrix-based representation. Our patching technique is decorated with *ribbons*, thin mesh strips around patches that eliminate the need to communicate between different computation thread blocks, resulting in consistent high throughput. We call our data structure *RXMesh*: Ribbon-matriX Mesh. We hide the complexity of our data structure behind a flexible but powerful programming model that helps deliver high performance by inducing load balance even in highly irregular input meshes. We show the efficacy of our programming model on common geometry processing applications—mesh smoothing and filtering, geodesic distance, and vertex normal computation. For evaluation, we benchmark our data structure against well-optimized GPU and (single and multi-core) CPU data structures and show significant speedups.

CCS Concepts: • **Computing methodologies → Massively parallel algorithms**; *Mesh geometry models.*

Additional Key Words and Phrases: mesh, data structure, GPU, parallel

ACM Reference Format:
Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Trans. Graph.* 40, 4, Article 104 (August 2021), 16 pages. https://doi.org/10.1145/3450626.3459748

1 INTRODUCTION

Polygonal meshes are the fundamental representation of geometric discrete objects for many computer-aided design, computer graphics, visualization, physical simulation, and computational geometry applications. Requirements of such applications range from sampling and evaluating the surface geometry or a subset of its attributes, querying the incidence or adjacency of mesh elements, or modifying the underlying geometry. Most of these requirements entail *local* processing of the underlying mesh, where the inputs to the computations on each mesh element are limited to a local neighborhood. Examples include computing discrete differential operators, mesh smoothing, and matrix assembly for solving variational problems.

Modern applications require working on meshes with millions of faces. Managing and manipulating such large meshes benefits from a programming model and a framework that implements that programming model, which ideally yields the benefits of performance, simplicity, correctness, and flexibility. Today's practitioners rely heavily on existing mesh data structure libraries and frameworks (e.g., CGAL [Kettner 2019], OpenMesh [Botsch et al. 2002], and libigl [Jacobson et al. 2018]) to liberate users from low-level implementation details and to facilitate finding user solutions more quickly, robustly, and with higher performance.

Most mesh processing today is still performed on CPUs. However, the superior performance of modern GPUs would seem to make them an excellent candidate for mesh processing applications due to the data-parallel nature of mesh processing workloads. However, existing CPU-based mesh library frameworks do not extend easily to the GPU because their programming models and implementations are designed around a serial mindset, or possibly target the limited parallelism of multi-core platforms, and are not designed for the hundreds of thousands of threads available on a modern GPU.

GPUs excel in processing structured grids (e.g., images or dense matrices) because they map well to the underlying hardware. In contrast, traditional approaches to mesh processing typically entail many levels of memory reference indirection with little correspondence between memory layout and the mesh topology layout, which shatters locality of reference and makes caching ineffective.

Current solutions on the GPU fall into two categories:

- Hardwired application-specific mesh processing implementations (e.g., Delaunay triangulation [Coll and Guerrieri 2017], mesh painting [Schäfer et al. 2014], and rendering subdivision surfaces [Tobler and Maierhofer 2006]). Such implementations may achieve best-of-class performance on a particular problem, but their data structures are specific to that problem. They may not be easily modified for new or related problems, and they may not make full use of the GPU's capabilities.
- Linear-algebraic reformulations of geometry processing workloads [Zayer et al. 2017] aiming to reduce intermediate data but do not optimize for locality, which is essential for top performance. This reformulation relies on representing meshes as sparse matrices and computation as operations on them. Our work shares this representation and advances the state of the art in the areas of performance, generality, and flexibility.

While not discussed in prior art, a careful implementation of a serial data structure on the GPU is possible. For example, storing the halfedge data of Directed Edges [Campagna et al. 1998] in a structure-of-array (SoA) format instead of array-of-structure (AoS) can make it a competitive alternative. However, such an optimization is beneficial only if the input is globally sorted; otherwise, caching is ineffective. We show that our implementation delivers superior performance even against Directed Edges with sorted input.

Delivering the highest performance on the GPU requires both a high-performance data structure and a powerful programming model. The data structure is responsible for capturing the locality of the underlying mesh topology in order to maximize GPU throughput. The programming model would permit its implementation to

Authors' addresses: Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens, Department of Electrical and Computer Engineering, University of California, Davis, One Shields Avenue, Davis, CA, 95616, USA, {ahmahmoud, sdporumbescu, jowens}@ucdavis.edu.

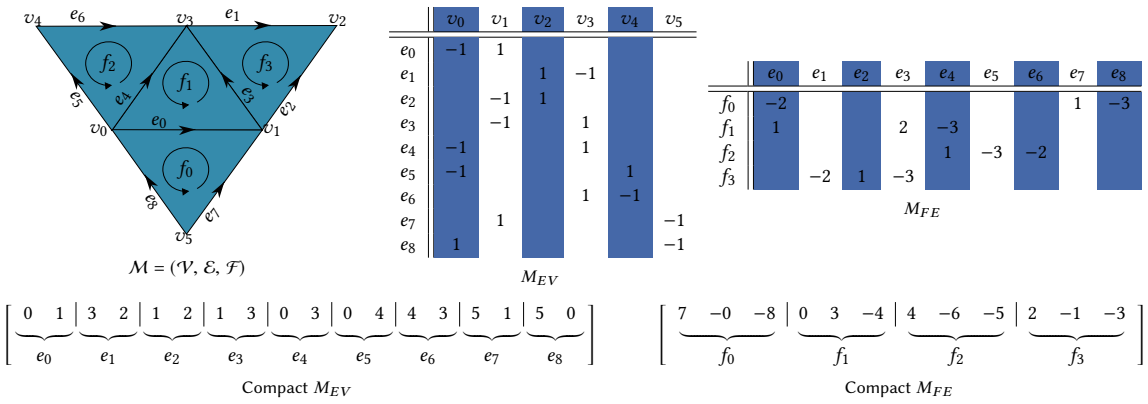


Fig. 1. It is possible to represent all incident and adjacency relations in a mesh in terms of sparse matrices. Here we show two such relations—EV and FE. In our implementation, we only store the top-down incidence relation between each k-cell and $(k-1)$-cell. Exploiting the matrices' sparsity, we store them compactly in flat arrays where indices indicate the elements connectivity. We generate all remaining incidence and adjacency matrices from these two (Appendix A).

transparently map work to computational resources through the data structure without user intervention. The final goal is to provide the user with the same experience as CPU-based libraries while simultaneously enjoying the high performance of the GPU.

We present *RXMesh*, a GPU triangle mesh data structure that captures locality by partitioning the input mesh into small patches that fit in the GPU's fast shared memory, ensuring excellent caching irrespective of query operations or input order. Our patching technique is fast, highly parallel, and accepts generic inputs. To eliminate communication during query operations, we extend each patch with information from neighboring patches with minimal overhead. Each patch is then represented independently using a compact sparse matrix representation that simultaneously allows for parallelization and excellent query load balance. RXMesh's complexity is hidden behind a simple programming model and interface that allows both ease of use and high performance across different applications. Our data structure and its applications are open-source[1].

In this paper, we make the following contributions:

- The design of RXMesh, a high-performance general-purpose static triangle mesh data structure on the GPU. Our data structure can capture the locality of the underlying mesh topology and uses bandwidth efficiently across the different levels of the modern GPU memory hierarchy. Our data structure enables a novel way of assigning work to computation resources in a load-balanced way that has not been used before for GPU mesh data structures.
- A clean programming abstraction that hides the complexity of the data structure behind a flexible programming model that allows the user to make the best use of our data structure without worrying about performance.
- The combination of our programming model and data structure is thoroughly evaluated via benchmarks and applications and compared against a well-optimized parallel Directed Edges [Campagna et al. 1998] data structure as well as (single- and multi-core) CPU-based frameworks. We achieve significant speedups over these frameworks.

RXMesh is meant primarily for GPU parallel applications. The CPU serial or multi-core mesh processing literature already features mature libraries and frameworks. Our work is limited to static applications that do not require changing the underlying mesh topology but may possibly change the geometric attributes of the mesh.

2 BACKGROUND

Triangular mesh $\mathcal{M}$ discretely represents the topology and geometry of some underlying 2D object embedded in 3D space. $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ consists of a collection of k-cells for $k = 0, 1, 2$ where 0-cells, 1-cells, and 2-cells are a set of vertices $\mathcal{V}$, edges $\mathcal{E}$, and faces $\mathcal{F}$ respectively and each $(k-1)$-cell lies on the boundary of a k-cell. We use the term *mesh element* or (*element* for short) to refer to a vertex, edge, or a face. A *2-manifold* mesh has no more than two faces sharing an edge and has two edges meeting at a vertex if and only if there is a face that contains them both. Otherwise, it is a *non-manifold* mesh. If an edge is on the boundary of only one face, it is a *boundary edge*, otherwise, an *internal edge*. A vertex and face incident to a boundary edge are a *boundary vertex* and *boundary face* respectively. Two vertices are *neighbors* if they share an edge. The *valence* of vertex V is the number of neighbor vertices of V. The *one-ring* of vertex V is the set of vertices that are neighbors to V. A face's *summit* is the set of vertices that form that face. Finally, *adjacency* relations assign a *neighboring* relation to two k-cells of the same dimension, while *incident* relations are between k-cells of different dimension. Such relations are summarized in Table 1.

3 RELATED WORK

The *halfedge* [Mäntylä 1988] data structure is the most commonly used mesh data structure supported by robust and reliable implementations (e.g., CGAL [Kettner 2019] and OpenMesh [Botsch et al. 2002]). Halfedge provides a straightforward way to manipulate

[1]https://github.com/owensgroup/RXMesh

Table 1. Full set of first-order queries.

Query	Definition
VV	For vertex V, return adjacent vertices
VE	For vertex V, return incident edges
VF	For vertex V, return incident faces
EV	For edge E, return incident vertices
EF	For edge E, return incident faces
FV	For face F, return incident vertices
FE	For face F, return incident edges
FF	For face F, return adjacent faces

meshes with constant-time queries for all local adjacency and incident relations. With few modifications, halfedge can represent non-manifold meshes [Dyedov et al. 2015]. Other data structures that have similar characteristics include quad-edge [Guibas and Stolfi 1985], winged-edge [Baumgart 1972], and Corner Table [Rossignac 2001]. Directed Edges [Campagna et al. 1998] specializes halfedge for triangular meshes and reduces the memory footprint of halfedge by devising special indexing rules that implicitly encode some connectivity information; in general, this and related "compact" mesh representations cannot support generic (e.g., non-manifold) inputs, which reduces their generality.

3.1 Mesh as a Matrix

Recent research, including this work, seeks to leverage the GPU's capabilities by formulating mesh data structures as matrices and computations as operations on them. The basis for this formulation rests firmly on algebraic topology and the elegant Linear Algebraic Representation (LAR) of mesh elements [DiCarlo et al. 2014]. LAR relies on encoding incidence relations between each k-cell to unordered $(k-1)$-cells in a *sparse* matrix format. These relations are also known as *boundary operators*. Query computations are realized in terms of sparse matrix-matrix multiplication—a well-studied topic within the HPC community. Furthermore, LAR naturally represents non-manifold meshes without any special treatment.

In LAR, any incidence or adjacency relation can be represented with sparse matrices where the matrix rows represent the *source* or input and the columns represent the *target* or output. Figure 1 shows two such matrices, where a nonzero value means the two mesh elements are incident. To reduce its memory footprint, LAR proposed storing only a subset of these matrices and dynamically computing the rest on demand from the stored subset.

For a triangle mesh, the minimum number of matrices to fully represent all mesh elements is two: one matrix for each top-down (from k-cell to $(k-1)$-cell) or bottom-up (from $(k-1)$-cell to k-cell) incident relations. For example, storing M_{EV} (incidence from edges to vertices) and M_{FE} (incidence from faces to edges) is enough to perform all queries as shown in the inset (and in more detail in Appendix A). Note that this matrix-matrix multiplication uses a different *semiring* than traditional matrix multiplication: replacing summation with logical *or* and multiplication with logical *and*, leading to a binary representation of incidence/adjacency. Higher-order queries can be answered similarly by using information from first-order queries. For example, computing the one-ring of the faces' summits is $M_{FVV} = M_{FV}M_{VV}$. While LAR sets the theoretical foundation for general-purpose high-performance mesh data structures on the GPU, it does not attempt to capture the mesh locality, which is crucial for high throughput.

$$
\begin{aligned}
M_{VV} &= M_{EV}^T M_{EV} \\
M_{VE} &= M_{EV}^T \\
M_{VF} &= M_{EV}^T M_{FE}^T \\
M_{EF} &= M_{FE}^T \\
M_{FV} &= M_{FE} M_{EV} \\
M_{FF} &= M_{FE} M_{FE}^T
\end{aligned}
$$

Representing meshes as sparse matrices indicates, in theory, that a general-purpose sparse matrix library could be used to implement LAR operations. However, we see two obstacles here. The first is that most sparse matrix libraries do not support alternate semirings. The second is that because meshes have a particular structure (e.g., M_{EV} will always associate one edge with two vertices), a general-purpose sparse matrix library misses opportunities for mesh-specific optimizations to sparse operations.

The LAR representation is the basis of recent work that targets mesh processing on the GPU, e.g., Mesh Matrix [Zayer et al. 2017] and the ternary sparse matrix representation [Mueller-Roemer et al. 2017] for volumetric meshes. Mesh Matrix represents surface meshes by encoding the relation between 2-faces (triangles) and 0-faces (vertices), limiting it to applications that do not require explicit edge representations. Mesh Matrix offers a compact representation as a single array augmented with an *action map*, a small local map that encodes the interaction between vertices. With the action map, Mesh Matrix claims to eliminate the need to create intermediate data. However, Mesh Matrix does not improve locality which, as we will show, is crucial for achieving top performance.

3.2 Programming Models for GPU Mesh Processing

GPU-specific programming models are found in many domains, e.g., graph processing [Wang et al. 2017], sparse voxel computation [Hu et al. 2019], and simulation [Bernstein et al. 2016]. The challenge for a programming model for GPU mesh processing is to provide both an intuitive, high-level abstraction for the programmer that encompasses a large set of mesh processing applications while making the best use of the underlying hardware.

The programming model of Mesh Matrix is one approach with its programming model centered on linear algebra primitives and action maps. Mesh Matrix refrains from mimicking existing halfedge-like operations and instead casts mesh processing workloads in the language of linear algebra. Mesh Matrix's programming model is orthogonal to ours. While Mesh Matrix seeks to reformulate the whole geometry processing pipeline, ours does not intervene in how the downstream computation is performed but only how it is scheduled and assigned to the computation resources. For example, Mesh Matrix requires re-writing applications in the language of linear algebra while ours provides the user with primitive query operations with which the user can compose their complex applications.

4 RXMESH PROGRAMMING MODEL

Traditionally, a mesh data structure provides the user with *handles* to operate over various elements. These handles abstract away element indices with *iterators* and *circulators* [Botsch et al. 2002]. This abstraction is suitable for serial processing since a single thread

works on elements sequentially. The same abstraction could be applied in the GPU-parallel context, where different threads work on different elements identified by the thread index. However, this could lead to poor performance due to memory divergence because elements that are topologically close could be assigned to threads that are not in the same index range. Thus, we depart from this traditional sequential mesh processing programming model in favor of one that offers higher performance.

Our programming model decouples user-specified computation from how that computation is assigned to GPU computation resources (e.g., threads). The user defines only the computation, which will typically include one or more query operations. Then our implementation assigns GPU threads to elements with the goal of exploiting locality for query operations and inducing load balance. This maximizes query performance, which is usually the bottleneck of mesh processing pipelines on the GPU. The user can define computation on either all or a subset of elements. While computing over all elements typically makes the best use of the GPU's computational power, our implementation is still able to exploit locality even when operating on a subset of the elements. A similar programming model has been used for sparse voxel computation [Hu et al. 2019] and proved to be powerful, performant, and flexible.

From a user perspective, our programming model is similar to the "think like a vertex" (TLAV) [McCune et al. 2015] programming model for the parallel processing of graphs. In TLAV, the user develops an algorithm by focusing on one vertex and the computation on that vertex based on its local data and incident and adjacent vertex and edge data, then applying that computation to all (or a subset of) vertices. Our programming model generalizes this idea to all three types of mesh elements: vertices, edges, and faces. Programs in our programming model, then, run in parallel over all elements, evaluate one or more queries into the mesh for each element, then combine those query results at each element with arbitrary user-specified computation. With this programming model, the user can specify single kernels that can operate on any combination of vertices, edges, or faces, and within those kernels, operate on each primitive set efficiently, in parallel.

As an example, consider computing the vertex normal at each vertex in a mesh by simply computing the normal of each face and atomically adding it to each of its three vertices. Our programming model requires the user to specify the computation, which includes making queries to fetch the three vertices of the face, computing the face normal, and then atomically adding the normal components to each vertex. It does not require the user to consider either parallel execution across faces, mapping threads to queries, or memory locality. This allows the user to write the computation kernels as shown in Listing 1 without worrying about these low-level details.

When operating on a subset of the elements, it is possible to query all the elements and then only use the results of those in the *active* set. However, this is a waste of memory bandwidth. Thus, we require the user to specify the participating elements in the active set using a lambda function that takes the element index as an input and returns a boolean indicating the element's membership in the active set. For example, the user can use the input element index to index a boolean array of the active set. By default, this lambda function returns true, thus the query should be applied on all elements.

```
__global__ void
  ComputeVertexNormal(RXMesh             rxmesh,
                      Vec3<float>*       VertexNormals,
                      const Vec3<float>* VertexCoords) {
  rxmesh.template kernel<Op::FV>(
    [&](const uint32_t f_id, const Iterator fv_iter){
    //The face's three vertices
    uint32_t v0(fv_iter[0]), v1(fv_iter[1]), v2(fv_iter[2]);

    //Compute face normal
    Vec3<float> faceNormal = ComputeFaceNormal(v0, v1, v2,
                                               VertexCoords);
    //Update vertex normals with faceNormal component
    atomicAdd<Vec3<float>>(VertexNormals[v0], faceNormal);
    atomicAdd<Vec3<float>>(VertexNormals[v1], faceNormal);
    atomicAdd<Vec3<float>>(VertexNormals[v2], faceNormal);});
}
```

Listing 1. Our parallel RXMesh programming model abstracts away the details of assigning work to processors. Here, threads are assigned automatically to faces, which leads to high throughput on queries. The user can focus on specifying only the computation, i.e., computing the face normal and adding it to the face's three vertices.

5 GOALS AND DESIGN PRINCIPLES

5.1 Goals

Given the programming model, we now describe a static triangle mesh data structure for the GPU that implements the queries in Table 1. Our data structure meets the following design goals:

Performance: Our primary goal is performance, measured by the elapsed time to process a mesh computation. We achieve this performance by exploiting locality, reducing memory operations, efficiently utilizing the different layers of the GPU memory hierarchy, and maximizing GPU occupancy. While much of a typical geometry processing application consists of local computations that are suitable for parallelization, the topological locality of the mesh representation is not usually captured in the data structures used in prior work. This lack of locality leads to poor memory performance. To maximize overall performance, we aim to capture this locality in our data structure. We focus on computation that queries mesh elements' *local* neighbors. A GPU-optimized data structure for *global* operations (e.g., BVH) is out of the scope of this paper.

Generality: While hardwired application-specific data structures can result in high-performance implementations of a specific application, their performance often degrades when they are deployed in a different application. Our goal is to provide a data structure that supports sustained high performance across a variety of applications. The target applications should be able to efficiently perform queries on any mesh element. Additionally, we make no assumption about the input mesh quality—we assume generic, possibly non-manifold, meshes expressed as indexed triangle inputs.

Compactness: Generality and high performance might be achieved by storing all possible query results, but at the cost of higher memory overhead, which can limit the user to small inputs only. More importantly, the limited size of the GPU's programmer-managed shared memory limits the amount of locality we can exploit. We strive to store a minimal amount of data and instead efficiently compute queries dynamically, resulting in a minimal memory footprint.

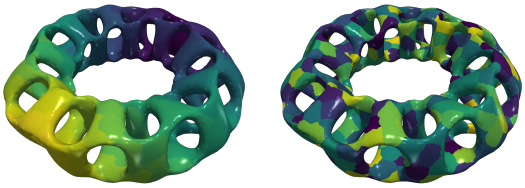


Fig. 2. Color indicates the face index, highlighting the different ways of capturing locality; global sorting (left) and patching (right).

Easy to use: Different applications may have different requirements for how they access a mesh data structure. The user might need to access the data structure directly from within user-defined GPU kernels. We aim to provide a data structure that allows efficient access with an intuitive access model for a variety of use cases.

5.2 Design Principles

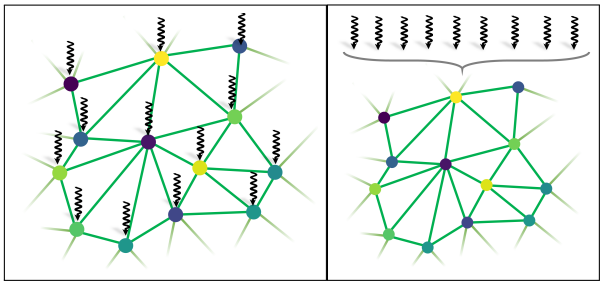
In this section, we explore two different methods to capture mesh locality: *sorting* and *patching*. We analyze why patching is the right choice, design a compact LAR-based representation for patches, and contrast it against alternative less-compact representations. Finally, we detail the importance of decorating the patches with *ribbons* for better locality.

5.2.1 Locality by Patching. Ideally, all the accesses necessary to perform a computation would be stored in the memory layer with the highest bandwidth. For the GPU, this is the L1 cache or the per-block shared memory. The mesh operations that we target have access patterns with high locality between mesh elements and their neighbors, so we would benefit from a data organization that can better capture that locality. Specifically, we aim for a coherent correspondence between the mesh topology and how the mesh is stored in the GPU global memory, i.e., elements that are topologically close are also stored nearby in memory.

The most straightforward implementation of our programming model would place all mesh data in global memory, making no optimization for locality. For this implementation, if the mesh data is unstructured, it is likely that hardware caches would capture little locality, limiting the performance.

We can improve locality capture, and hence performance, by sorting the input mesh coherently [Kerbl et al. 2018], using some kind of spatial information as the sort key (Figure 2). However, this approach has two disadvantages:

- It requires sorting not just the topology but also the mesh attributes (e.g., coordinates, texture coordinates, normals), which have a considerably larger storage requirement than the requirement for the mesh topology alone.
- Even if the initial sorting and data movement is not an issue, there is no generic method of sorting all mesh elements coherently, especially for meshes with a high genus number. For example, vertices can be sorted lexicographically based on their coordinates, but this leaves the faces and edges unsorted, and thus accesses to them will not be cache-friendly. Additionally, such sorting would always create occasional gaps between two neighbor elements, i.e., seams where there will be a transition in the element index.



(a) Traditional work assignment (b) RXMesh work assignment

Fig. 3. Directly assigning threads to mesh elements leads to load imbalance due to irregular mesh topology. Our RXMesh programming model assigns mesh patches to blocks, enabling threads to cooperatively perform their queries, leading to well-balanced computation.

The L1 cache may also be rapidly exhausted if mesh attributes are queried simultaneously with topology queries, which is the common case. Thus, we implemented an alternate design: subdivide the mesh into small patches that can fit in the user-managed shared memory and perform all the computation/queries in the shared memory. This guarantees that we always exploit the highest memory bandwidth even if the mesh attributes are used in the computation.

5.2.2 Work Mapping. Now that our accesses are within the fast shared memory, we turn to efficiently scheduling our computation. Because mesh data is sparse, simple methods to map work to processing resources often leads to idle threads, branch divergence, and memory divergence. For example, consider assigning mesh vertices to threads where each vertex may have a different valence (Figure 3). Ensuring good computation performance through load balancing is one of our key design goals. We achieve this load balance by appropriate work mapping where threads cooperate to perform their respective queries. We discuss this implementation in Section 6.2.

5.2.3 Index Space. The memory footprint of a patch can be reduced by using 16-bit indices to represent its elements. However, such *local* indices can only represent standalone *independent* patches, which is insufficient since the user expects a single index/handle per element that can be used for accessing mesh attributes. For that, we map each local index to a global one, resulting into two index spaces: a *local* and *global* index space. The local index space is used to perform the query operations, which return their results after being mapped to the global index space, thus hiding the complexity and details of patching from the user. The mapping has a low overhead as it only requires a single coalesced bulk read from the global memory, which we discuss in Section 6.2.

5.2.4 Compact Patch Representation. Our top priority in choosing a data structure is supporting the operations specified in our programming model (Section 4). Not all data structures support all operations; for instance, indexed triangles do not allow working on edges. Beyond this, we aim for a data structure that gives us both

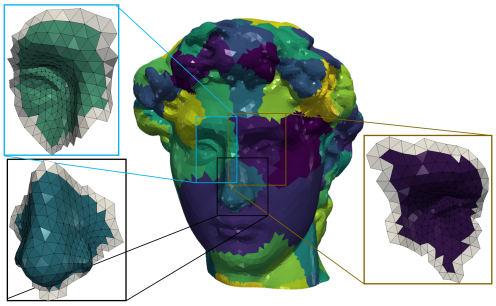


Fig. 4. We augment each patch by a *ribbon* (shown in white) to eliminate the communication between different thread blocks.

compactness and high performance. While a smaller overall memory footprint for a mesh is desirable, even more important for high performance, in the presence of our patching strategy, is minimizing storage per patch. Patches must be able to fit into shared memory, so storing more elements per patch allows more work per patch, increasing GPU utilization. Here we discuss two viable options:

Directed Edges: Directed Edges is a variant of the halfedge data structure specialized for edge manifold meshes. Directed Edges, which support all first-order queries, requires 21 bytes/face using 16-bit indices to represent the vertices, edges, halfedges, and faces within a patch in its local index space. The disadvantages of using directed edges are:

- It uses more memory than necessary (as we show below).
- It is only limited to edge manifold meshes. Generalizing it to non-manifold requires extra storage [Dyedov et al. 2015].
- It does not easily allow for threads to cooperate to fulfill queries; instead, each thread works independently, which might incur load imbalance in irregular meshes.

Mesh as a Matrix: Our choice, LAR, is an attractive representation for patch information because:

- A patch can be represented using only 12 bytes/face by only storing EV and FE in two matrices, namely M_{EV} and M_{FE} (Figure 1). All other relations can be queried on the fly.
- It can represent any mesh irrespective of its quality (e.g., non-manifold) without any special treatment.
- Threads can cooperate to perform queries. For example, computing VE is simply transposing M_{EV}, which can be computed by dividing rows equally among the threads. Since a row in M_{EV} contains only two entries, the computation is well-balanced across threads. We discuss these implementation details in Section 6.2.

5.2.5 Ribbons. Once we divide our mesh into patches, we find that the boundary mesh elements of the patches require special treatment because otherwise they will falsely represent boundaries of the mesh. For instance, querying the neighbor vertices of a patch boundary vertex will require reading another patch from global memory. This approach leads to both branch divergence and additional global memory accesses. Thus, we augment each patch with a *ribbon*—the union of the one-ring of the patch's boundary vertices—and add it to the patch local index space (Figure 4). This moderately increases memory usage per patch (we quantify this in Section 7) while significantly increasing the locality. Note that if the input mesh has a boundary, we do not add ribbons so we can accurately identify boundary elements.

6 IMPLEMENTATION DETAILS

We discuss here some details crucial for implementing our data structure.

6.1 Memory Storage

For every ribbon-augmented patch we store the following:

- M_{FE} and M_{EV} matrices with three and two nonzero entries per row, respectively.
- Local-to-global mapping for vertices, edges, and faces stored in a flat array, indexed by the element's local index and storing its respective global index.
- The total and *owned* number of vertices, edges, and faces in the patch.

Matrices: M_{FE} and M_{EV} are very sparse and thus a sparse matrix format is a natural way to store them (e.g., as compressed-sparse-row [CSR]). Zayer et al. [2017] showed that sparse-matrix-based storage of meshes can be further reduced from what CSR provides by taking advantage of the fixed number of entries per row. For M_{EV}, we store the edges in a flat array of size $2n_e$, where n_e is the number of edges and each pair of entries represents an edge. We store the vertices such that the first vertex is the source and the second is the target. Similarly, we store M_{FE} in a flat array of size $3n_f$, where n_f is the number of faces. We reorder the edges of each face such that their order indicates the face orientation and reserve one bit in each entry for the edge sign, as shown in Figure 1.

Local-to-Global Mapping: Since we augment patches with ribbons, some mesh elements are shared between more than one patch. For that, we define the "ownership" of an element by a patch as the patch that possesses all the information necessary to perform all queries for this element. Each mesh element is owned by only one patch, which we enforce by a mapping between (globally indexed) elements and their owning patches. When assigning local index to the different elements within a patch, we make sure that lower indices are given to the owned elements. Thus, if we want to check if an element is owned by a patch, we check its local index against the number of owned elements by the patch of that element type.

Global-to-Local Mapping: We store the patch owning the mesh elements as three arrays (for vertices, edges, and faces), indexed by the mesh global index. Once the patch is known, we can search within its local-to-global mapping array for the local index. However, this search is not needed for first-order queries.

6.2 Queries

We now discuss how to perform efficient queries given the patched mesh from how we assign threads to mesh elements to how we perform such queries. Ashkiani [2017] presented a novel per-thread-assignment, per-warp-processing strategy for parallel tasks on the GPU. In this strategy, the *user*, through the programming model,

assigns parallel tasks to threads, but the *implementation* instead assigns threads within a warp to work together to fulfill their tasks collaboratively. The result is increased warp efficiency due to better load balance when compared to traditional per-thread work assignment and processing, where branch and memory divergence may significantly inhibit high performance. Such an approach has been previously used in high-performance hash tables [Ashkiani et al. 2018] and graph data structures [Awad et al. 2020] on the GPU.

In this work, we extend Ashkiani's strategy to per-thread assignment, per-block processing. From the user's perspective, each thread is responsible for a mesh element's queries, but all threads in the block cooperate to fulfill their queries by both sharing useful information via fast shared memory and leveraging the even-faster intra-warp communication when possible. While queries are handled cooperatively, per-element computation instead uses traditional per-thread processing. Note that this processing and assignment is not exposed to the user and is done automatically. The user only implements the operation that each thread performs on the given mesh query output, closely following our programming model (Section 4). Since many mesh processing applications perform identical operations on the mesh elements (utilizing information about the mesh element's local neighborhood), our programming model can be adopted easily for these applications.

6.2.1 Structuring All Queries: For all queries, we first assign a single CUDA block to each patch. Let the number of source mesh elements owned by the patch be N_i and the number of the threads in a block T; each thread is nominally responsible for N_i/T mesh elements. One primary goal of our query implementation is to minimize global-memory communication and ensure load balance by performing as much computation locally within a block (in shared memory and registers) as possible and let threads collaborate to perform otherwise imbalanced queries. We start by loading the patch information from global memory into shared memory. Because we bound the maximum size of a patch, we guarantee that all storage can fit within shared memory, as it allows using 16-bit unsigned integers to represent the indices of per-patch mesh elements.[2]

M_{FV}, M_{EV}, and M_{FE} queries return a fixed number of outputs ($k = 2$–3); we term these *fixed offset* queries because we know for any input element i, its output will be stored in output locations $[ki, k(i+1))$. Other queries are *variable offset* because queries on all elements do not return the same number of outputs for each query (e.g., M_{VV} produces a variable-sized one-ring). We store the output of fixed-offset queries in a flat array in shared memory where the offset determines the boundary of each source's output. For variable-offset queries, we store the output in two arrays: one for the values and another for the prefix-sum of the offsets.

Now, the output needs to be mapped to the global index space. Reading the mapping from global memory would entail many scattered memory reads. Instead, we load the local-to-global mapping of the output element type into shared memory. Once the output is computed in local space, we use the local-to-global mapping to map the output of the query into the global index space. The mapping happens on the fly only when the user fetches the query's output.

It is possible to structure similar queries that only act on a subset of the mesh elements. In this scenario, each thread checks if any of its assigned source elements are part of the active set (Section 4). If one thread in the block has an active source element, the whole block performs the request query for the respective patch.

Queries that go beyond the first-order queries benefit from having the majority of the information resident in the shared memory after performing the first-order query. For example, querying the vertex two-ring is done by reading the one-ring of the one-ring. After performing the first one-ring, the next one-ring is already resident in the shared memory. However, for near-ribbon elements, this may require reading neighbor patches from global memory. For that, each thread adds to a shared-memory buffer the patch it needs to read in order to complete its query. This list is then filtered in place to generate a list of unique patches. The whole block then iterates over this list, performs the query for the whole patch, then allows threads to complete their queries. Subsequently, if additional patches still need to be read, they are scheduled in the next pass. Additional details on individual queries are discussed in Appendix B.

6.3 Patching

6.3.1 Patch Quality: We seek to partition the input mesh into a set of disjoint *patches* $\mathcal{P}$. A single patch should be contiguous, i.e., a single connected component. The patch size S_p is identified by its faces count. Ideally, we seek equal-sized patches to ensure perfect load balance when patches are assigned to different blocks. However, this is not feasible since partitioning a graph into roughly equal partitions is NP complete [Buluç et al. 2016]. Additionally, our experiments showed that occasional small patches do not degrade performance. Since we assign one CUDA block per patch, if a patch is small, its assigned block will finish in a shorter time, freeing the SM for another block ensuring full occupancy of the GPU. Smaller patches require less shared memory and thus may allow more thread blocks to be resident on one SM at the same time. However, smaller patches also increase storage overhead due to ribbons. Thus, our partitioning goal is contiguous patches of as equal size as possible, while tolerating small patches. The patching process should be fast, easy to parallelize, and incur low memory overhead.

Partitioning and clustering graphs and meshes for the purpose of distributing them across parallel processors reduces complexity and induces load balance. Buluç et al. [2016] summarize many of the plethora of techniques for graph and mesh partitioning. Mesh partitioning is used as a preprocess step to improve vertex locality to increase rendering performance [Kerbl et al. 2018], to approximate 3D shapes [Cohen-Steiner et al. 2004], to simplify meshes [Kalvin and Taylor 1996], and for mesh parameterization [Carr et al. 2006]. What makes our problem unique is our requirement for small-sized contiguous patches, with a patch size of $S_p = \sim 512$–768 faces.

State-of-the-art graph partitioning tools are not suitable to meet these requirements. For example, ParMETIS [Schloegel et al. 1997] is a MPI-based multi-core parallel graph and mesh partitioning

[2]The size of a patch S_p does not exceed 768 faces. The Euler-Poincaré characteristic, then, implies no more than $3/2 S_p$ edges per patch. Since we store two vertices for each edge (in M_{EV}) along with the three edges for each face (in M_{FE}), the total storage is $6S_p$. Since we use unsigned 16-bit indices to store the patch information, we require less than 10 kB per patch, which can fit in shared memory on any NVIDIA GPU. In addition and depending on the query operation, we might need to load only M_{FE} or only M_{EV}, leading to less shared memory usage and potentially better occupancy.

tool based on multilevel recursive-bisection, multilevel k−way, and multi-constraints partitioning schemes. ParMETIS excels at producing patches of equal size; however, it does not guarantee that result patches are contiguous. nvGRAPH[3] is a CUDA-based high-performance tool for solving various graph-based problems. nvGRAPH provides graph partitioning routines based on spectral clustering [Naumov and Moon 2016]. While nvGRAPH is parallel, fast, and able to partition graphs into roughly equal-sized partitions, it can only do this for coarse-grain partitions, i.e., fewer than 40 partitions. Otherwise, the required memory footprint is too high.

6.3.2 Patching Algorithm: We design a new mesh partitioning technique on the GPU to meet our requirements while taking advantage of 1) having no hard constraints over the number of patches and 2) having only upper bounds on the patch size. We leverage ideas from Lloyd's k-means clustering algorithm [1982], which is a highly parallel process to partition a given graph.

Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with nonnegative edge weights $\omega \in \mathbb{R}_{>0}$, k-means seeks to partition $\mathcal{G}$ into k partitions $P_1, \ldots, P_k$ of equal weights, i.e., the sum of weights of all edges in a partition is equal. Alternatively, weights could be associated with the vertices and the sum will run over the vertex weights. Lloyd's clustering algorithm is an iterative process to compute these partitions. After randomly selecting k (vertex) seeds, it iterates over two phases:

- Assigning vertices to the "nearest" seed to create k partitions.
- Updating the partition's seed with the partition's "centroid."

The algorithm iterates until seeds are no longer updated or a maximum number of iterations is reached. The algorithm requires a distance metric between vertices to compute the vertex's nearest seed and the partition's centroid.

We employ Lloyd's algorithm where the mesh faces are considered the vertices of the graph to be partitioned, two vertices are neighbors if the two faces they represent are adjacent, and all edges have a weight of 1. This formulation makes Lloyd's algorithm applicable regardless of the input mesh quality (e.g., non-manifold). Lloyd's algorithm is excellent in minimizing large variances in size between initial patches. However, the convergence of the algorithm plateaus after a few iterations, often leaving a few overly large patches. We overcome this by inserting more seeds in the large patches, effectively reducing their sizes in subsequent iterations. Our patching process stops when the largest patch size is less than S_p. Appendix C provides more implementation details.

7 EVALUATION

We evaluate our data structure and programming model on both fundamental query operations and full applications in Section 8. We perform our comparisons on an NVIDIA DGX Station with an NVIDIA Tesla V100 GPU with 32 GB of device memory. The CPU is an Intel Xeon E5−2698 v4 with 20 cores and 256 GB main memory. All code was compiled on Ubuntu 20.04 with gcc 9.3 and CUDA 11.1. Input meshes are collected from Thingi10K [Zhou and Jacobson 2016] and Smithsonian [Smithsonian Institution Digitization Program Office 2020] repositories.

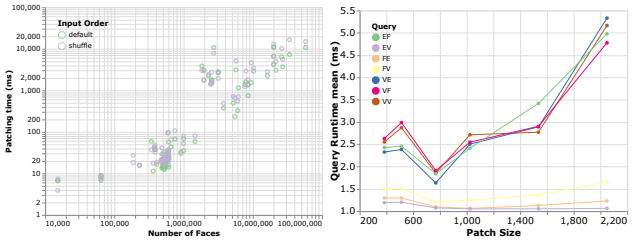
We compare our RXMesh against GPU and CPU data structures:

(1) Parallel Directed Edges (PDE): our well-optimized GPU-parallel implementation of Directed Edges [Campagna et al. 1998]. We have implemented the best possible version of this data structure that we could. Our implementation includes the following optimizations:
   - Using index-based instead of pointer-based data structures.
   - Using SoA instead of AoS to store a halfedge's information. We store the target vertex, face and next halfedge in three different buffers, each indexed by a halfedge index. Each vertex and face stores their halfedge in two different buffers indexed by the vertex and face index respectively. All other indirections are calculated implicitly, e.g., the two halfedges of an edge i take the indices $2i$ and $2i+1$ and thus the twin halfedge can be referenced implicitly.
   - Storing query outputs in registers (when possible) before storing them to slower global memory. This helps reduce memory transactions yielding better memory performance when the output size is known beforehand (e.g., FV query).
(2) OpenMesh (Version 8.1) [Botsch et al. 2002] and CGAL [Kettner 2019], the state-of-the-art CPU mesh libraries. We compare against both serial and OpenMP-parallel implementations (with the thread count set to `omp_get_max_threads` [40 on a DGX machine]).

Our measurements do not include the time it takes to read meshes from disk or the time to transfer data to the GPU. Additionally, our timings do not include the time it takes to create the patches designed to fit into GPU local memory. We do this for two reasons: (1) we get more fine-grained information about the performance of different parts of our system; (2) the cost of patch construction is quickly amortized over subsequent query operations. Patching time ranges from a few tens of milliseconds for small models up to 15 seconds for very large ones. Figure 5a shows the performance measurements of our patching technique on all data sets used in all experiments where it shows that patching time scales linearly with face count. Additionally, we compare against ParMETIS [Schloegel et al. 1997] on 8 input meshes containing 16−57M faces. On average, our patching technique is ~4.1× faster than ParMETIS while producing contiguous patches that ParMETIS does not guarantee.

Patch Size. One factor that affects RXMesh performance is the patch size. There are several competing factors that determine the patch size. While larger patches are desirable because they decrease ribbon overhead, they require more shared memory, which leads to fewer resident blocks per SM. The ratio between owned and ribbon mesh elements could be too low for small patches, leading to less useful work returned when the whole block processes a patch. To determine the best patch size, we tested six candidate patch sizes. We measure their performance on seven queries on an input with 20M faces, as shown in Figure 5b. We choose a maximum patch size of 768 faces for all experiments in this paper as it strikes the best balance between the competing factors delivering the best performance. We expect newer GPUs with more shared memory capacity would allow for larger patch sizes without degrading performance, leading to lower ribbon overhead and a more compact data structure. Figure 5c shows examples of the output patches on a few selected inputs.

[3] nvGRAPH is available at https://developer.nvidia.com/nvgraph/.



(a) Time to patch (b) Patch size

(c) An assortment of the output patches

Fig. 5. We show different aspects of the patching process; a) timing performance of the patching algorithm on all inputs used in this paper using default and shuffle input order, b) experimenting with different patch size on different queries, and c) examples of the output patches.

7.1 Query Operations

We evaluate the performance of RXMesh on the eight first-order query operations listed in Table 1. In each test, we pre-allocate enough memory to store the results of a query and do not count this allocation in the runtime.

PDE. We assign a thread to each mesh element source identified by the thread global index and launch enough blocks to cover all sources. Each thread then works independently, in parallel, to perform the query operation, then writes its output to its pre-allocated location in global memory.

RXMesh. Our programming model frees the user from explicit thread assignment. Instead, the user only writes the operations to be performed by every source element. Here, this is simply writing the output to global memory. Since we decouple the source mesh element index from the thread index, we could choose to write them in an order determined by either the element index or thread index. We choose the latter as it is likely to have better cache performance.

OpenMesh and CGAL. Both provide the user with iterators to circulate over different mesh elements. Given a query operation, we iterate over the source mesh elements in a `for` loop, use the provided iterators to query the given source mesh element for the query operation, and finally write the output to the memory.

Each experiment is run 10 times and we report the average timing in milliseconds. We run and analyze three different input orders for each query operation:

- *Default*: Input face and vertex order as specified in the input
- *Shuffle*: Randomly shuffle the vertex and faces
- *Sorted*: Using patching output to sort the vertex and faces

In general, OpenMesh and CGAL have the slowest performance by a factor of more than 100 when compared against the GPU alternatives (Figure 6). This performance gap motivates the development of a general-purpose mesh data structure on the GPU. Table 2 shows the speedup comparison between RXMesh and PDE of the three variants. We additionally report the timing for all methods in Figure 6. The shuffle input order is used to highlight the importance of locality for PDE. The PDE data structure does not require the input to expose any locality, instead relying on the user to capture locality, leading to 1) a loss in performance in the worst case scenario (i.e., randomized inputs) and 2) a failure to make the best use of locality even if the inputs are sorted (e.g., VV queries). In contrast, RXMesh exploits locality more efficiently by making patching an integral part of the data structure, leaving a randomized input no chance to jeopardize its performance. Below we analyze the different query operations, focusing only on sorted inputs.

VV, VE, and VF. In these queries, RXMesh outperforms PDE by factors of ~3.49× on average with sorted inputs. For PDE, these operations require each vertex to iterate over its halfedges. These memory reads could be at best cached but never coalesced. RXMesh relies on coalesced reads from global memory and confining computation within the shared memory, leaving no change for scattered global memory read. In addition, our efficient parallel matrix transpose in shared memory allows multiple threads to work cooperatively, avoiding any thread or memory divergence.

FV, FE, and FF. For PDE, these queries require (at most) three memory reads after each thread reads the halfedge associated with its face. These reads cache well, making PDE's performance match RXMesh's. For FF queries, RXMesh requires more shared memory, which lowers the GPU occupancy and leads to lower performance. We note that PDE is limited to edge-manifold input and is thus less generic compared to RXMesh.

EV and EF. Given an edge, PDE only does two memory reads for these queries to read the two vertices (or faces) associated with an edge's halfedges. These reads are always cached regardless of the input order, since the halfedges of an edge take two consecutive IDs. RXMesh performs a matrix transpose for EF since we assume non-manifold inputs, leading to a slight slowdown for this query.

The design we chose for PDE will always cache (and thus is best suited for) edge queries (EV and EF). An alternative design we could have chosen for PDE instead caches and prioritizes face queries (FV, FE, FF). In this design, we would implicitly reference a face's three halfedges, i.e., given a face i, assign its interior halfedges to indices $3i$, $3i+1$, and $3i+2$. This design would require slightly more memory (44 bytes/face). In contrast, the RXMesh user need not face such a design choice since RXMesh effectively caches all queries. This design advantage of RXMesh is most evident for the vertex

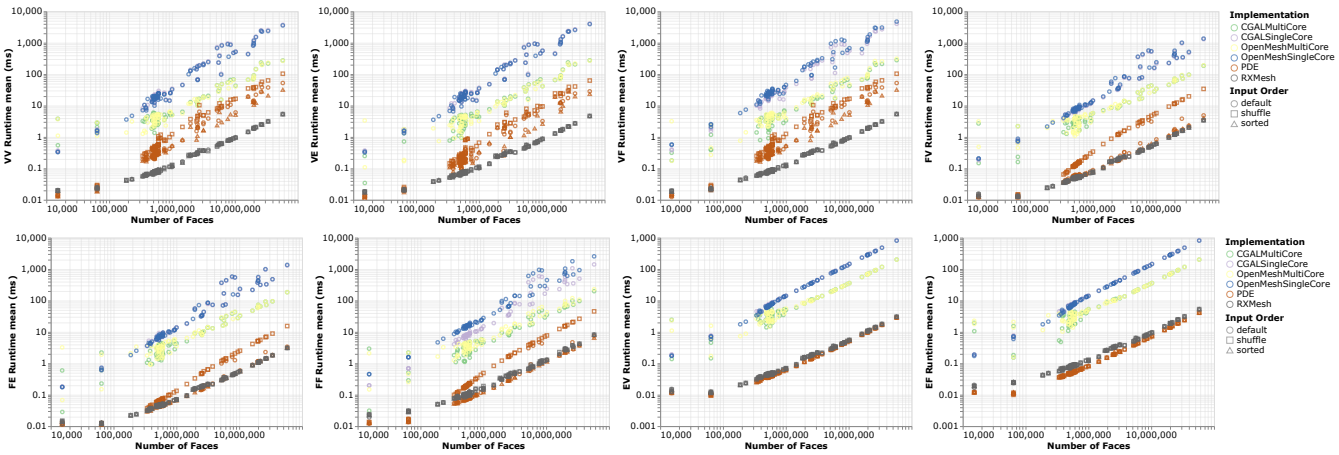


Fig. 6. Results of the first-order queries using RXMesh, PDE, OpenMesh, and CGAL. Each query shows runtime vs. number of input faces on the default, sorted, and shuffle input mesh ordering.

Table 2. Speedup of RXMesh over PDE for all query operations on different input order.

Operation		VV	VE	VF	FV	FE	FF	EV	EF
Order	default	4.95	3.48	4.8	1.27	1.05	0.87	0.86	0.64
	sorted	3.92	2.89	3.77	1.04	0.93	0.72	0.86	0.63
	shuffle	8.37	5.48	8.19	3.86	2.01	2.55	0.85	0.62

queries (VV, VE, and VF); no alternative design for PDE can cache such queries.

PDE enjoys a speedup over RXMesh for some queries (FE, FF, EV and EF) averaging ~1.28× for sorted inputs. For the other queries, RXMesh's speedup over PDE is on average ~2.58×. Since complex applications require a mix of these operations, RXMesh strikes a good balance in optimizing different operations and allowing working on generic meshes (i.e., non-manifold) without any specialization.

Memory Footprint Comparison. RXMesh requires 45.4 bytes/face (a detailed calculation is shown in Appendix D) while PDE requires 42 bytes/face, i.e., ~8% less memory than RXMesh. It is possible to reduce RXMesh's memory footprint to 34 bytes/face if computations are restricted to only first-order queries, leading to a ~19% savings in global memory vs. PDE.

8 APPLICATIONS

In this section we put the RXMesh system to work on a set of real-world geometry processing applications and evaluate its performance against both parallel CPU and GPU implementations using OpenMesh and PDE, respectively. In all our experiments, we use the sorted input order for fair comparisons.

Each application explores and tests a certain aspect of our data structure and programming model. Mesh smoothing using mean curvature flow (Section 8.1) integrates significant computation that repeatedly queries the vertices' one-ring exemplifying many applications that have a similar pattern of computation. Geodesic distance

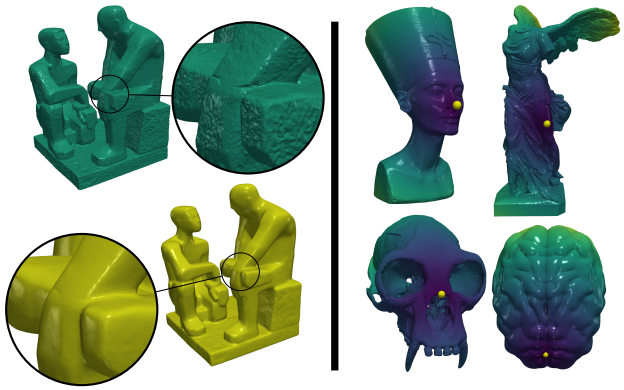


Fig. 7. Examples of removing high-frequency noise by smoothing using mean curvature flow (left) and computing geodesic distance (right).

(Section 8.2) illustrates the programmability of our programming model and how it could be adopted for computations that are run on a subset of the vertices while delivering high performance. Bilateral filtering (Section 8.3) shows the performance of RXMesh on queries that go beyond the first-order queries and how our programming model is amenable for such computation. Finally, vertex normal (Section 8.4) is a simple application that shows that RXMesh is able to match the performance of a hardwired application-specific data structure with almost no performance penalty.

8.1 Mean Curvature Flow

Geometry processing applications that require solving a linear system of equations are ubiquitous. Examples include mesh editing [Yu et al. 2004], mesh parameterization [Mullen et al. 2008], and simulation [Narain et al. 2012]. Unfortunately, many of these solvers are difficult to parallelize on the GPU. One costly aspect of solving these systems is the need to perform *matrix assembly* from the underlying mesh data structure into a sparse matrix while also maintaining the

original mesh representation. Iterative *matrix-free* solvers, which we use in this application, remove this obstacle and are the standard approach for parallelizing solver-based applications on large inputs containing millions of faces.

Here we implement smoothing/fairing of irregular meshes based on mean curvature flow [Desbrun et al. 1999], which is an effective method to remove the high-frequency noise from an input mesh (Figure 7). Our implementation uses an (un-preconditioned) conjugate gradient (CG) solver [Shewchuk 1994] using cotan weights computed at every iteration on-the-fly using VV queries. Given RXMesh's fast neighborhood queries, we can directly perform the necessary matrix-vector computations and make use of the easily parallelizable matrix-free iterative CG solver. Appendix E shows the user implementation of this application highlighting the simplicity of using RXMesh programming model where all the complexity of making the best use of the GPU is hidden from the user enabling them to focus on the application implementation.

For comparison, we implemented the matrix-free CG using (single and multi-core) OpenMesh and PDE. Figure 8a shows a single CG iteration timing for the different methods. We observe more than 210× (18.3×) speedup between RXMesh and OpenMesh single-core (multi-core) implementations, respectively. This further emphasizes the benefits of utilizing the GPU for such computation. RXMesh is on average 4.6× faster than PDE. We attribute this speedup to how computation is scheduled. For RXMesh, we first perform all the queries in shared memory, which requires coalesced memory reads. Any uncoalesced/scattered memory reads of mesh attributes are scheduled afterward, allowing the L1 and L2 cache to have a smaller working set. PDE performs both global memory reads of the data structure and mesh attributes simultaneously, stressing the caches and leading to more memory transactions and slower performance.

8.2 Geodesic Distance

In this application, we compute the geodesic distance from a single source vertex to all other vertices (Figure 7). Our implementation is based on the front propagation for computing approximate geodesic distance using the minimalistic parallel algorithm by Romero et al. [2019]. The core idea is based on propagation of distance information from a set of vertices *closer* to the source vertex to the set of vertices further away such that multiple vertices can be updated in parallel. These sets of vertices are called *topological level sets* and their topological distance (i.e., number of hops away) from the source is computed as a preprocessing step.

The algorithm iteratively selects a set of eligible vertices based on topological distance, updates their distance in parallel, and then computes an error to decide the next set of vertices. This application demonstrates how to limit computation to a set of *active* vertices and compares the performance against PDE and single-core OpenMesh. With RXMesh, we simply check if the vertex is contained in the active set based on the topological set eligible for update, following our programming model (Section 4). If the vertex is active, it then performs the necessary query (VV) and updates its distance. Behind the scenes, RXMesh assigns blocks to patches and threads to vertices, performs the queries for active patches, and retires blocks that are assigned to patches with no active vertices. In contrast, PDE only launches enough threads to cover the active vertices, which are read in a coalesced manner.

While RXMesh performs more work by checking on non-active vertices, it outperforms PDE since the extra work is trivial compared to the overall computation. Figure 8b shows the timing for RXMesh, PDE and OpenMesh. On average, RXMesh is 15.5× and 122× faster than PDE and OpenMesh respectively. This shows that patching and careful scheduling of queries delivers significant performance gains even if the computation is limited to a subset of the vertices.

8.3 Bilateral Filtering

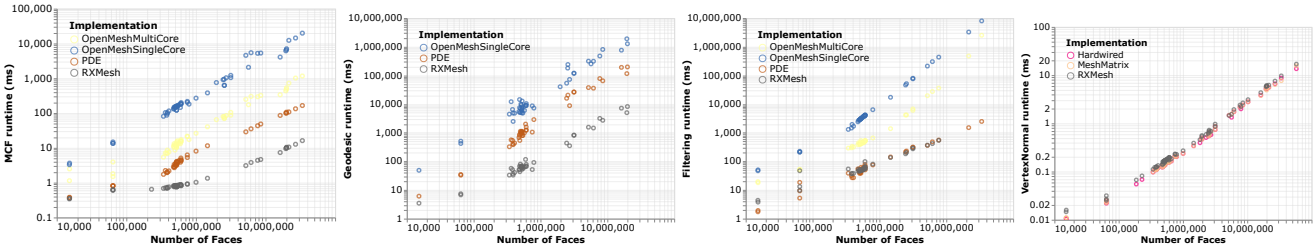
We implement an additional denoising application based on Bilateral Mesh Denoising (BMD) [Fleishman et al. 2003] to explore RXMesh's programmability and ability to generate *k-ring* queries. BMD is an iterative process that computes new, smoother coordinates for the input mesh by filtering the vertices in the normal direction using their local neighborhoods. The critical part of BMD is calculating the local neighborhood for a vertex. We calculate this neighborhood by generating a *ball* centered at the vertex based on the shortest edge length to the one-ring vertices. We then gather all the neighbor vertices that fall inside this ball by querying the k-ring where $k > 1$. This stops when none of the vertices of a k-ring fall inside the ball. The resulting vertices are then used to determine the distance to move the vertex along its normal vector.

In our implementation, the user starts with a VV query, during which each thread is assigned to a vertex and its output is used to computed the ball's radius. Subsequent queries beyond the one-ring are first checked if they are for vertices that are owned by the currently processed patch. If so, the output is resident in shared memory and is returned. Otherwise, the required patch is scheduled to be processed later. The user only implements the computation performed on the vertex and its query output while the programming model takes care of scheduling and processing the patches.

We implemented the BMD algorithm and compared RXMesh's performance against PDE and an OpenMesh-based implementation. RXMesh's speedup (Figure 8c) is on average 68.4× and 9.6× times faster than the single-core and multi-core OpenMesh implementation, respectively. PDE is only 1.12× faster than RXMesh. The reason behind this is RXMesh has to read more patches to fulfill the query of near-ribbon vertices, which could be as high as 10–20 additional patches depending on the mesh topology. The amount of useful information obtained by processing these patches is too low compared with the amount of work that needs to be done since these additional patches are processed to only benefit few vertices. We leave further optimizing higher-order queries as future work.

8.4 Vertex Normal

Computing a vertex normal is a fundamental mesh computation in many applications (e.g., smooth shading computation and discrete differential operators). We implement the weighted vertex normal [Max 1999] using RXMesh and compare against a hard-wired, vertex-normal-specific data structure, where the incident vertices of each face are stored directly in global memory (i.e., in an indexed triangle format). We also implemented the vertex normal using Mesh Matrix [Zayer et al. 2017], as outlined in their paper.



(a) Mean curvature flow (b) Geodesic distance (c) Bilateral filtering (d) Weighted vertex normal

Fig. 8. Timing performance of RXMesh, PDE, and OpenMesh on four different applications. All applications run on sorted input.

Listing 1 shows the implementation of vertex normal using RXMesh. Mesh Matrix and the hardwired implementation have the same memory layout, where each face only reads its three vertices without any additional memory indirection. Even without a perfectly-matched data structure, RXMesh is able to match their performance (Figure 8d); the hardwired implementation and Mesh Matrix are only 1.12× and 1.14× faster than RXMesh. This demonstrates that RXMesh's generality imposes only a minimal performance penalty.

9 LIMITATIONS AND FUTURE WORK

RXMesh currently targets only static meshes that do not change over time. While processing static meshes has broad applicability, we hope to enlarge the class of applications we can target by designing a programming model and implementation that supports dynamic meshes. One primary obstacle is a lack of well-defined semantics for low-level parallel mesh manipulation when multiple threads attempt to modify the same mesh element. Nonetheless we believe that our data structure, with its focus on mesh partitioning, is well suited for an evolution toward dynamic capability because parallelizing across partitions helps ease the problems with concurrency. We are further encouraged by recent advances in mesh subdivision on the GPU [Mlakar et al. 2020] and dynamic GPU graph data structures [Awad et al. 2020; Winter et al. 2018].

Currently, the patch creation time could be higher than the application runtime. Exploring further relaxation of the patch requirements in favor of speeding up the process is left for future work. Additionally, we have no theoretical guarantees that our patching process will always work. We may also encounter meshes with extreme structures (e.g., one vertex with a high valence that exceeds S_p) that may violate our assumptions and where no conforming patching is possible. We note that we have not encountered either a failed partition or such an extreme case in our experiments to date.

We assume that mesh-based computation is dependent on local neighborhood information only, which covers a large set of applications (e.g., finite-element/finite-volume methods). Our programming model is not well suited for computation that extends beyond the local neighborhood and requires global information, e.g., ray tracing or winding number.

Our data structure is specialized for triangle meshes only, which are ubiquitous within the geometry and mesh processing community. This specialization made it possible to greatly optimize our implementation. Extending our data structure and programming model to other type of discrete geometry (e.g., quad and volumetric meshes) is left for future work.

10 CONCLUSION

In this paper, we showed the importance of mesh locality for triangle mesh processing. While relying on hardware caching has been the standard way to exploit locality, we showed how programmer-managed caching, hidden from the user beneath the programming model, can improve the performance even further. It is particularly challenging to exploit locality in generic mesh processing because the user may interact with and compute on three different kinds of elements. However, we believe patching is the right way to exploit mesh locality and deliver sustained high performance regardless of operations or input quality. RXMesh's patch representation allows for several optimizations that are otherwise infeasible, e.g., PDE has no straightforward way to profitably use shared memory. Our programming model allows both flexibility to implement different and complex applications as well as an implementation that allows us to schedule query operations with high cache efficiency.

ACKNOWLEDGMENTS

The authors would like to thank Nina Amenta, Michael Garland, Pradeep Kumar Jayaraman, Kerry A. Seitz, and the anonymous reviewers for their feedback on the manuscript. We also thank Nigel Morris and Massimiliano Meneghin for the illuminating discussions and their continuous support. The authors appreciate the research support of the National Science Foundation (award # CCF-1637442), DARPA (AFRL awards # FA8650-18-2-7835 and # HR0011-18-3-0007), a UC Davis New Initiative Grant award, and Sandia National Laboratories, as well as equipment donations from NVIDIA.

This material is based on research sponsored by the Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

Saman Ashkiani. 2017. *Parallel Algorithms and Dynamic Data Structures on the Graphics Processing Unit: a warp-centric approach.* Ph.D. Dissertation. University of California, Davis. https://escholarship.org/uc/item/5qd0r4ws

Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. 2017. GPU Multisplit: an extended study of a parallel algorithm. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 2:1–2:44. https://doi.org/10.1145/3108139

Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018).* 419–429. https://doi.org/10.1109/IPDPS.2018.00052

Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. 2020. Dynamic Graphs on the GPU. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2020).* 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

Bruce G. Baumgart. 1972. *Winged Edge Polyhedron Representation.* Technical Report STAN-CS-72-320. Stanford University Computer Science Department, Stanford, CA, USA. https://apps.dtic.mil/dtic/tr/fulltext/u2/755141.pdf

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (May 2016), 12 pages. https://doi.org/10.1145/2892632

M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. 2002. OpenMesh – a generic and efficient polygon mesh data structure. In *1st OpenSG Symposium.* https://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf

Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent Advances in Graph Partitioning.* Springer International Publishing, 117–158. https://doi.org/10.1007/978-3-319-49487-6

Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. 1998. Directed Edges—A Scalable Representation for Triangle Meshes. *Journal of Graphics Tools* 3, 4 (Dec. 1998), 1–11. https://doi.org/10.1080/10867651.1998.10487494

Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. 2006. Rectangular Multi-Chart Geometry Images. In *Symposium on Geometry Processing (SGP06).* 181–190. https://doi.org/10.2312/SGP/SGP06/181-190

David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. 2004. Variational Shape Approximation. *ACM Transactions on Graphics* 23, 3 (Aug. 2004), 905–914. https://doi.org/10.1145/1015706.1015817

Narcís Coll and Marité Guerrieri. 2017. Parallel Constrained Delaunay Triangulation on the GPU. *International Journal of Geographical Information Science* 31, 7 (July 2017), 1467–1484. https://doi.org/10.1080/13658816.2017.1300804

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. 1999. Implicit Fairing of Irregular Meshes Using Diffusion and Curvature Flow. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99).* ACM Press/Addison-Wesley Publishing Co., USA, 317–324. https://doi.org/10.1145/311535.311576

Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. 2014. Linear algebraic representation for topological structures. *Computer-Aided Design* 46 (2014), 269–274. https://doi.org/10.1016/j.cad.2013.08.044 2013 SIAM Conference on Geometric and Physical Modeling.

Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Timothy J. Tautges. 2015. AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. *Engineering with Computers* 31, 3 (July 2015), 389–404. https://doi.org/10.1007/s00366-014-0378-6

Shachar Fleishman, Iddo Drori, and Daniel Cohen-Or. 2003. Bilateral Mesh Denoising. *ACM Transactions on Graphics* 22, 3 (July 2003), 950–953. https://doi.org/10.1145/882262.882368

Leonidas Guibas and Jorge Stolfi. 1985. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi. *ACM Transactions on Graphics* 4, 2 (April 1985), 74–123. https://doi.org/10.1145/282918.282923

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics* 38, 6 (Nov. 2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

Alec Jacobson, Daniele Panozzo, et al. 2018. libigl: A simple C++ geometry processing library. https://libigl.github.io/.

Alan D. Kalvin and Russell H. Taylor. 1996. Superfaces: Polygonal Mesh Simplification with Bounded Error. *IEEE Computer Graphics and Applications* 16, 3 (May 1996), 64–77. https://doi.org/10.1109/38.491187

Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 2018. Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the Modern GPU. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2, Article 29 (Aug. 2018), 16 pages. https://doi.org/10.1145/3233302

Lutz Kettner. 2019. Halfedge Data Structures. In *CGAL User and Reference Manual* (4.14 ed.). CGAL Editorial Board. https://doc.cgal.org/4.14/Manual/packages.html#PkgHalfedgeDS

Stuart P. Lloyd. 1982. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (March 1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

Yu Lu and Harrison H. Zhou. 2016. *Statistical and Computational Guarantees of Lloyd's Algorithm and its Variants.* arXiv:1612.02099v1 [math.ST]

M. Mäntylä. 1988. *Introduction to Solid Modeling.* W. H. Freeman & Co., New York, NY, USA.

Nelson Max. 1999. Weights for Computing Vertex Normals from Facet Normals. *Journal of Graphics Tools* 4, 2 (March 1999), 1–6. https://doi.org/10.1080/10867651.1999.10487501

Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *Comput. Surveys* 48, 2, Article 25 (Oct. 2015), 39 pages. https://doi.org/10.1145/2818185

D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer. 2020. Subdivision-Specialized Linear Algebra Kernels for Static and Dynamic Mesh Connectivity on the GPU. *Computer Graphics Forum* 39, 2 (2020), 335–349. https://doi.org/10.1111/cgf.13934

J. S. Mueller-Roemer, C. Altenhofen, and A. Stork. 2017. Ternary Sparse Matrix Representation for Volumetric Mesh Subdivision and Processing on GPUs. *Computer Graphics Forum* 36, 5 (2017), 59–69. https://doi.org/10.1111/cgf.13245

Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. 2008. Spectral Conformal Parameterization, In Proceedings of the Symposium on Geometry Processing. *Computer Graphics Forum*, 1487—1494. https://doi.org/10.5555/1731309.1731335

Rahul Narain, Armin Samii, and James F. O'Brien. 2012. Adaptive Anisotropic Remeshing for Cloth Simulation. *ACM Transactions on Graphics* 31, 6 (Nov. 2012), 152:1–152:10. https://doi.org/10.1145/2366145.2366171

Maxim Naumov and Timothy Moon. 2016. *Parallel Spectral Graph Partitioning.* Technical Report NVR-2016-001. NVIDIA Research. https://research.nvidia.com/publication/parallel-spectral-graph-partitioning.

Luciano A. Romero Calla, Lizeth J. Fuentes Perez, and Anselmo A. Montenegro. 2019. A minimalistic approach for fast computation of geodesic distances on triangular meshes. *Computers & Graphics* 84 (2019), 77–92. https://doi.org/10.1016/j.cag.2019.08.014

J. Rossignac. 2001. 3D compression made simple: Edgebreaker with Zip&Wrap on a Corner-Table. In *Proceedings of the International Conference on Shape Modeling and Applications.* 278–283. https://doi.org/10.1109/SMA.2001.923399

H. Schäfer, B. Keinert, M. Nießner, and M. Stamminger. 2014. Local Painting and Deformation of Meshes on the GPU. *Computer Graphics Forum* 34, 1 (Aug. 2014), 26–35. https://doi.org/10.1111/cgf.12456

Kirk Schloegel, George Karypis, and Vipin Kumar. 1997. *Parallel Multilevel Diffusion Algorithms for Repartitioning of Adaptive Meshes.* Technical Report 97-014. University of Minnesota, Department of Computer Science. http://glaros.dtc.umn.edu/gkhome/node/87.

Jonathan Richard Shewchuk. 1994. An introduction to the conjugate gradient method without the agonizing pain. https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

Smithsonian Institution Digitization Program Office. 2020. *Smithsonian 3D Digitization.* https://3d.si.edu/.

Robert F. Tobler and Stefan Maierhofer. 2006. A Mesh Data Structure for Rendering and Subdivision. In *Proceedings of WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision).* 157–162. http://www.vrvis.at/publications/PB-VRVis-2006-007

Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (Aug. 2017), 3:1–3:49. https://doi.org/10.1145/3108140

Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18).* Article 60, 13 pages. https://doi.org/10.1109/SC.2018.00063

Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. 2004. Mesh Editing with Poisson-Based Gradient Field Manipulation. *ACM Transactions on Graphics* 23, 3 (Aug. 2004), 644–651. https://doi.org/10.1145/1015706.1015774

Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. 2017. A GPU-adapted Structure for Unstructured Grids. In *Computer Graphics Forum (Proceedings of Eurographics 2017)*, Vol. 36. 495–507. Issue 2. https://doi.org/10.1111/cgf.13144

Qingnan Zhou and Alec Jacobson. 2016. Thingi10K: A Dataset of 10,000 3D-Printing Models. *CoRR* (May 2016). arXiv:cs.GR/1605.04797

A LAR EXAMPLE:

Figure 9 shows the six incident and adjacency queries with reference to the example shown in Figure 1. Each of these queries is represented as a sparse matrix resulting from a matrix-matrix multiplication or matrix transpose shown under the matrix.

B STRUCTURING SPECIFIC QUERIES:

We discuss here the implementation of RXMesh data structure on how to compute different queries where the needed patch information (M_{FE}, M_{EV}, or both) resides in shared memory. To minimize our shared-memory footprint, we aggressively reuse shared memory wherever possible, e.g., by overwriting the query output where patch information is stored.

In our implementation, we only store M_{FE} and M_{EV} and synthesize any other queries we require, as described below. The significant advantage of this decision is that we minimize storage (Section 5.2.4) and thus enable larger patches with their greater efficiency. This advantage comes with the computation cost of having to construct the queries listed below on the fly. Thus, we have invested significant effort into making this computation as inexpensive as possible, aided significantly by the storage of the relevant per-patch matrices in fast shared memory.

FE *and* **EV***:* They do not require any further computation after reading them from global memory.

FV*:* Since $FV = FE \times EV$, each thread reads the three edges of the face(s) assigned to it from M_{FE} and replaces the edges with three vertices. We incorporate information about the face orientation and edge direction to result in three unique vertices for each face (i.e., we write the first vertex of the edge unless the edge is flipped). The code snippet in Listing 2 shows how such computation can be done without thread divergence.

EF *and* **VE***:* They are simply matrix transposes of M_{FE} and M_{EV} respectively. Below, we discuss how to efficiently compute matrix transpose.

VF*:* We first compute FV as shown earlier and then transpose the output matrix in place. The input matrix has the same structure as M_{FE}, i.e., three entries per row.

VV*:* This query can be computed by first computing VE and then replacing each edge with the appropriate (other) vertex.

FF*:* Since $FF = FE \times EF^T$, we first transpose M_{EF}. Then each thread reads the three edges of its face(s), counts the number of (other) faces incident to this edge, and stores the results in a shared memory buffer. We then compute a prefix-sum of this buffer so that each face knows where to store its results in shared memory. This query requires both M_{FE} and its transpose to be resident at the same time in shared memory, which slightly increases the shared memory requirement for this particular query.

Matrix Transpose as Multisplit: It is now obvious that matrix transpose is such an important kernel for the majority of the queries (5 of the 8 queries require matrix transpose). Given the structure of the input matrices, we realize matrix transpose as a multisplit operation. Multisplit [Ashkiani et al. 2017] is a GPU parallel primitive that, given an unordered set of keys and a function that splits those input keys into buckets, outputs the buckets such that each bucket output is contiguous but otherwise unordered. This exactly matches the matrix transpose operation where the input is M_{FE}, M_{EV}, or M_{VF}, the function is the key itself, and the buckets are the matrix columns. We implement a custom multisplit for our transpose, which differs from Ashkiani et al.'s [2017] in that our total number of buckets (the number of columns) is significantly larger than any bucket's output. For instance, the valence of a vertex is on average 6 while a patch can have on average 384 vertices.

Listing 3 shows how we implement matrix transpose (inspired by multisplit) in a scenario where the offset is overwritten in the input matrix buffer. It requires a single template parameter, which can be derived from the maximum allowed size of a patch. Threads can than read a fixed number of entries from the input matrix (line 5–13). Each thread atomically adds the number of buckets it reads (line 18–22) so that a prefix sum can be computed (line 24) that tells each thread where to place its results (line 26–32). This kernel illustrates how threads can collaborate to perform an otherwise imbalanced computation by distributing the work among the threads.

C PATCHING ALGORITHM:

Our patching process implements Lloyd's algorithm on the GPU by iterating over three stages in order: *patch assignment*, *patch construction*, and *seed update*. Since we have no hard constraint on the number of patches, we add a fourth stage, *seed addition*, to accelerate convergence. Initially, patches are imbalanced and the traditional Lloyd's algorithm helps to deliver patches of equal size. However, this process is slow to converge [Lu and Zhou 2016]. Since our only hard requirement is patches below a certain size, adding a modest number of additional patches helps us to quickly meet our convergence criterion.

Initialization. We start the patching process by selecting random faces as the seeds for Lloyd's algorithm. If the mesh is composed of multiple components, we first add one seed per component and then distribute the remaining seeds proportionally to the components' size. The initial number of seeds is the number of input faces divided by the desired patch size S_p. Each seed face is assigned to a distinct patch. We also store the face count in each patch, initialized to one.

Patch Assignment. We implement a parallel iterative process for patch assignment such that a face assigned to a patch propagates its patch ID to each of its neighboring faces if they have not been assigned yet (using *atomic compare-and-swap*).

We store patch IDs per mesh face in a pre-allocated buffer in global memory. Patch assignment ends when all faces have been assigned to a patch. Because faces are assigned to patches only by their neighbors, our patch assignment process guarantees that each patch is a single connected component. If the seeds are well-spaced, this stage tends to produce patches of relatively uniform size. Isolated faces are identified at the beginning as separate components. These faces will be seeds but will not grow further.

Construct Patches. After assigning faces to patches, we construct a patch data structure. We represent patches in a compact format that consists of an *offset array* and a *value array*. The offset array is the

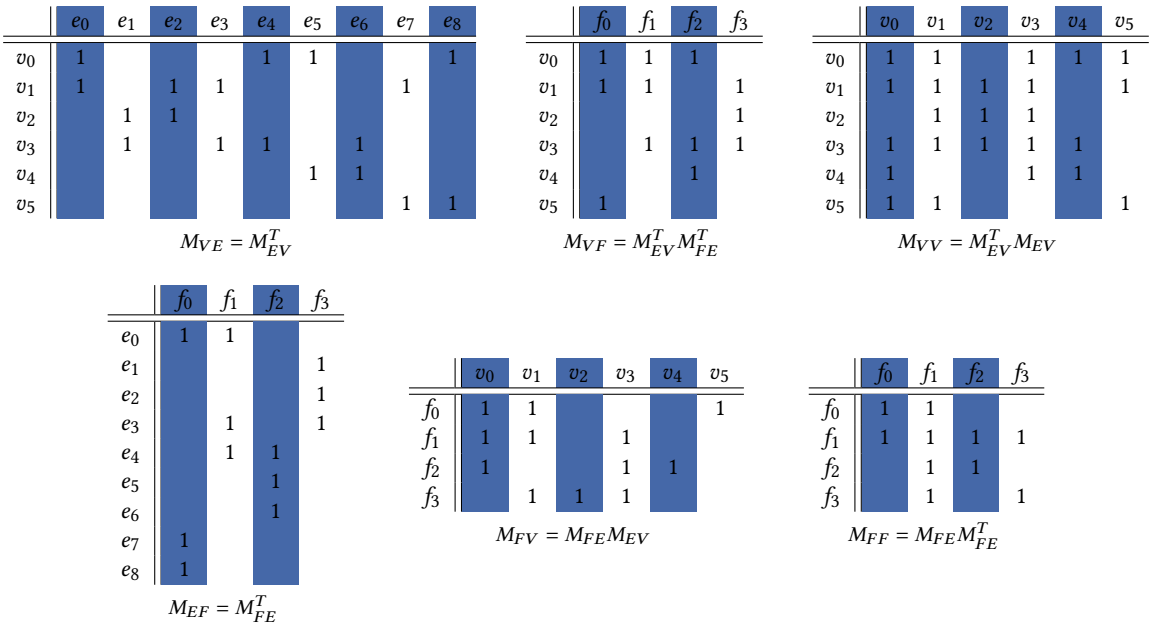


Fig. 9. With reference to the example mesh shown in Figure 1, here we show the remaining incident and adjacency relations between different mesh elements, expressed as sparse matrices, and computed as matrix-matrix multiplication and matrix transpose of the two stored matrices i.e., M_{EV} and M_{FE}.

```
__device__ void
ComputeFV(const uint32_t pNumFaces, const uint16_t* s_Mev,
          uint16_t* s_Mfe) {

  for (uint32_t f = threadIdx.x; f<pNumFaces; f+= blockDim.x) {
    for (uint32_t e =0; e< 3; ++e) {
      uint32_t edge = s_Mfe[f*3 + e];

      // get edge direction
      uint32_t edge_dir = edge & 1;

      // shift right to get the actual edge index
      edge = edge >>1;

      // if the edge is flipped, take the second vertex
      uint16_t vertex = (2*edge) + (1 + edge_dir);
      vertex = s_Mev[vertex]

      //store results
      s_Mfe[f*3 + e] = vertex;
    }
  }
}
```

Listing 2. Computing FV using patch matrices in shared memory.

prefix sum of the patch size array while the value array stores the IDs of the patch's faces. We construct this compact format by first computing the maximum patch size, necessary for termination, with CUB's[4] parallel reduce (with the *maximum* operator) on the patch size buffer. Next, we run CUB's inclusive prefix sum to compute the offset array. Finally, we launch a kernel where threads are assigned to different faces. Each thread atomically adds its face to its patch value array.

[4]CUB is included in CUDA: https://docs.nvidia.com/cuda/archive/11.1.1/cub/.

```
__device__ void template <uint16_t itemPerThread>
MatrixTranspose(uint16_t* Matrix, uint16_t* Output,
                uint16_t* nRows, uint16_t* nCols,
                uint16_t* nnzPerRow) {
  uint16_t nnz = nRows * nnzPerRow;
  uint16_t thread_data[itemPerThread];
  uint16_t local_offset[itemPerThread];
  for (int i = 0; i < itemPerThread; ++i) {
    uint32_t index = itemPerThread * threadIdx.x + i;
    if (index < nnz) {
      thread_data[i] = Matrix[index];
      Matrix[index] = 0;
    } else {
      thread_data[i] = 0xFFFF;
    }
  }
  __syncthreads();
  for (int i = 0; i < itemPerThread; ++i) {
    if (thread_data[i] != 0xFFFF) {
      local_offset[i]=atomicAdd(Matrix[thread_data[i]],1);
    }
  }
  __syncthreads();
  CUBPrefixSum(Matrix, nCols);

  for (int i = 0; i < itemPerThread; ++i) {
    if (thread_data[i] != 0xFFFF) {
      uint16_t offset = Matrix[thread_data[i]]+local_offset[i];
      uint16_t row = (itemPerThread * threadIdx.x + i) /
                     nnzPerRow;
      Output[offset] = row;
    }
  }
}
```

Listing 3. Matrix transpose

Update Seeds. The next step chooses a new seed per patch. We aim to choose a seed that is as central within the patch as possible. We begin by launching a kernel that assigns one block per patch. Each

```
void SmoothingMCF(RXMesh rxmesh, Attribute<float, 3> X0,
                  float tol, float time_step,
                  uint32_t max_iter){
    //CG variables. X is the smoothed coordinates
    //initialized to X0 i.e., the initial coordinates
    Attribute<float, 3> Q, D, R, B, X;
    Vec3<float>         alpha, beta, delta, delta_old, delta_0;
    uint32_t            iter(0);
    //*********** Initialization ***********//
    InitB(rxmesh, X0, B);

    //Q = matrix*X
    MatVec(rxmesh, X0, time_step, X, Q);

    //This computation takes place on the GPU
    R = B - Q;
    D = R;

    delta = R.dot(R);
    delta_0 = delta;
    //*********** CG loop ***********//
    while (iter < max_iter) {
        //Q = matrix*D
        MatVec(rxmesh, X0, time_step, D, Q);

        alpha = Q.dot(D);
        alpha = delta / alpha;
        X = X + alpha * D;
        R = R - alpha * Q;
        delta_old = delta;
        delta = R.dot(R);
        if (delta < tol * tol * delta_0) {
            break;
        }
        beta = delta / delta_old;
        D = beta * D + R;
        ++iter;
    }
}
//*************************************************************
__global__ void MatVec(RXMesh rxmesh, Attribute<float, 3> X0,
                       float time_step, Attribute<float, 3> IN,
                       Attribute<float, 3> OUT){
    rxmesh.template kernel<Op::VV>(
    [&](const uint32_t v_id, const Iterator vv_iter) {
        float    v_weight(0), sum_e_weight(0);

        //Last vertex in the one-ring
        uint32_t q_id = vv_iter.back();
        Vec3     res(0);
        for (const uint32_t r_id : vv_iter) {//iterate over 1-ring
            uint32_t s_id = vv_iter.next();  //vertex next to r_id
            float e_weight = edge_cotan_weight(v_id, r_id, q_id,
                                               s_id, X0);
            e_weight = max(0, e_weight);
            e_weight *= time_step;
            sum_e_weight += e_weight;
            res -= e_weight * IN(r_id);
            float tri_area = partial_voronoi_area(v_id, q_id,
                                                  r_id, X0);
            v_weight += max(tri_area, 0);
            q_id = r_id;
        }
        float diag = 2* v_weight + sum_e_weight;
        OUT(v_id) = res + diag * IN(v_id);
    });
}
//*************************************************************
__global__ void InitB(RXMesh rxmesh, Attribute<float, 3> X0,
                      Attribute<float, 3> B){
    rxmesh.template kernel<Op::VV>(
    [&](const uint32_t v_id, const Iterator vv_iter){
        float    v_weight = 0;

        //Last vertex in the one-ring
        uint32_t q_id = vv_iter.back();
        for (const uint32_t r_id : vv_iter) {//iterate over 1-ring
            float tri_area = parital_voronoi_area(v_id, r_id,
                                                  q_id, X0);
            v_weight += max(tri_area, 0);
            q_id = r_id;
        }
        B(v_id) = X0(v_id) * (2 * v_weight);
    });
}
```

Listing 4. Smoothing by Mean Curvature Flow

block starts by constructing the patch boundary faces, i.e., faces in this patch incident to faces assigned to a different patch. We store these boundary faces in a "visited" shared-memory buffer. Starting from these boundary faces, we use "push" traversal to identify the faces neighbor to the boundary faces and inside this patch. We assign threads to visited faces, and on each round, each thread checks if any of its incident faces is inside this patch and is not visited (using *atomic compare-and-swap*). If so, the thread marks the neighbor face as visited and adds it to the visited buffer. When all faces in the patch have been added to the visited list, we pick a face randomly from the faces added in the final round. This face—hopefully one at the "center" of the patch—is a seed in the next iteration.

Seed Addition. We repeat the above three stages until the maximum patch size is less than S_p. We accelerate the convergence by inserting new seeds along the boundaries of the large patches that violate the patch-size criterion. However, we do not do this on every iteration, instead prioritizing Lloyd's algorithm's opportunity to rebalance the existing patches toward equal sizes. We only insert new seeds when the convergence rate slows down. Our experiments show that inserting new seeds after every fifth iteration best balances accelerating the convergence without excessively increasing the number of patches.

D MEMORY FOOTPRINT:

We consider a simplified manifold input for RXMesh memory calculation. For each patch, RXMesh stores two matrices, M_{FE} and M_{EV}, each of size $3S_p$ (following the Euler-Poincaré characteristic). The entries of these matrices are 16-bit unsigned integers, totaling $12S_p$ bytes/patch. In addition, we store the vertex, edge, and face local-to-global mappings as 32-bit unsigned integers, requiring an additional $12S_p$ bytes/patch. The total storage per patch without the ribbon is thus $24S_p$ bytes. We observe that the increase in memory due to the ribbon does not exceed 39%. The total memory requirement per patch is thus $\approx 33.4S_p$ bytes. The total number of patches is F/S_p, where F is the total number of faces in the input mesh. Thus, the storage requirements for all patches is 33.4 bytes/face. This is the total memory footprint needed if computation is restricted to the first-order queries. Additionally, we also store the owner patch using 32-bit unsigned integers, which requires in total 12 bytes/face. This extra storage is needed only for higher-order queries (e.g., k-ring), increasing the memory storage of RXMesh to 45.4 bytes/face.

E APPLICATION CODE:

Listing 4 shows the user implementation of the smoothing application (Section 8.1). We follow the same implementation of the conjugate gradient by Shewchuk et al. [1994] and the variable names match those in Appendix B2 therein.