Unstructured Geometric Data Processing on the GPU
Data Structures & Programming Models

By

AHMED MAHMOUD

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Nina Amenta

Michael Neff

Committee in Charge

2024

To Mai, Mom, and Dad.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Unstructured Geometric Data Processing on the GPU
Data Structures & Programming Models

The surge of interest in 3D geometric data processing arises from the increasing demand for algorithms capable of analyzing and manipulating complex 3D geometry across various applications, including computational design, physical simulation, shape analysis, and virtual reality. Recent advancements in machine learning and data-driven approaches have further expanded the field's applications to areas like computer vision and AI-driven asset creation. Despite the growing influence of geometric data processing, most geometry processing algorithms are implemented using serial processes on the CPU. With more careful design and implementation, however, the latent parallelism in geometric data processing algorithms can be unlocked, enabling dramatic acceleration on highly parallel hardware such as the GPU.

In this dissertation, we argue that data structures and programming models inherited from serial/limited-parallelism processing of unstructured geometric data are not well-suited for efficient execution on the GPU. Instead, we propose the creation of tailored data structures and programming models designed explicitly for GPU hardware, aiming to achieve significant speedups in geometric data processing tasks. The rationale behind new data structures lies in the fundamental differences in the hardware architectures of GPUs and CPUs. Traditional CPU-optimized data structures often fail to exploit the massive parallelism of GPUs effectively, leading to suboptimal performance. Similarly, new programming models abstract the intricacies of low-level implementation details and GPU hardware optimization allowing users to focus on solving their computational problems efficiently.

In this dissertation, we focus primarily on the explicit representation of geometric data as unstructured triangle surface meshes. We introduce high-performance data structures tailored for both *static* and *dynamic* local triangle mesh processing on the GPU. These data structures effectively capture the locality of mesh topology, optimize memory bandwidth usage, and eliminate the need for any CPU-GPU data transfer. Our data structures support generic triangle meshes without stringent requirements on mesh quality and facilitate a wide range of static and

dynamic local mesh operators commonly found in CPU-based libraries. We propose different mesh processing optimizations, e.g., confining all topology operations within the GPU's shared memory, and relying on speculative processing for dynamic update operations. We also design intuitive programming models that abstract the complexity of these data structures, providing users with a clean interface without sacrificing performance.

Furthermore, we present a comprehensive system design that integrates these data structures and programming models, enabling end-to-end execution of various geometric data processing applications on the GPU. We conduct thorough evaluations and benchmarks, comparing our system against well-optimized GPU parallel data structures and CPU-based frameworks. Our evaluations cover a range of applications including geodesic distance computation, bilateral filtering, mesh smoothing, surface tracking, isotropic remeshing, and Delaunay edge flip. In static applications, we achieve notable speedups ranging from 4–15$\times$ over existing GPU parallel mesh data structures. For dynamic applications, our system sets new standards for GPU-based dynamic mesh processing, outperforming state-of-the-art multithreaded CPU alternatives by an order of magnitude on complex applications. Our system, along with its applications, is made openly accessible as an open-source project.

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to those who have supported and guided me throughout my PhD journey. At the heart of my academic and professional growth is my advisor, Professor John D. Owens. John taught me how to be a better researcher, author, collaborator, and presenter. His kindness and empathy as an educator and a mentor have been a constant source of inspiration. For me, John is a lifelong mentor whose influence will extend far beyond my academic career.

Dr. Mohamed Ebieda has been an invaluable mentor. His endless advice and guidance, especially during my transition from Naval Architecture to Computer Engineering, has been crucial. Mohamed opened my eyes to the beauty of research and computer graphics, and his unwavering support has been a cornerstone of my success. I am also grateful to Dr. Ahmed Abdelkader and Dr. Muhammad Awad for being not only great friends but also amazing co-authors. As fellow Egyptians, they helped build a sense of community that has been incredibly supportive.

Dr. Serban D. Porumbescu has been an exceptional research partner. His assistance in refining my research and papers, as well as our many meetings where he patiently listened to my thoughts until they made sense, has been invaluable. I am also deeply grateful to Professor Nina Amenta, Professor Michael Neff, Professor Zhi Ding, and Professor Bernd Hamann for serving on my Qualifying Examination and Dissertation Committees. Their valuable feedback has been instrumental in refining my work.

I owe a debt of gratitude to my managers at Autodesk Alex Tessier, Dr. Adrian Butscher, and especially Dr. Nigel Morris, for believing in me and offering me the opportunity to be part of the amazing Autodesk Research group even before I completed my PhD. Thank you for making it possible for me to balance my PhD with a full-time job.

To my good friend and co-author Dr. Massimiliano Meneghin, thank you for your excitement in exploring new ideas with me and for the work we have accomplished together at Autodesk. I will miss our collaborations. To all my friends and colleagues at Autodesk, including Dr. Jenmy Zhang, Mehran Ebrahimi, Dr. Farhad Javid, Dr. Pradeep Kumar Jayaraman, Dr. Hesam Salehipour, Akmal Bakar, Dr. Aditya Sanghi, Dr. Amir Khasahmadi, Dr. Hui Li, Dr. Hyunmin

Cheong, Dr. Nastaran Shahmansouri, Rhys Goldstein, and the rest of the Simulation, Optimization, and Systems Research group at Autodesk Research, your support has been pivotal in my professional growth.

I also want to acknowledge the Owensgroup past and current students: Dr. Matthew Drescher, Dr. Yangzihao Wang, Dr. Afton Geil, Dr. Jason Mak, Dr. Kerry Seitz, Dr. Leyuan Wang, Dr. Yuechao Pan, Vehbi Eşref Bayraktar, Dr. Saman Ashkiani, Collin McCarthy, Dr. Carl Yang, Weitang Liu, Dr. Muhammad Osama, Yuxin Chen, Dr. Zhongyi Lin, Jonathan Wapman, Agnieszka Łupińska, Radoyeh Shojaei, Dr. Nima Johari, Chuck Rozhon, Toluwanimi Odemuyiwa, Daniel Loran, Mythreya Kuricheti, Cameron Shinn, Matthew Yih, and Teja Aluru. You all made me feel like being at home during my initial years in Davis, CA. I would like to also thank the UC Davis Graduate School and EEC department staff and administration for their support throughout my academic journey. Finally, I am grateful to Changcheng (Eric) Yuan and Brooke Dolny for allowing me the privilege of becoming a mentor at one point in your career paths.

My research would not have been possible without the financial support from the National Science Foundation (award # CCF-1637442); DARPA (AFRL awards # FA8650-18-2-7835 and # HR0011-18-3-0007); the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR), Applied Mathematics Program; the Laboratory Directed Research and Development program (LDRD) at Sandia National Laboratories; UC Davis New Initiative Grant award; and equipment donations from NVIDIA. I also extend my appreciation to all the anonymous reviewers of my papers. Your critiques have helped me improve my work—sometimes against my will!

Last but not least, my deepest gratitude goes to my family. To my dad, for always pushing me to be a better version of myself. To my mom, the kindest person I have ever met, for the endless love and support. To my wife and the love of my life, Mai, for her unwavering support, kindness, and understanding throughout this journey and for the many journeys we will share in the future. Your love and encouragement have been my greatest strength.

Thank you all for your support, guidance, and believing in me!

Chapter 1

Introduction

The field of 3D geometric data processing, traditionally linked to simulation, visualization, and computer-aided design (CAD), is currently experiencing a surge of interest. This heightened attention stems from the increasing need for algorithms capable of analyzing, synthesizing, and manipulating 3D geometry. Applications of geometry processing span a wide range, including shape analysis, computational design, virtual reality, and 3D printing. Additionally, the integration of machine learning and data-driven approaches into algorithmic design has sparked significant advancements in this domain. These developments have propelled the field forward and expanded its application to areas like computer vision and the AI-driven creation of virtual assets.

Recent advances in 3D scanning, medical imaging, LiDAR, photogrammetry, and simulation have led to the creation of massive datasets of high fidelity geometry. Current geometric datasets feature meshes with millions of faces capturing intricate and highly detailed geometry. Unfortunately, processing such complex meshes on CPU-based systems is a time-consuming task, often taking hours to complete. There is a simultaneous demand for increased fidelity in other domains, such as real-time rendering. Real-time rendering innovations include the Nanite system [37], which enabled ray-tracing millions of triangles and micropolygons in real-time using the GPU. These advancements highlight the pressing need to explore alternative computational architectures for efficient and fast geometric data processing.

Digital representations of geometric data include meshes, point clouds, implicit surfaces, and polynomial patches (e.g., splines and NURBS). While other representations are gaining

traction, meshes remain the dominant geometric shape representation for applications like CAD, simulation, visualization, and geographic information systems (GIS). This situation explains the multitude of libraries for mesh processing (e.g., CGAL [50], OpenMesh [12], VCGlib [24], and libigl [45]) that have significantly lowered the entry bar to facilitate geometric data processing. However, existing libraries are predominantly CPU-based. These libraries exhibit strong offline performance and reasonable interactive performance on small mesh models. CPU-based solutions, however, struggle to handle highly-detailed meshes efficiently, causing geometric data processing to become a bottleneck in many domains as geometric detail increases.

GPUs offer a vast number of processing cores that can concurrently execute computations, allowing for significant acceleration in geometric data processing tasks. Leveraging the parallel nature of GPUs can yield substantial speedups when compared to CPU-based approaches. While processing unstructured geometry has lots of latent parallelism across the millions of geometric elements in a detailed mesh, it usually entails complex dependencies, synchronization, and many levels of memory reference indirections leading to inefficient utilization of massively parallel hardware. Thus, efficiently harnessing the power of GPUs for unstructured geometry processing remains an ongoing challenge, demanding further research to develop new approaches and optimizations.

1.1 Thesis

My thesis posits that by creating data structures and programming models specifically designed for the GPU hardware, we could achieve a significant speedup in processing unstructured geometric data by leveraging the GPU's massive parallelism.

Why new data structures: Data structures inherited from single-/multi-thread CPU processing of geometric data is not well-suited for efficient execution on the GPU due to the fundamentally different hardware architectures. GPUs are designed to handle thousands of threads in parallel, capitalizing on *data parallelism* to perform computation efficiently. Data structures optimized for CPU hardware tend to be designed for sequential accesses and operations which does not leverage the GPU's strength in handling multiple data points in parallel. GPUs generally have higher memory bandwidth compared to CPUs but also higher latency for memory

accesses. Data structures that are optimized for low latency accesses might not perform as well on the GPU where the preference is for data structures that can exploit the high bandwidth even if they incur higher latency. While both GPU and CPU architectures benefit from data locality, GPUs achieve peak performance with *coalesced* memory accesses where threads access contiguous memory locations concurrently. GPU performance can significantly degrade when threads in the same warp follow different execution paths, i.e., warp divergence. CPU-optimized data structures that lead to many conditional logic can cause this divergence. Traditional CPU data structures may utilize locks or atomic operations extensively to ensure safe concurrent access. While GPUs do support atomic operations, their overuse can severely impact performance due to the high cost of synchronization in a massively parallel environment. Data structures that minimize the need for synchronization are better suited for GPUs. Finally, GPUs also offer specialized memory and processing features, e.g., shared memory, texture memory, and various memory caching mechanisms. Data structures that are not designed with these features in mind might not utilize the GPU's capabilities fully.

Why new programming model: Providing efficient GPU data structures is not enough to take the full advantage of the GPU hardware. Programming directly at the data structure level requires in-depth knowledge of the architecture, memory hierarchies, and parallel execution models. Higher abstractions and programming models simplify the development process by hiding this complexity, allowing developers to focus on solving computational problems rather than on the intricacies of hardware optimization. Since geometric data processing involves irregular data structures and memory access patterns, a GPU-focused programming model could abstracts the details of data partitioning, load balancing, and synchronization and ensure efficient execution of these tasks. By offering a higher level of abstraction, these programming models improve user productivity and open up new possibilities for shared optimization and performance gains that might be difficult to achieve at the user level.

1.2 Scope

To better position this thesis, we first explore the landscape of geometric data processing. Here, we focus on the explicit representation of geometric data as unstructured triangle surface mesh

due to its wide use in many computer graphics and scientific computation applications. We try to characterize the pattern of computation in triangle mesh processes by identifying broad categories for different aspects of computations.

Topology: Static vs. Dynamic: We can categorize mesh processing workload into two categories based on their impact on mesh topology: static and dynamic. Static mesh processing involves queries on the mesh to gather topological and geometric information without altering the mesh structure/connectivity. Topological queries are queries related to the mesh connectivity, e.g., stencil operations as in vertex one-ring, computing the Euler characteristic. Geometric queries pertain to mesh attributes, e.g., face normals. Dynamic mesh processing modifies the mesh topology by adding or removing elements. Dynamic mesh processing could be considered as superset of the static one. However, it is possible that by focusing solely on static processing, we could unlock optimizations not possible in dynamic scenarios. Note that operations altering only the mesh's geometric attributes without affecting its topology are still classified under static processing, e.g., mesh smoothing [36].

Space: Local vs. Global: The space of operation is defined by the topological distance or coverage extent of the operations. Topological distance is determined through the mesh's connectivity, independent of its geometric coordinates. Local operations affect a confined area surrounding a given mesh element, e.g., subdividing a triangle or calculating cotangent weights. In contrast, global operations involve analyzing the entire mesh, often utilizing accelerated data structures, and cannot be confined to a local neighborhood. Examples of global operations include ray-mesh intersection and hole filling [100].

Component: Single vs. Multi Mesh: Many applications require only a *single* mesh to depict the underlying geometry, e.g., for analyzing and solving partial differential equations (PDEs). Conversely, the concept of *multi-mesh* encompasses those applications that employ multiple meshes to represent the same geometry at varying levels of detail. An example of multi-mesh usage is in level-of-detail techniques for rendering. In geometry processing, multi-mesh applications can involve using a coarser mesh as a proxy for a more detailed one (e.g., a mesh cage for deformation [33] or geometric multigrid on a surface mesh [96]) or representing the same geometry precisely across different meshes (e.g., in intrinsic mesh processing [87]). The main

computational pattern in working with multi-mesh is establishing correspondence between them where each mesh element on one mesh maps correctly to its counterpart on the other, despite differences in resolution or detail. While correspondence can be initially approached as a global operation, possibly accelerated by bounding volume hierarchies (BVH), the inherent similarity between the meshes representing the same geometry allows for the exploitation of locality. This exploitation could enhance the efficiency of calculating correspondence.

Given that the majority of mesh processing operations are memory-bound, understanding the memory access patterns in geometric data processing is crucial for optimizing performance on the GPU. The Space of computation significantly influences these memory access patterns. Local operations—whether static or dynamic—involve accessing or modifying a small area around a specific mesh element. Thus, these operations are characterized by having high locality between mesh elements and their neighbors. In contrast, global operations for static tasks typically require navigating a tree-like data structure. For dynamic tasks, global operations might involve modifying areas of the mesh that are not closely connected topologically, e.g., closing gaps or removing overlaps in an uncleaned mesh. Nevertheless, in parallel execution, global operations can leverage locality to improve efficiency, e.g., threads in the warp access the same/close-by region in the mesh.

In this landscape of geometric data processing, a significant amount of prior research has focused on improving the parallel execution of global static operations through the development of fast tree-like data structures (e.g., BVH, kd-tree, and octree) or hash tables [2, 58]. Some of these operations are hardware accelerated (e.g., ray tracing) that then find their uses for geometric data processing [72, 93, 94]. For global dynamic operations, the challenge of modifying or updating tree data structures on the GPU has garnered attention, particularly due to its relevance in rendering applications that demand rapid update operations [54, 54].

While the exploration and optimization of global mesh operations remain critical for advancing geometric data processing, the role of local operations within computational mesh processing cannot be understated. Local operations are paramount in many computational modeling and simulation applications. Requirements of such applications range from sampling and eval-

uating 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. For example, to simulate complex turbulence or multiphase flow phenomena, adaptive mesh refinement locally refines and coarsens the mesh as needed. This adaptability ensures that transient features are captured accurately, without burdening the computational resources [3]. Similarly, in materials science, simulations that deal with crack propagation or material failures often hinge on the ability of the mesh to locally refine around the evolving crack tip, ensuring that the details of the propagation pathway are well-represented [80]. In topology optimization [59], where material distributions within a design space evolve to meet performance metrics, the underlying mesh must dynamically adjust to these innovative configurations. Other domains that require local mesh processing include real-time interactive applications like surgical simulation [102] and cloth manipulation [75].

1.2.1 State of The Art

In this dissertation, we focus on local operations, both static and dynamic. Prior to the work done in this dissertation, the efficient execution of local operation on the GPU has not been widely addressed despite the pivotal role they play in mesh processing. Below, we briefly discuss the state-of-the-art of static and dynamic local mesh processing.

Static Local Mesh Processing: Solutions for static local operations on the GPU fall into two categories:

- Hardwired application-specific mesh processing implementations (e.g., Delaunay triangulation [27], mesh painting [84], and rendering subdivision surfaces [91]). 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 [99] 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.

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 [19] in a structure-of-array (SoA) format instead of an 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 will show in Chapter 4 that our data structure delivers superior performance than Directed Edges with sorted input.

Dynamic Local Operations: Current solutions for dynamic mesh processing on the GPU are only application-specific. Examples of these applications include surface tracking [23], mesh simplification [53, 79], mesh subdivision [56], and Delaunay refinement [21]. Thus, similar to the static case, solutions within these applications do not generalize well to other applications. A possible solution—albeit hypothetical—is to serialize dynamic updates on the CPU. This would leave the GPU underutilized for the duration of memory transfer and serialized update operations on the CPU. Such a solution will not scale well as the mesh size increases since the transfer of increasingly large amounts of data between the CPU and GPU becomes a significant bottleneck, severely limiting the overall efficiency and scalability of the process in handling extensive mesh datasets.

The inherent data-parallel nature of local static and dynamic mesh processing makes it perfectly suited for execution on the GPU. With a more principled design and implementation, the latent parallelism in mesh processing algorithms can be unlocked, enabling dramatic acceleration on highly parallel hardware such as the GPU. 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 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.

1.3 Contributions

In this dissertation, we make the following contributions that widen the set of geometric data processing applications that can be executed efficiently on modern GPUs:

- **Data structures** for high-performance static and dynamic triangle mesh data structure on the GPU that capture the locality of the underlying mesh topology and uses bandwidth efficiently across the different levels of the modern GPU memory hierarchy while avoiding CPU-GPU data transfer. Our data structures assign work to computation resources in a load-balanced way that has not been used before for GPU mesh data structures. Our data structures handle generic triangle meshes without hard requirements on mesh quality, i.e., non-manifoldness or orientability. We do not impose any requirements on the type of static or dynamic operations—so long as they have a local area of impact. We support almost all common static and dynamic local mesh operators found in open-source CPU-based libraries.

- **Programming models** that provide a clean abstraction that hides the complexity of the data structure allowing the user to make the best use of our data structures without worrying about performance. Our programming models provide intuitive concise semantics for both static mesh queries and dynamic mesh updates and for resolving conflicts.

- **System** design that combines the above data structures and programming models that serve a wide range of applications to be executed end-to-end on the GPU. Our design considers both the topology and geometry (i.e., attributes) of the mesh, liberating the user from low-level intricate implementation details.

- **Applications** and benchmarks that thoroughly evaluate our system against well-optimized GPU parallel data structure as well as (single- and multi-core) CPU-based frameworks. Our applications include geodesic distance computation, mesh fairing, bilateral filtering, surface tracking, Delaunay edge flip, and surface remeshing. For static applications, we achieve 4–15$\times$ speedup on selected applications over GPU parallel mesh data structures. For dynamic applications, our system sets the bar for dynamic mesh processing on the

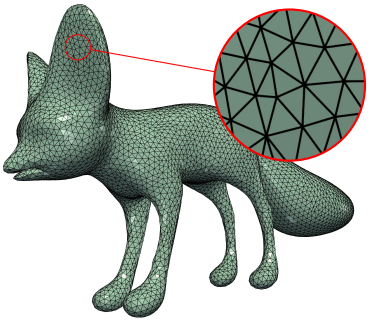
GPU and delivers an order-of-magnitude better performance compared to state-of-the-art multithreaded CPU alternatives. We also showcase scenarios where our performance may not be leading, yet it remains on par with other mesh frameworks. Our system and its applications are open-source.[1]

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

Chapter 2

Background & Preliminaries

2.1 Mesh Terminology



A *mesh* is a surface of polygonal faces glued together along the edges (as shown in the inset). These polygons could be of any shape, e.g., triangles, quads, and irregular polygons. In this work, we focus primarily on triangle surface meshes.

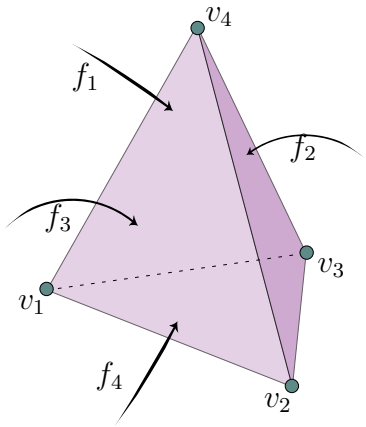
Formally speaking, triangular mesh $\mathcal{M}$ discretely represents the topology and geometry of some underlying 2D object embedded in 3D space. $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ consists of a collection of k-cells for $k = 0, 1, 2$ where 0-cells, 1-cells, and 2-cells are a set 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 face. A *2-manifold* mesh is one where edges and triangles meet at a vertex can be arranged in a cyclic order $t_1, e_1, t_2, e_2, \ldots, t_n, e_n$ without repetitions such that edge e_i is an edge of triangles t_i and t_{i+1}—implying that an edge is incident to exactly two faces. 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 is 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 edges connected to V. The *one-ring* of vertex V is the set of vertices that are connected to V by an edge. A face's *summit* is the set of vertices that

Table 2.1: List of first-order queries on surface triangle mesh

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

form that face. A mesh is said to be *orientable* if it is possible to assign a consistent orientation to all its faces. Finally, *adjacency* relations assign a *neighboring* relation to two k-cells of the same dimension, while *incident* relations are between k-cells of different dimensions. Such relations are summarized in Table 2.1. Higher-order queries can be composed of these first-order queries. Of particular interest is the *two-hop* neighbors that require stepping ("hopping") on the first-order neighbors before reaching the answer; for instance the union of one-ring of the one-ring vertices.

Meshes can be specified in many formats, however, *indexed triangles* format is by far the most widely used, exemplified by the *Wavefront OBJ* file format. In this format, mesh vertices are expressed as ordered triples of coordinates followed by an ordered list of faces' vertex indices. An example is shown in Figure 2.1 for a simple tetrahedron. The OBJ file format additionally allows isolated edges and vertices, making it feasible to represent triangle meshes regardless of their quality, e.g., self-intersection, non-manifold.




```
v 0.0 0.0 0.0
v 0.5 1.0 0.0
v 1.0 0.0 0.0
v 0.5 0.5 1.0
f 1 4 3
f 4 2 3
f 4 1 2
f 1 3 2
```

Figure 2.1: Example of indexed triangles format to represent a tetrahedron composed of four triangular faces.

2.2 GPU Terminology

This work targets NVIDIA GPUs using the CUDA programming model [77]. Here we outline the most important aspects of achieving high performance on these GPUs.

Modern NVIDIA GPUs are throughput-oriented manycore processors that use massive parallelism and latency-hiding techniques to achieve peak performance. The GPU features both computational and memory hierarchy that allow tens of thousands of hardware-scheduled threads to run simultaneously. The orchestration between these hierarchies is the determining factor for the full utilization of these computational threads. NVIDIA GPUs contain many parallel *cores* called "streaming multiprocessors" or "SMs", each has several parallel *processors*. The GPU thus can achieve parallelism within and across cores. NVIDIA GPUs use the Single Instruction Multiple Thread (SIMT) programming model to achieve data parallelism where multiple independent threads execute concurrently using a single instruction.

GPU programs are called "kernels" and are launched over a grid of thread blocks (virtualized cores). The GPU hardware is responsible for assigning thread blocks to SMs in a way that keeps cores busy whenever thread blocks are available. All blocks and threads can access a large shared pool of global memory and an on-chip L2 cache. Each thread block also has access to a small pool of user-programmable "shared memory" accessible only to threads in the thread block as well as a small hardware-managed L1 cache, also local to the thread block. In recent

GPUs, L1 cache and shared memory are combined within a 128 KB of data cache. Data loaded from global memory is implicitly cached in L1 and L2. Each thread has exclusive access to registers (maximum of 255 registers per thread and 64k per SM). Registers have the lowest latency, highest throughput, and smallest aggregate size, followed by shared/L1, followed by L2, followed by global memory. One key to high performance is to design data structures to primarily access the fastest levels of the memory hierarchy.

Within a thread block, the GPU hardware runs 32 threads (a "warp") at a time in lockstep. If those 32 threads access global memory, they obtain the highest bandwidth only if they access neighboring addresses in global memory (these accesses are termed "coalesced"). When a warp stalls execution (e.g., because of a long-latency memory read), the GPU quickly switches to a different warp and hence can hide the latency of the first warp. The GPU can best hide latency if it has many warps available ("resident") to run at any time. The metric of "occupancy" reflects the fraction of time that the GPU is busy running an active warp; maximizing occupancy by minimizing per-warp and per-block resources, allowing more warps to be resident, is another key to high performance.

Chapter 3

Related Work

3.1 Mesh Data Structures

Efficient mesh data structures enable faster processing, reduced memory usage, and enhanced accuracy in the representation and manipulation of complex 3D geometries. Here we focus on data structures of mesh *topology* (i.e., connectivity information), which is distinguished from the mesh *geometry* (i.e., geometric attributes on the mesh elements). The study and development of efficient mesh data structures, an area as old as the inception of personal computers [9], have been a significant focus in computer graphics research. We still rely on this early work on mesh data structures even with the massive evolution of computer hardware architecture. The Winged Edge data structure [9] stores adjacency information, enabling efficient navigation across the mesh by linking faces and vertices to edges. The Halfedge data structure [67]—one of the most widely used data structures for polygonal meshes—splits each edge into two half-edges with opposite directions, facilitating the traversal and manipulation of mesh surfaces with mature, well-maintained implementations in various libraries, e.g., CGAL [50]. With few modifications, halfedge can represent non-manifold meshes [34]. Directed Edges [19] specializes halfedge for triangular meshes and reduces the memory footprint of halfedge by devising special indexing rules that implicitly encode some connectivity information. The Cell-tuple [15] is used for higher-dimensional meshes, providing a more flexible representation for complex geometries. Linear Algebraic Representation (LAR) [32] was introduced as an alternative representation for polygonal meshes. Departing from the graph-like representation, LAR represents meshes as

sparse matrices while query and update operations are sparse matrix multiplication or matrix transpose.

3.2 Parallel Mesh Processing

Due to the limited processing and memory capacity of a single-core system, researchers and practitioners have long sought to process meshes more quickly and efficiently through distributed and multi-core systems. The data structures used in parallel systems are generally the same as those used in sequential processing systems but with modifications to facilitate and reduce communication across partitioned mesh boundaries, to manage mesh attributes, and to maintain correspondence between the geometric representation and its discretized mesh representation [25, 44].

In an effort to leverage existing codes, Cirrottola and Froehly [25] designed a system and algorithm where existing sequential remeshers are used within a parallel framework. They also describe a repartitioning algorithm to more easily move interfaces between parallel regions. Creating new parallel applications, improving performance, or porting parallel systems to new hardware is often difficult because of the tight coupling of code responsible for functionality with code responsible for achieving performance. Tsolakis et al. [92] breaks this coupling by creating a tasking framework for speculative mesh operations based on a *separation of concerns*, i.e., functionality vs. performance. Our work in this dissertation is similar in this regard and we seek to abstract away mesh operations from how those operations are performed in parallel on the GPU. Jiang et al. [46] address many of these issues through a declarative programming approach where users focus on their desired mesh processing steps by specifying *invariants* and *desiderata* and where the underlying system deals with the necessary scheduling and parallelization of low-level mesh operations. PUMI [44] focuses on alleviating the bottleneck (i.e., geometry and mesh processing) in end-to-end simulation runtime on massively parallel computers through infrastructure that provides a link between the mesh and the original domain, a partitioning model that facilitates interactions across nodes, and load balancing.

3.3 GPU Mesh Data Structure

Mesh as a Matrix: Recent research, including some of the work in this dissertation, 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 [32]. 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 operations—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 4.4 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. 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.

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

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 [99] and the ternary sparse matrix representation [73] 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 is crucial for high throughput.

3.4 Mesh Processing Programming Model

GPU-specific programming models are found in many domains, e.g., graph processing [95], sparse voxel computation [43], and simulation [10]. The challenge for a programming model for GPU mesh processing is to provide 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 a programming model that is 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, in this dissertation, we do not wish to intervene in how the downstream computation is performed but only in 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.

Many runtime libraries expose different programming models for dynamic operations. The

most widely used approach for exposing mesh manipulation is through *local operators*, e.g., edge flip, vertex split [12, 14, 24, 29, 50]. These operators are inherently linked to the underlying data structures, leading to an inseparable intertwining of the user interface and the implementation details. The cavity operator [63] was introduced for anisotropic mesh adaptation as a generic operator for implementing mesh updates. With the cavity operator, every operation creates a hole in the mesh and then fills it with a different set of mesh elements. A similar idea was used for mesh improvement [1] where the cavity could shrink or expand to meet different objectives for mesh improvement e.g., non-obtuse triangulation. While not extensively explored in prior work, the cavity operator offers an elegant and generic programming model for mesh updates, distinguished by its independence from specific data structures.

3.5 Domain Specific Languages (DSL)

DSL and compiler techniques can be used to improve portability across different architectures. Liszt [31] is a DSL designed for building mesh-based PDE solvers, featuring specialized language statements for interacting with unstructured meshes and managing data. Its compiler leverages program analysis to uncover parallelism, locality, and synchronization in Liszt programs, enabling the generation of applications optimized for various platforms, including clusters and GPUs. Ebb [10] is a DSL for simulation that is efficiently executable on both CPUs and GPUs, distinct from prior DSLs due to its three-layer architecture that separates simulation code, data structure definitions for geometric domains, and runtimes for parallel architectures. This structure allows for the easy addition of new geometric domains through a unified relational data model, enabling programmers to focus on simulation physics and algorithms without the complexities of parallel computing implementation. While compiler-based techniques and DSL provide concise easy-to-use interfaces, their main disadvantage is the need for relatively time-consuming static analysis of the input data. These compiler techniques are not easily amenable to dynamic mesh updates, which generate their workloads at runtime. Additionally, static analyses are unable to reveal the parallelism in dynamic mesh update applications, as the parallel schedule is heavily reliant on runtime data and cannot be determined at the time of compilation [55].

Chapter 4

Static Triangle Mesh Processing[1]

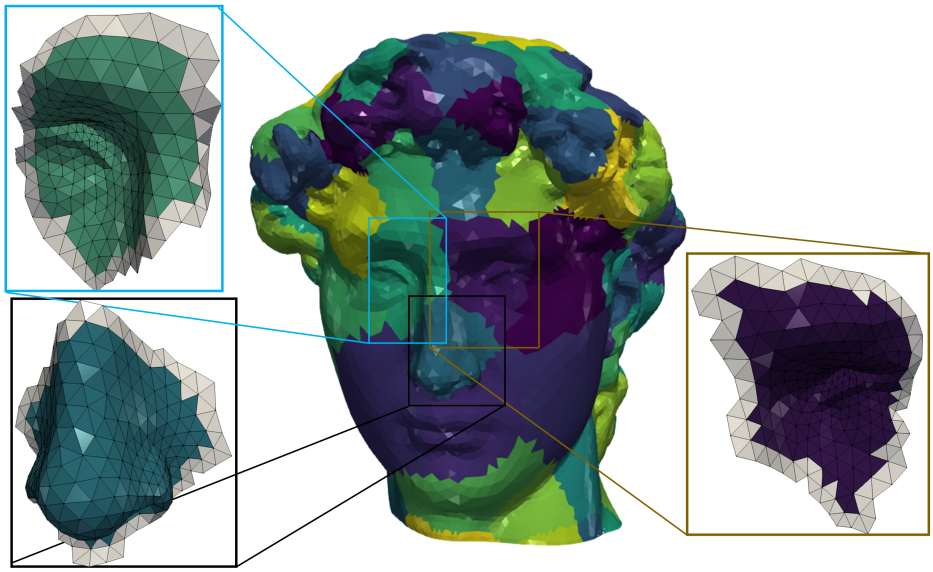


Figure 4.1: In RXMesh, we partition the mesh into small patches that fit in the GPU's shared memory. To eliminate the communication between different thread blocks, we augment each patch by a *ribbon* (shown in white).

In this chapter, we present *RXMesh*, a GPU static 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 dur-

[1]This chapter substantially appeared as "RXMesh: A GPU Mesh Data Structure" published at SIGGRAPH 2021 [66], for which I was the first author and responsible for most of the research and writing.

ing 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.

The contributions of this chapter are:

- 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 [19] data structure as well as (single- and multi-core) CPU-based frameworks. We achieve significant speedups over these frameworks.

RXMesh, as presented in this chapter, is limited to static applications that do not require changing the underlying mesh topology but may possibly change the geometric attributes of the mesh. In Chapter 5, we will show how to extend RXMesh for dynamic mesh operations.

4.1 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* [12]. 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 [43] and proved to be powerful, performant, and flexible.

From a user perspective, our programming model is similar to the "think like a vertex" (TLAV) [69] 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

```
__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 4.1: Vertex normal computation using RXMesh. Our parallel programming model abstracts away the details of assigning work to processors. 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.

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 4.1 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.

4.2 Goals and Design Principles

4.2.1 Goals

Given the programming model, we now describe a static triangle mesh data structure for the GPU that implements the queries in Table 2.1. 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.

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.

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-

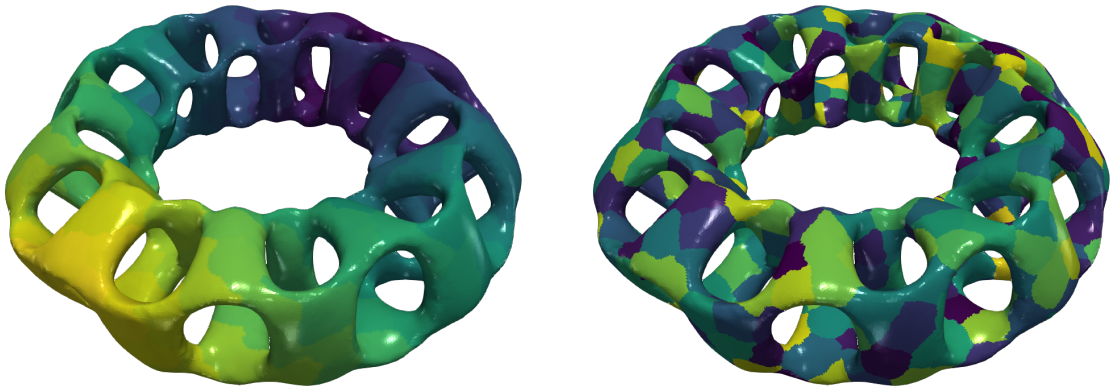


Figure 4.2: Color indicates the face index, highlighting the different ways of capturing locality; global sorting (left) and patching (right).

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.

4.2.2 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.

4.2.2.1 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 [49], using some kind of spatial information as the sort key (Figure 4.2). 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.

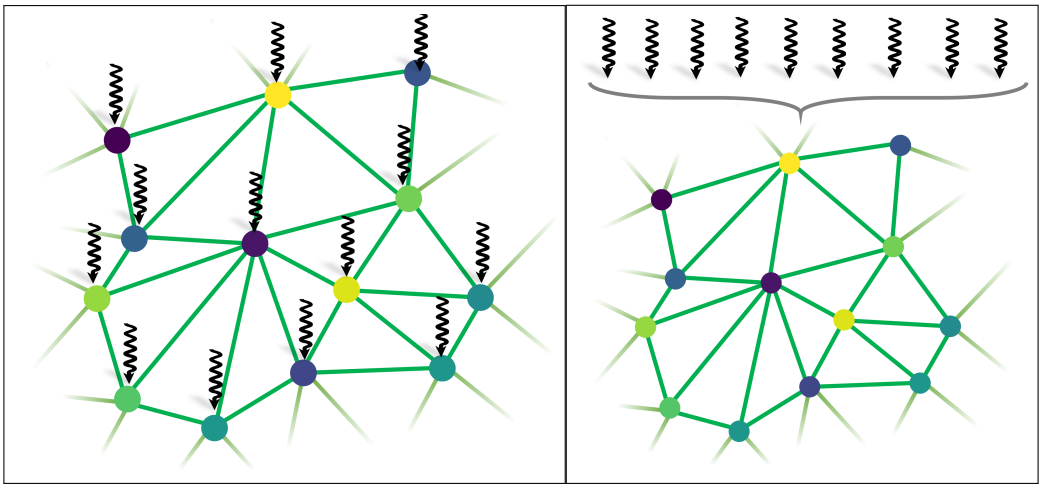
The L1 cache may also be rapidly exhausted if mesh attributes are queried simultaneously with topology queries, which is the common case. Thus, we implemented an alternate design: subdivide the mesh into small patches that can fit in the user-managed shared memory and perform all the computation/queries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

4.2.2.2 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 4.3). 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 4.3.2.

4.2.2.3 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



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

Figure 4.3: Directly assigning threads to mesh elements leads to load imbalance due to irregular mesh topology. Our programming model assigns mesh patches to blocks, enabling threads to cooperatively perform their queries, leading to well-balanced computation.

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 4.3.2.

4.2.2.4 Compact Patch Representation

Our top priority in choosing a data structure is supporting the operations specified in our programming model (Section 4.1). 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 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:

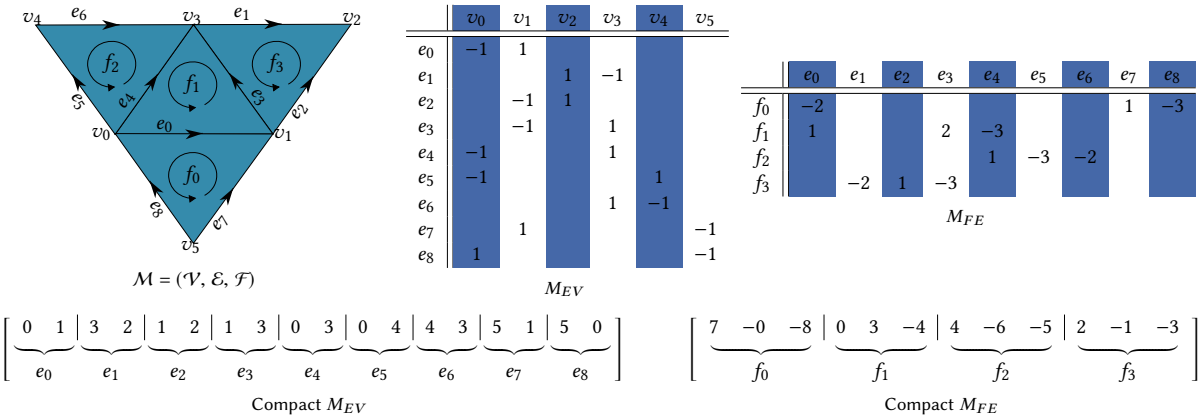


Figure 4.4: It is possible to represent all incident and adjacency relations in a mesh in terms of sparse matrices. Here we show two such relations—EV and FE. In our implementation, we only store the top-down incidence relation between each k-cell and $(k-1)$-cell. Exploiting the matrices' sparsity, we store them compactly in flat arrays where indices indicate the elements connectivity. We generate all remaining incidence and adjacency matrices from these two .

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 [34].
- 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, Linear Algebraic Representation (LAR) [32], 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 4.4). 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 4.3.2.

4.2.2.5 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 4.1). This moderately increases memory usage per patch (we quantify this in Section 4.4) 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.

Section 4.2.2.4 notes the importance of a compact representation to maximize the size of the patch that can fit into shared memory. The patch representation must store both the mesh elements "within" the patch as well as the patch's ribbon. Because the size of the ribbons scales only with the perimeter of the patch, maximizing the size of the patch also maximizes the fraction of storage that stores the patch's elements vs. its ribbon. Thus a more compact mesh representation better uses its storage to work on that patch's elements, which directly leads to higher performance.

4.3 Implementation Details

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

4.3.1 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. [99] 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 4.4.

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.

4.3.2 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 [4] 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 [6] and graph data structures [7] on the GPU.

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 4.1). 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.

4.3.2.1 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]

[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

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 4.1). 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.

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.

4.3.2.2 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 4.2.2.4) 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 4.2 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

```
__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 4.2: Computing FV using patch matrices in shared memory.

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 [5] 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 [5] 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 4.3 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.

4.3.3 Patching

4.3.3.1 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 [18]. 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. [18] 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 [49], to approximate 3D shapes [26], to simplify meshes [47], and for mesh parameterization [20]. What makes our problem unique is our requirement for small-sized contiguous patches, with a patch

```
__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 4.3: Multisplit-inspired matrix transpose where all threads in the block collaborate to carry out the computation to improve load balance.

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 [85] is a MPI-based multi-core parallel graph and mesh partitioning 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 [76]. 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.

4.3.3.2 Patching Algorithm

Overview: 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 [61], which is a highly parallel process to partition a given graph.

Given an undirected 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

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

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.

Implementation Details: 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. 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 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.

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 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.

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

4.4 Evaluation

We evaluate our data structure and programming model on both fundamental query operations and full applications in Section 4.5. 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 [101] and Smithsonian [90] 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 [19]. 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).

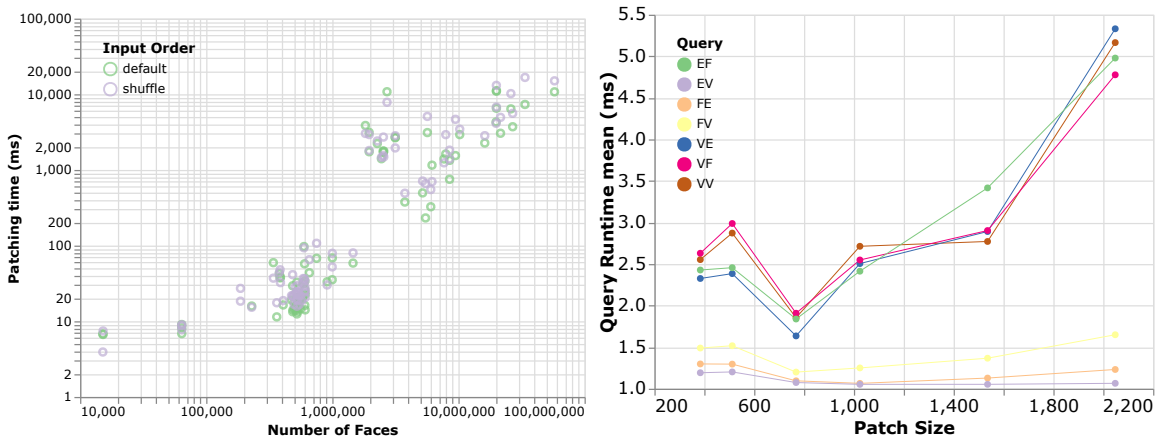
2. OpenMesh (Version 8.1) [12] and CGAL [50], the state-of-the-art CPU mesh libraries. We compare against both serial and OpenMP-parallel implementations (with the thread count set to `omp_get_max_threads` [40 on a DGX machine]).

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

Figure 4.5: Assortment of RXMesh patches

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 4.6a 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 [85] on 8 input meshes containing 16–57M faces. On average, our patching technique is $\sim$4.1$\times$ 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 4.6b. We choose a



(a) Time to patch

(b) Patch size

Figure 4.6: We show different aspects of the patching process; a) timing performance of the patching algorithm on all inputs used in this chapter using default and shuffle input order, b) experimenting with different patch size on different queries

maximum patch size of 768 faces for all experiments in this chapter 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 4.5 shows examples of the output patches on a few selected inputs.

4.4.1 Query Operations

We evaluate the performance of RXMesh on the eight first-order query operations listed in Table 2.1. 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

Figure 4.7: We consider three types of input order (from left to right): default, shuffle, and sorted.

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 (shown in Figure 4.7) 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 4.8). This performance gap motivates the development of a general-purpose mesh data structure on the GPU. Table 4.1 shows the speedup comparison between RXMesh and PDE of the three variants. We additionally report the timing

for all methods in Figure 4.8. 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 $\sim$3.49$\times$ 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

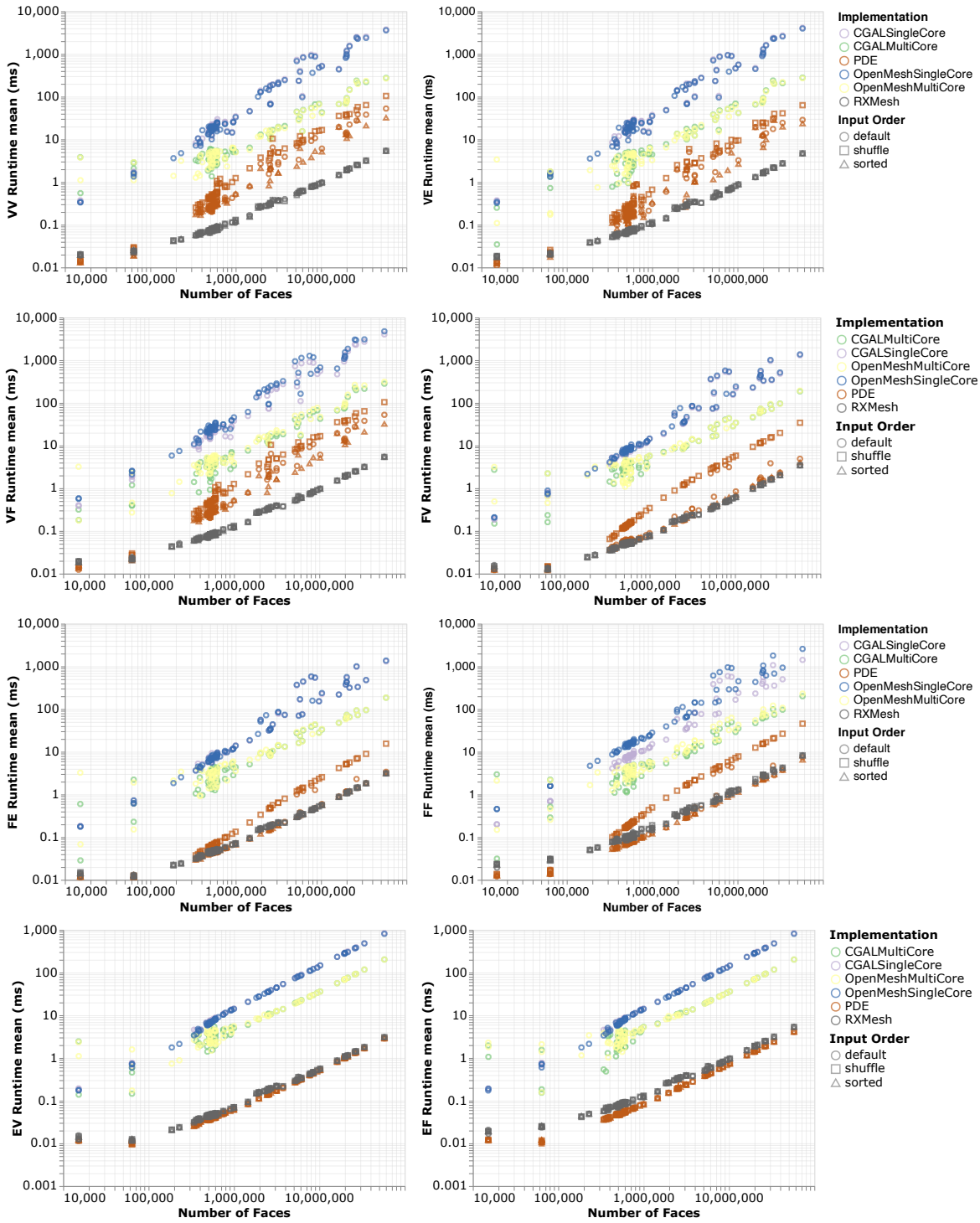


Figure 4.8: Results of the first-order queries using RXMesh, PDE, OpenMesh, and CGAL. Each query shows runtime vs. number of input faces on the default, sorted, and shuffle input mesh ordering.

Table 4.1: 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

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 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 $\sim$1.28$\times$ for sorted inputs. For the other queries, RXMesh's speedup over PDE is on average $\sim$2.58$\times$. 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: 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. PDE requires 42 bytes/face, i.e., $\sim$8% less memory

than RXMesh. However, 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 $\sim$19% savings in global memory vs. PDE.

4.5 Applications

In this section, we put the RXMesh system to work on a set of 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 4.5.1) integrates significant computation that repeatedly queries the vertices' one-ring exemplifying many applications that have a similar pattern of computation. Geodesic distance (Section 4.5.2) 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 4.5.3) 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 4.5.4) 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.

4.5.1 Mean Curvature Flow

Geometry processing applications that require solving a linear system of equations are ubiquitous. Examples include mesh editing [98], mesh parameterization [74], and simulation [75]. 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 [30], which is an effective method to remove the high-frequency noise from an input mesh (Fig-

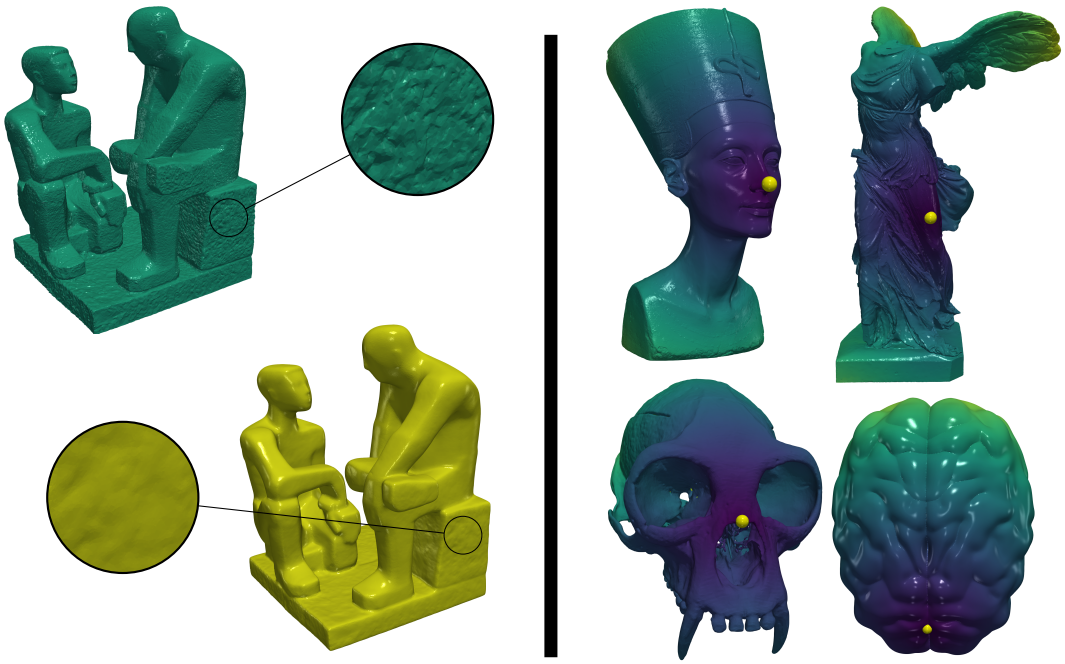
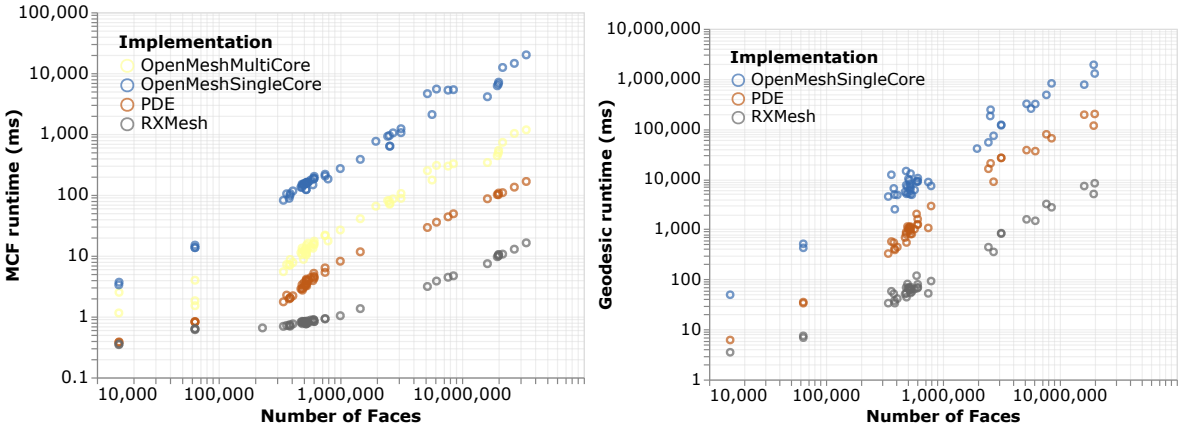


Figure 4.9: Examples of removing high-frequency noise by smoothing using mean curvature flow (left) and computing geodesic distance (right).

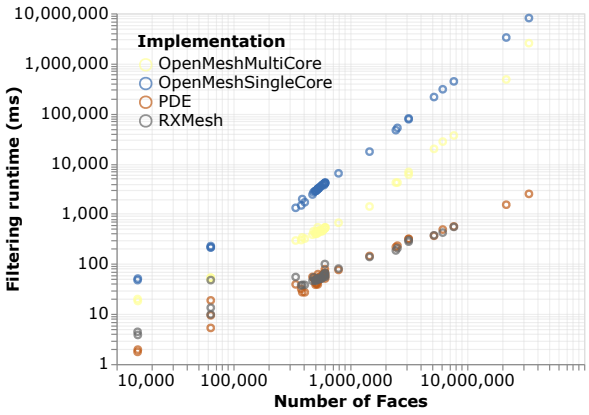
ure 4.9). Our implementation uses an (un-preconditioned) conjugate gradient (CG) solver [88] using cotan weights computed at every iteration on-the-fly using VV queries. Given RXMesh's fast neighborhood queries, we can directly perform the necessary matrix-vector computations and make use of the easily parallelizable matrix-free iterative CG solver.

For comparison, we implemented the matrix-free CG using (single and multi-core) OpenMesh and PDE. Figure 4.10a 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

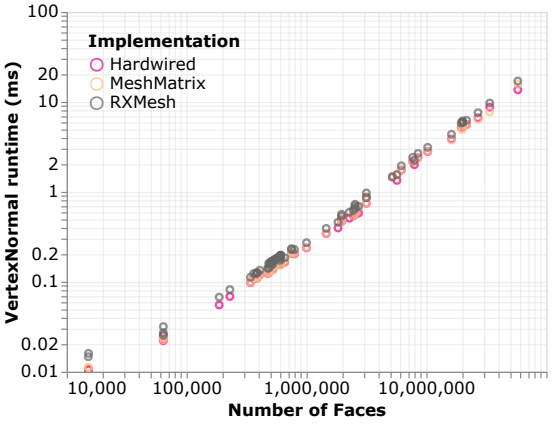


(a) Mean curvature flow

(b) Geodesic distance



(c) Bilateral filtering



(d) Vertex normal

Figure 4.10: Timing performance of RXMesh, PDE, and OpenMesh on four different applications. All applications run on sorted input.

slower performance.

4.5.2 Geodesic Distance

In this application, we compute the geodesic distance from a single source vertex to all other vertices (Figure 4.9). Our implementation is based on the front propagation for computing approximate geodesic distance using the minimalistic parallel algorithm by Romero et al. [83]. 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 4.1). 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 4.10b 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.

4.5.3 Bilateral Filtering

We implement an additional denoising application based on Bilateral Mesh Denoising (BMD) [36] 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 4.10c) 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.

4.5.4 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 [68] using RXMesh and compare against a hardwired, 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 [99], as outlined in their paper.

Listing 4.1 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 4.10d); 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.

4.6 Summary

In this chapter, we discussed the design and implementation of our first step towards efficient local mesh operations on the GPU, i.e., local *static* mesh operations. In this design, we capitalized on the programmer-managed caching to leverage mesh locality in order to achieve best-in-class performance across various query operations, and more importantly, across different applications. We also presented a simple programming model that facilitates application authoring without extensive parallel computing knowledge from the user. In the next chapter, we will show how we can extend both the data structure and programming to allow—for the first time—generic local dynamic update operations on the GPU.

Chapter 5

Dynamic Triangle Mesh Processing[1]

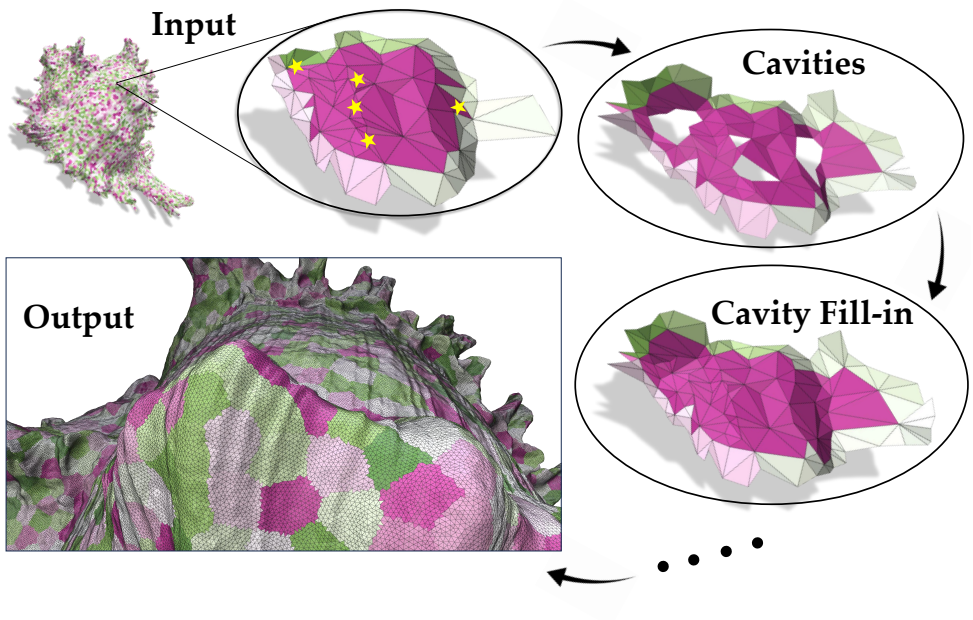


Figure 5.1: Our system casts local dynamic mesh operations as *cavity operators*, which provide intuitive semantics for mesh updates (e.g., edge collapse here, during one iteration of isotropic remeshing). With the cavity operator, the user defines mesh updates by creating cavities and then fills in these cavities with new elements while our system handles potential conflicts.

In this chapter, we present a new dynamic mesh processing system that operates entirely on the GPU. Our system leverages the GPU's parallelism for high-performance local dynamic triangle mesh updates by tackling the following challenges:

[1]This chapter substantially appeared as "Dynamic Mesh Processing on the GPU" currently under review for SIGGRAPH Asia 2024 [65], for which I was the first author and responsible for most of the research and writing.

1. **Locality**: Since we focus on dynamic local operation, the ideal implementation will take advantage of the locality of accessing and changing the mesh data structure on the GPU.

2. **Conflict Handling**: Conflict handling involves two related challenges. First, we need a data structure that can detect if two (or more) operations conflict, i.e., applying them simultaneously will lead to an invalid mesh. Second, we need a data structure that can resolve conflicts, i.e., given two (or more) conflicting operations, the data structure should decide on which subset of these operations should proceed and how.

3. **Compactness**: A mesh data structure must satisfy two conflicting demands. On one hand, the limited GPU memory favors lightweight data structures, since manipulating such a data structure requires fewer memory transactions. On the other hand, lightweight data structures offer limited information about conflict detection and resolution.

4. **Scheduling**: With serial execution, conflict resolution is straightforward. However, high performance requires maximizing parallelism, and thus the need to process conflicts in parallel. Our philosophy is that the data structure is responsible for detecting and resolving conflicts, maintaining a valid mesh at all times, and the scheduler maximizes parallelism given the correctness constraints imposed by the data structure.

In addressing these challenges, we design a system that achieves the following design goals:

1. **Efficient Incremental Mesh Updates**: Our system's primary goal is enabling high performance for *incremental* triangle mesh updates on large meshes on the GPU. We do this by avoiding CPU-GPU data transfer, maximizing GPU memory locality, improving load balance, and reducing thread divergence. Our system sets the bar for dynamic mesh processing on the GPU and delivers an order-of-magnitude better performance compared to state-of-the-art multithreaded CPU alternatives.

2. **Efficient Static Performance**: Our system does not compromise the performance of static applications for the benefit of dynamic applications. On static applications, our system could deliver better performance than RXMesh (Chapter 4).

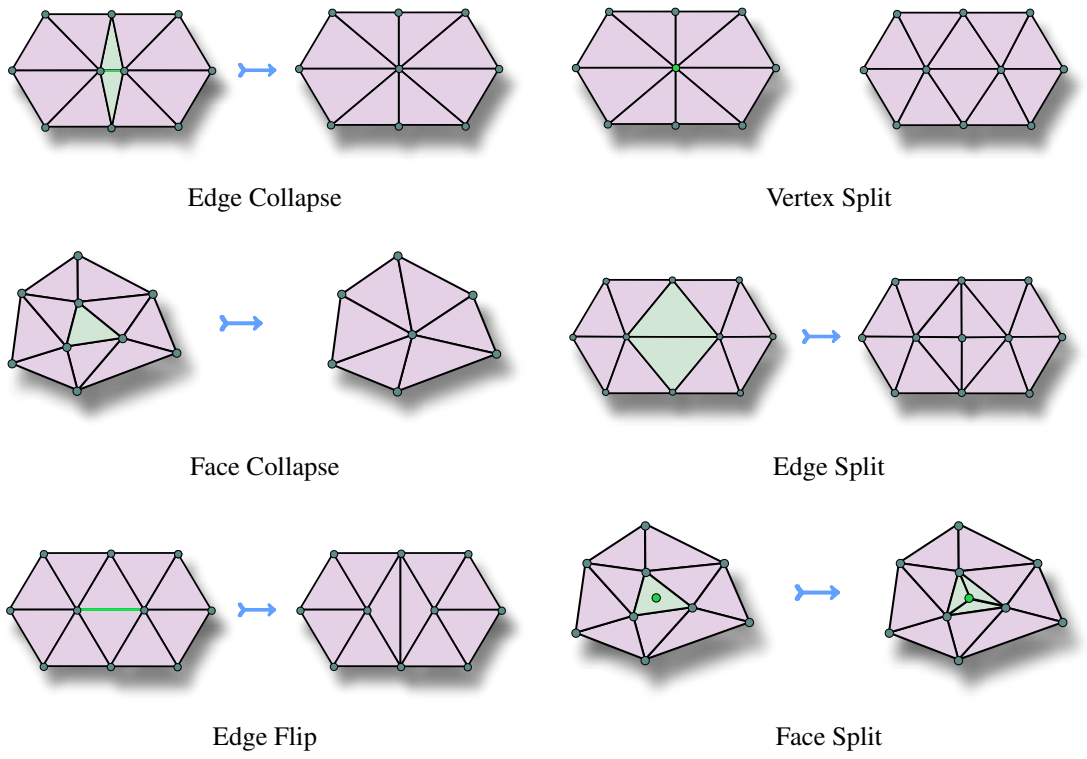


Figure 5.2: Examples of dynamic triangle mesh local operators.

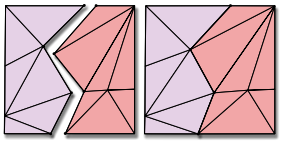
3. **Intuitive Semantics**: Our system provides intuitive concise semantics for mesh updates and for resolving conflicts. Our design considers both the topology and geometry (i.e., attributes) of the mesh, liberating the user from low-level intricate implementation details.

4. **Robust Update Operations**: Our system handles generic triangle meshes without hard requirements on mesh quality, i.e., non-manifoldness or orientability. Our system does not impose any requirements on the type of dynamic operations—so long as they have a local area of impact. We support almost all common mesh operators found in open-source libraries (see Figure 5.2 for examples). Our system is also extensible and allows the user to implement new operations.

5. **Compact Mesh Data Structure**: The data structure used in our system is compact to ensure that users are not limited to small input meshes due to limited GPU memory. Our data structure needs 2× less memory than RXMesh. This compact data structure requires less bookkeeping and exhibits greater locality, both leading to higher performance.

Using our system, we implemented three dynamic geometry processing applications, i.e., surface tracking, uniform isotropic remeshing, and Delaunay edge flip. In comparison with the state-of-the-art multithreaded CPU framework, our system speedup is between 2–86× on large meshes with millions of faces and more than two order of magnitude faster than single-threaded CPU solutions. Besides memory efficiency, our data structure outperforms RXMesh with a geometric mean speedup of 1.5×.

Non-goals: Our system supports applications that rely on incremental mesh updates and aims to set the baseline for enabling such applications fully on the GPU in a generic way. However, our system does not support applications that alter the whole mesh in one step, e.g., mesh subdivision [71], nor does it support operations with non-topologically local area of impact for update operations, e.g., mesh "zippering" [16] (see inset). Additionally, our system does not have inherent support for (partially) ordered update operations and relies on the user to manage the order of updates.



Mesh zippering

5.1 Programming Model

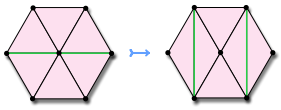
Goals: We begin by describing what we believe are the important attributes of a programming model for GPU dynamic mesh processilng:

- Allow generic mesh update operations that have local area of impact.
- Separate operations on meshes (the "what") from the implementations of those operations (the "how"). The programming model should hide internal data structure details.
- Have an intuitive interface for the user to reason about conflicting operations.
- Maximize the number of concurrent update operations that can be executed concurrently.
- Propagate dynamic topological changes to mesh attributes. These attributes could be associated with vertices, edges, or faces.

Next, we will discuss different alternative designs for such a programming model to better motivate our programming model.

Alternative design: A collection of low-level operations: The predominant programming model for dynamic mesh updates relies on defining local operations, e.g., half-edge data-structure-based systems such as PMP [14] and CGAL [50]. These systems offer the user a set of basic dynamic operators (e.g., edge flip, vertex split) that the user could compose to implement a dynamic application. However, the user here is limited to the set of operators offered by these systems. We surveyed all major dynamic mesh processing libraries (PMP [14], CGAL [50], OpenMesh [12], Wild Meshing Toolkit (WMTK) [46], libigl [45]) and found no consistency in the set of operators that they offer—each library is missing at least one core operator offered by another. This inconsistency argues against implementing a similar programming model on the GPU because of the lack of portability to other existing (CPU-based) systems.

Another problem with building a system that relies on a fixed set of dynamic operators is conflict handling. These operators do not provide an out-of-the-box conflict handling mechanism because they did not target parallel processing in their design. The straightforward solution to enabling parallelism is to lock a local neighborhood during processing. For example, WMTK [46] (which leverages the parallelism of multi-threaded CPUs) locks the entire two-ring neighborhood of an edge for any edge-based operation, e.g., edge flip (see the inset). This approach might be justified for a limited-parallelism environment like a multithreaded CPU. However, on the GPU, with its many thousands of parallel threads, the extensive locking of neighborhoods for each update operation significantly restricts the potential for parallelism, leading to severe underutilization of the GPU and hence lowered performance.



While the two green edges could be flipped concurrently, Jiang et al. [46] locks the one-ring of the edge's vertices leading to serializing these two independent edge flips.

To mitigate the contention problems from overlocking, we could consider a system where the user implements local operators where the locking region is user-defined. For example, not all dynamic mesh libraries offer $1 \rightarrow 3$ triangle split operations, but a user (who knows the details of the underlying data structure) could implement it. The main issue with such a design is the locking region must be defined based on the internal data structure. For example, in the $1 \rightarrow 3$ triangle split operator, users might assume that they do not need to lock the vertices of the split

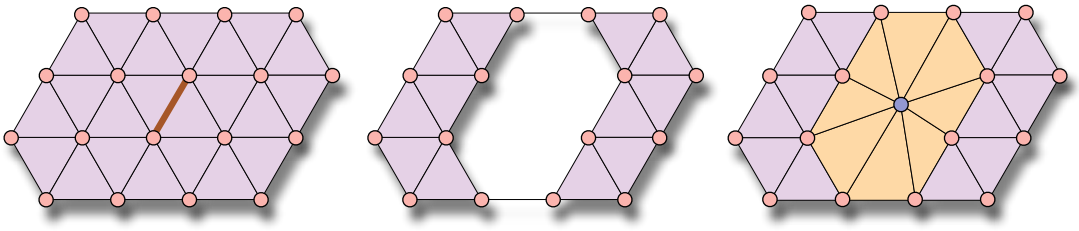


Figure 5.3: An example of edge collapse, cast as a cavity-based operator.

triangle. However, if the data structure stores topological information per vertex, not locking those vertices may lead to race conditions and result in an invalid mesh. Thus, such a system requires exposing its internal data structure implementation details to the user, violating our design goal of separating the concerns of mesh operations from the underlying implementation.

Our programming model: Given these difficulties, we choose a different abstraction for our programming model. To support dynamic operations, we choose the *cavity operator* [64] as our fundamental abstraction. A *cavity* is a set of vertices, edges, and faces that forms a single connected component such that removing this set creates a single hole in the mesh. The *cavity operator* is a universal operator that encompasses all local dynamic mesh operators (Figure 5.2). The cavity operator defines any mesh update operation as *element reinsertion* by removing a set of mesh elements and inserting others in the created void. The cavity operator splits a local mesh update into two operations: cavity creation and cavity fill-in. First, *cavity creation* removes a mesh element and its incident/adjacent elements effectively creating a hole/cavity in the mesh. Then, *cavity fill-in* covers the cavity by optionally adding mesh elements inside the hole. Figure 5.3 shows an example of edge collapse operations cast as a cavity-based operation. While the cavity operator was originally proposed for metric-based anisotropic mesh adaption on serial and multithreaded CPUs, we generalize it and use it as the basis of our programming model. More importantly, we use the cavity operator as an intuitive interface for conflict detection. Cavities create a simple mental model that a user can use to reason about conflicts: overlapping cavities lead to conflicting operations. We further use cavities to resolve conflicts (Section 5.3). The cavity-based operator provides the right ingredients for an extensible frame-

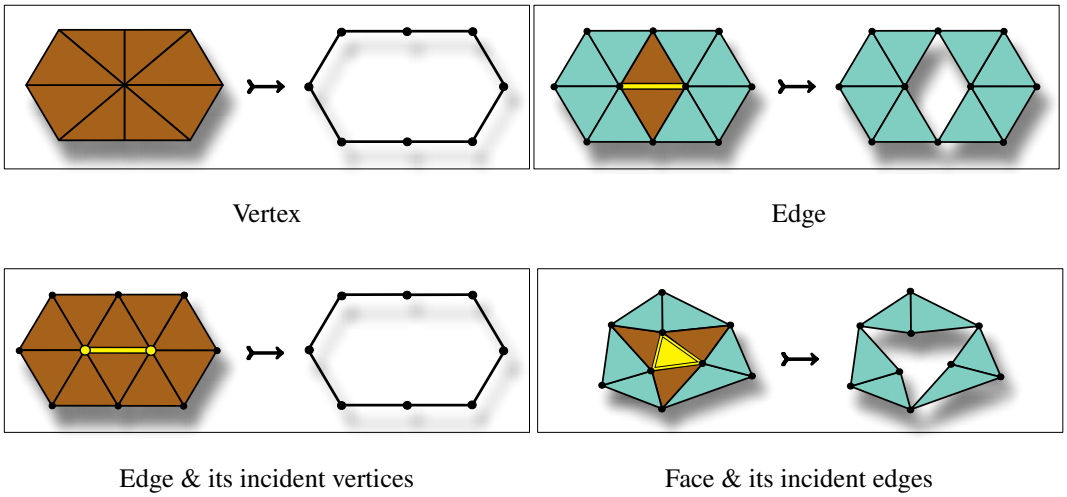


Figure 5.4: Examples of cavity templates provided by our system. The seed for each cavity is highlighted in yellow and the deleted neighborhood is shown in brown.

work, lowering the cost of maintaining the system without limiting the user to a predefined set of operators. It is possible to easily cast all major local operators (e.g., edge collapse, vertex split, edge split, edge flip, face collapse, face split, and delete vertex/edge/face) as cavity-based operations. Cavity fill-in allows the user to expand beyond the traditional dynamic operator, e.g., instead of adding a single vertex during Figure 5.3's cavity fill-in, the user may instead add three vertices to create a more refined mesh.

Separating the update operation into two steps allows the system to present an intuitive model of conflicting operations to the user. The user first declares a set of cavities. Then the underlying system detects conflicting cavities and only proceeds with a subset of the cavities that are conflict-free. The user, then, fills in this subset of cavities. Our system maximizes the set of conflict-free cavities, e.g., the two edge flips in the inset above can be done concurrently since their cavities do not overlap. Thus, separating update operations into two steps allows us to have a simple interface for conflict handling without exposing any internal data structures or requiring the user to reason about locking.

To create a cavity, the user needs to add one or more mesh elements to the cavity. For example, for a vertex split operation, the cavity will be the vertex and all its incident edges and faces. A single cavity could be declared by a single or multiple threads; however, commonly

it is easier to declare a single cavity using a single thread. To facilitate cavity declaration, our system optionally offers a set of predefined *templates* that resemble the common mesh update operations. Each template consists of a *seed* and a neighborhood around it that will be deleted. The seed could be any type of mesh element, i.e., vertex, edge, or face. The neighborhood to be deleted is defined in terms of incidence/adjacency relation on the seed. For example, a template used for edge flip has the edge as a seed and the faces incident to the edge as the neighborhood. With these templates, the user can create a cavity by only specifying the seed and our system handles assigning the local neighborhood to the cavity. Figure 5.4 shows a subset of templates that our system offers. Note that using these templates is entirely optional.

The underlying system resolves conflict by calculating a maximal independent set of non-conflicting cavities which maximizes the number of cavities/update operations that can be processed concurrently. This set is returned back to the user which can be processed in parallel. For each cavity in the set, our system offers an iterator to retrieve the edges and vertices of the cavity's boundary. Using this iterator, the user can iterate over the cavity boundary edges/vertices and connect them with new vertices, edges, or faces that the user adds into the interior of the cavity. Additionally, the user can access the old connectivity of the created cavity. This aids the user in creating fill-in that may require old information from the cavity (e.g., calculating the mid-point of a collapsed edge). Our system also handles attribute allocation and facilitates accessing attributes of deleted cavities during fill-in.

5.2 Design Principles

The core of our system is the combination of data structure and scheduler that together allow us to implement the programming model (Section 5.1) while achieving our design goals. Here we discuss the design principles we followed in designing the data structure (Section 5.2.1) and the scheduler (Section 5.2.2). Finally, we discuss how the whole system works (Section 5.2.3).

5.2.1 Data Structure

Maximize Locality: In a design where all mesh data is stored in global GPU memory, mesh-based operations are mostly out-of-cache. Topological query operations involve multiple levels of memory indirection, frustrating attempts at exploiting locality. Geometric information (i.e.,

mesh attributes) is hard to coalesce when neighbor attributes are accessed, leading to irregular memory accesses. As we showed in Chapter 4, we can mitigate this problem by partitioning the mesh into small *patches* that fit into the GPU's small but fast shared memory, which additionally does not require coalesced access to achieve high bandwidth. We follow a similar approach here, which helps localize both query (static) and update (dynamic) mesh operations. For static operations, our system is similar to RXMesh, except for how we access ribbon information and how we localize accessing geometric attributes (see Section 5.3.1 for more details). With this design, once a CUDA block is assigned to a patch, the block operates on the patch and performs all update operations by reading from/writing to shared memory. Once complete, the block commits the updated patch to global memory. This way, reading and writing the patch requires only one coalesced patch-sized read and write to global memory. The majority of update operations that require irregular memory access happen in low-latency, high-bandwidth shared memory.

Optimistic Parallelism: Processing patches from shared memory creates two copies of the patch: a working copy in shared memory and the original in global memory. This opens the door for *optimistic parallelism* [55], as we will discuss in Section 5.2.2. From a data-structure perspective, optimistic parallelism requires that the data structure has a cheap way to roll back its updates when the scheduler detects a conflict. Since all updates happen in shared memory, rollback is simple and no-cost: discard the changes in shared memory. This strategy is enabled by CUDA's explicit programmer-controlled shared memory. Such a design principle is unique to our data structure and system that, to the best of our knowledge, has not been exploited in other optimistic parallelism systems (e.g., Galois [55]).

Trading global memory writes for reads: In our design, conflicting updates within the patch can be easily detected and resolved. At any instant in time, a mesh element may be part of no more than one cavity. If more than one cavity aims to incorporate one particular element, this causes a conflict and one cavity must be deactivated. The restriction that each mesh element must belong to a single cavity is enforced by our system. This constraint can be managed either by the system itself or by the user (see Section 5.3.2).

Cavities that cross a patch boundary pose a challenge since detecting and resolving their

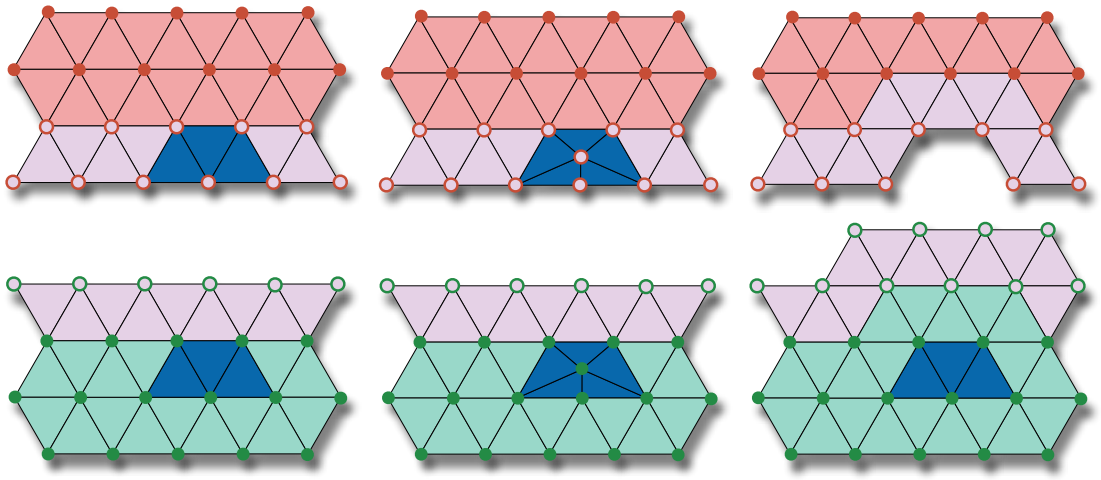


Figure 5.5: Inter-patch conflicts between two patches (top q, bottom p), where ribbon elements are shown in a lighter color. Conflicts occur when a cavity (bottom blue) has an imprint on a neighbor patch (top). One way to resolve the conflict is to lock q and update both patches (middle). We choose to instead remove the imprint from q (right) by deactivating some of its mesh elements, leading to reduced memory accesses because we do not write cavity fill-in in q.

potential conflicts requires coordination across patches and thus global memory accesses. To maximize performance, these memory accesses should be kept to a minimum, and to be as coalesced as possible. Consider a patch p with a cavity that has an *imprint* on a neighbor patch q. We considered an approach that resolves possible conflicts by locking q while processing p and then re-applying the cavity fill-in on both p and q (Figure 5.5, middle). This approach had the disadvantage of locking q for an extended period, which limits parallelism. More importantly, applying fill-in led to more writes to global memory. We instead chose to *expand* p such that the entire cavity falls inside p and has no imprint on q. Compared to the first approach, our choice reduces write operations and increases reads. Expanding a patch simply deletes a few mesh elements from q, which amounts to flipping bits in a bitmask (see Section 5.3), and can be easily coalesced. In contrast, the first approach required writing topological information (i.e., face and edge connectivity) which requires more memory transactions.

Amortized Locking: The above strategy for inter-block cavities localizes all changes for any cavity within the patch that contains that cavity. Patches are large enough, and cavities are small enough, that one patch may contain multiple cavities that require cavity operations within a single patch. In this case, we can aggregate all requests to modify a neighbor patch, lock the

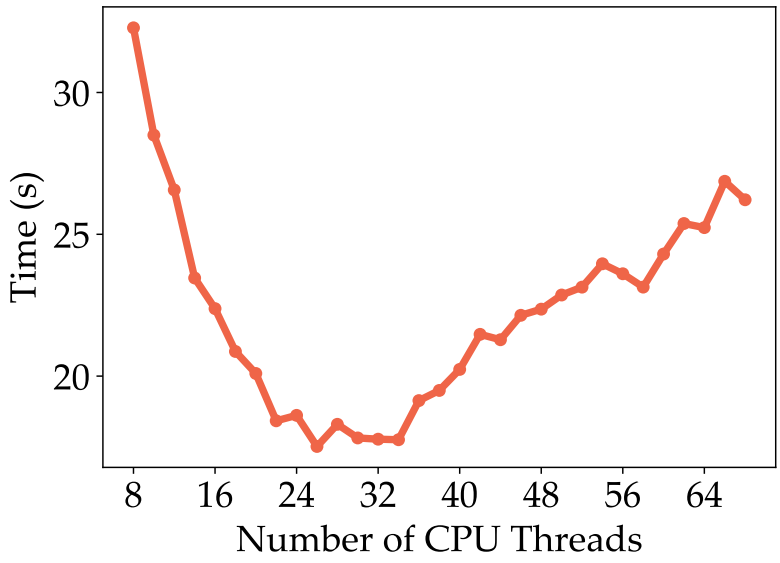


Figure 5.6: Strong scaling of WMTK [46] on Delaunay Edge Flip application

neighbor patch, and then satisfy all these requests at once, thus amortizing the cost of locking the neighbor patch. This design prioritizes throughput over latency, which is a good match for the throughput-oriented GPU hardware. In contrast, locking small neighborhoods around each operator/cavity scales poorly even for hardware with more limited parallelism like a multithreaded CPU. Figure 5.6 shows the result of running the Delaunay Edge Flip application (Section 5.4.2) on a model with 2M faces on a 64-Core AMD EPYC 7742 while varying the number of threads (i.e., strong scaling). As the number of threads increases, locking and unlocking becomes the dominant cost, overshadowing any benefits of increased parallelism. A GPU requires considerably more parallelism than hundreds of threads; reducing the cost of locking is critical to achieve good performance.

5.2.2 Scheduler

In the following discussion, we use CUDA terminology, but we believe our scheduler design will be applicable to other programming languages that offer a similar level of flexibility as CUDA, i.e., access to the GPU's shared memory. The main responsibility of a scheduler in our system is to manage how and when patches are assigned to compute resources. In the case of static GPU mesh processing, we relied on the GPU hardware scheduler, i.e., we assign each

patch to one thread block and then the hardware scheduler assigns thread blocks to the GPU's streaming multiprocessors (SMs). Once an SM finishes processing one thread block (patch), the hardware scheduler assigns another thread block (patch) to it, continuing until all patches have been processed.

As we noted above, our system processes cavities that may cross patch boundaries. In this case, we expand one patch to remove the dependence on the other patch, but this requires coordinating across both patches. This strategy is potentially problematic if both patches are simultaneously scheduled and are executing on different GPU SMs (the resulting conflicts may result in incorrect output). Because the hardware scheduler has no knowledge of patch dependencies, we have no easy way to avoid this situation. Thus we turn to implementing our own scheduler and next we discuss four potential design choices.

(1) Serialization: One potential scheduler-based solution to conflicts is to serialize conflicting updates. Such a solution might be fine for limited-parallelism environments such as a desktop multithreaded CPU, where a relatively few number of patches can be executed concurrently. However, a modern GPU features more than 100 independently running SMs. Thus, serializing conflicting patches will lead to idle SMs and a loss of performance.

(2) Two-Phase Approach: All cavities that are wholly within the interior of a single patch can be processed concurrently. Thus we can consider a scheduler that alternates between processing interior cavities and boundary elements. In such a two-phase approach, the first blocks process elements in the *deep* interior of the patch (i.e., those not incident to the ribbon elements). Then, in the second phase, ribbon-element processing is done serially after a global synchronization. This approach works well for large coarse-grained partitions/patches that are suitable for CPU multicores [62] where the interface between partitions is relatively small. In our system, however, the patch size is small enough to fit in shared memory, thus the interface between patches is correspondingly larger, and this approach serializes a large fraction of the work. Thus, this approach has the same disadvantage as serializing conflicting updates.

(3) Graph coloring: Since conflicts are only between neighbor patches, we could potentially use graph coloring to generate an independent set of patches that can be processed concurrently. We could define a graph where each patch is a node and neighboring patches share an edge.

After coloring this graph, we could process all patches of one color in parallel without any conflicts, and sequentially process colors. However, such an approach becomes too expensive in dynamic workloads where we update the connectivity of the patches, which would require a new coloring on each update.

(4) Speculative Processing: Our final alternative, and the one we chose, is speculative processing, which Kulkarni et al. [55] called "the only plausible approach to parallelizing many, if not most, irregular applications". This strategy allows processes/threads to execute independently without synchronization with other threads. If a conflict is detected, the process/thread rolls back to a conflict-free state and then continues execution. Since we process patches in shared memory, we require no synchronization for processing, i.e., two possibly conflicting operations in two different patches could both proceed since they both operate in separate shared memories. Instead, synchronization is only necessary before *committing* updates to global memory (and then only for updates that impact both a patch and its neighbor). Of all alternatives above, this design has the least overhead for synchronization, and thus has the potential to exploit the most parallelism.

Speculative processing, however, has three costs. The first is the cost of *detecting conflicts*, which happens whenever a cavity crosses the patch boundary, requiring that the neighbor patch not be processed concurrently. However, this cost is inevitable and would be paid for other alternative designs as well. The second is the cost of *rolling back*. In our design, this cost is low, as it only requires discarding changes in shared memory (Section 5.2.1) for conflicting updates without any impact on the global state of the data structure. The third cost is the *work that must be discarded*. The ample computational resources of a modern GPU, in general, motivate reducing synchronization costs (that limit parallelism) even at the cost of more work. While we do not have a theoretical bound on the latter cost, we show empirically in our system that discarding work is infrequent, and decreases proportionally with larger meshes. Figure 5.7 shows the ratio of discarded patches vs. the number of scheduled patches for different input meshes. We use a pathological corner case representing the worst-case scenario, where each patch depends on all of its neighboring patches. With the smallest meshes, the cost is high ($\sim$60%) since the number of patches that can be executed concurrently is low. As the mesh size

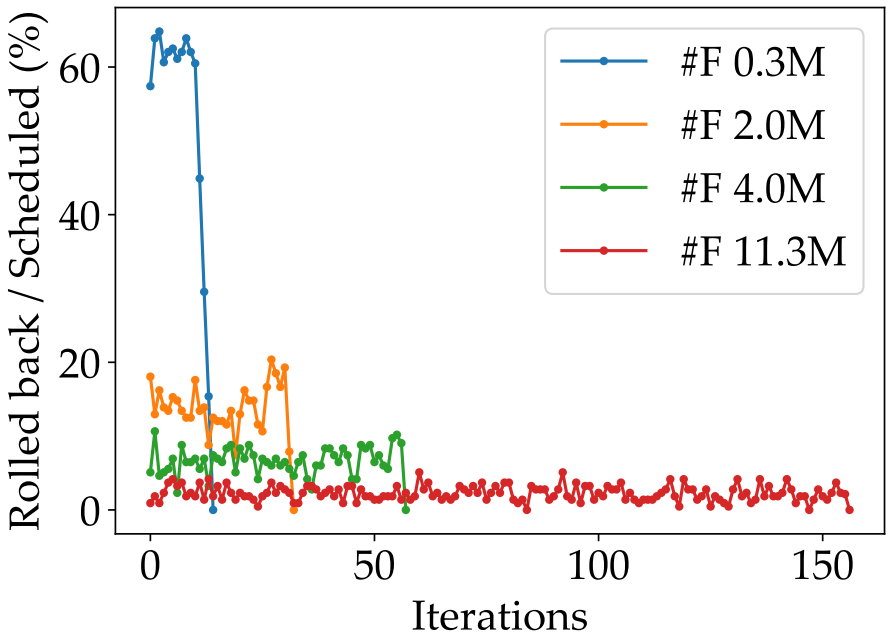


Figure 5.7: We measure the cost of speculative processing by measuring the ratio of rolled-back work (due to conflict) with respect to scheduled work, i.e., work that has been done in shared memory but not saved in global memory yet. Here we assume that every patch must lock *all* its neighbor patches (e.g., to update them). The overhead of wasted work decreases significantly as the size of the mesh increases.

increases, the fraction of wasted work becomes negligible (less than 2%) and thus, the overall cost of speculation is low.

5.2.3 Putting it All Together

Finally, we discuss the high-level architecture of our system and how we implement the aforementioned design principles, deferring detailed implementation aspects to the next section. The main controller of our system is the scheduler, which operates in a loop on the CPU. In each iteration, the scheduler launches a GPU kernel with as many blocks as possible to maximize GPU occupancy while avoiding the simultaneous processing of neighboring patches. This strategy minimizes the cost of speculative processing. The loop terminates when all patches in the mesh have been processed.

Within each iteration, the scheduler assigns blocks to patches. Figure 5.8 provides an overview of the tasks performed by each CUDA block during an iteration. Each block at-

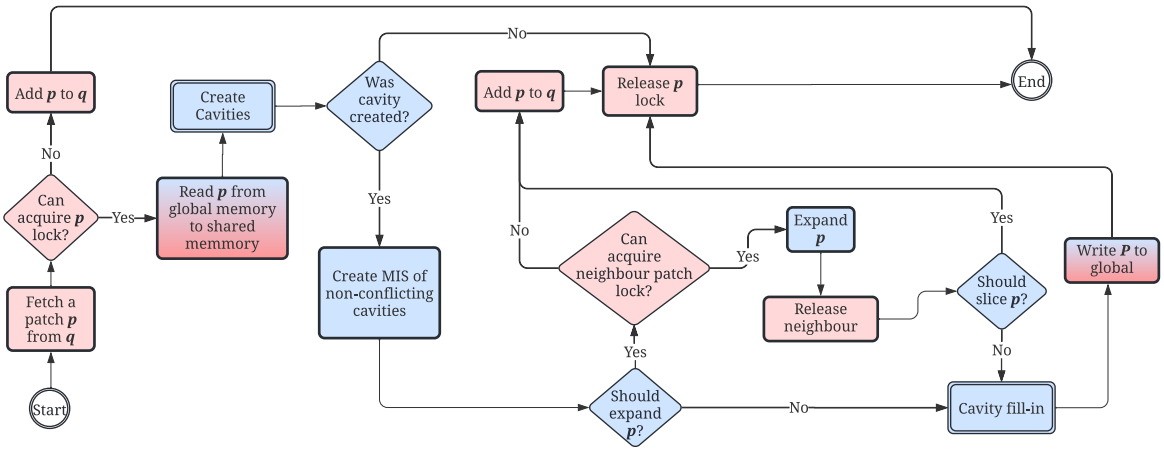


Figure 5.8: An overview of our system. On the CPU, we iteratively launch kernels until all patches are processed, indicated by an empty (queue-based) scheduler (q). The flowchart illustrates the operations performed by a single CUDA block on the GPU. Red blocks represent memory operations on global memory, while blue blocks represent memory operations on shared memory, and gradient blocks are for memory transfers from/to global memory to/from shared memory. Double-boundary blocks indicate user code (i.e., Create Cavities and Cavity fill-in); other blocks are within our system and are not user-visible. The process begins with fetching a patch (p) from the scheduler. If the block successfully acquires p's lock, it proceeds by transferring p's information from global memory to shared memory. Otherwise, we reschedule p and exit. Next, the user initiates cavity creation. If at least one cavity is created within p, we form a maximal independent set (MIS) of conflict-free cavities. If a cavity necessitates expanding p to prevent inter-patch conflicts, we try to acquire the neighbor patch lock. If successful, p is expanded; otherwise, p is re-added to the scheduler, and we exit. During the expansion of p, if slicing is required, we mark p for slicing, we release its lock, and we exit. If no slicing is needed, the user proceeds with cavity fill-in. Finally, we write p back to global memory, release its lock, and exit.

tempts to acquire the lock for its assigned patch. If unsuccessful, the block exits. Once the lock is acquired, the block reads the patch information from global memory to shared memory. This information can also be used for static query operations necessary for evaluating predicates, e.g., checking if an edge is short enough for an edge collapse.

The threads within the block then call a user-defined (or system-defined for predefined cavity templates) function to create cavities within the patch. Our system first resolves intra-patch conflicts by ensuring that a mesh element belongs to a single cavity by creating a maximal independent set of conflict-free cavities within the patch. Unsuccessful cavities are marked accordingly, allowing the user to attempt them in subsequent iterations.

Next, our system resolves inter-patch conflicts by checking for cavities that imprint on

neighboring patches. If at least one cavity has an imprint on a neighboring patch, the entire block cooperates to expand the patch. Expanding a patch requires locking the neighboring patches before reading from them. If locking fails, the patch is discarded and rescheduled for subsequent iterations. If successful, the patch is expanded, and the information is written in shared memory. After a successful expansion, the neighboring patches are unlocked, allowing them to be processed by other blocks. Finally, the block performs the user-specified cavity fill-in before committing the patch to global memory.

5.3 Implementation Details

5.3.1 Patch Data Structure

Mesh Topology: As mentioned in Section 5.2.1, we partition the mesh into small patches that fit in shared memory. Here, we discuss how we represent patch information and how we perform operations on it. Similar to RXMesh, we store for each patch the connectivity from face to edges ($\mathcal{FE}$) and from edges to vertices ($\mathcal{EV}$). Storing this incidence information allows us to represent and operate on meshes without restrictions on mesh connectivity, meaning we can handle non-manifold meshes. Since the patch is small, we can use 16-bit indices to index and enumerate all mesh elements, which saves memory allocation and, more importantly, reduces the amount of needed shared memory per patch. For static query operations, our implementation is similar to RXMesh (see Section 4.3 for more details). Unlike RXMesh, we store a bitmask that indicates if a mesh element is active or not. Thus, deleting elements amounts to changing their bit in the active bitmask. Adding new elements requires knowing their top-down connectivity, i.e., incident edges for faces, and incident vertices for edges. Adding new vertices requires only incrementing the number of vertices since we do not store any per-vertex connectivity information.

Mesh Geometry: In RXMesh, we stored mesh attributes as a single array in global memory. These attributes could be associated with vertices, edges, or faces. To access this array, we mapped per-patch *local* indices to *global* ones to index the attribute array. In dynamic settings, resizing a patch would necessitate updating this global index space, leading to costly synchronization across all patches. Instead, we make a different choice. We localize mesh attributes

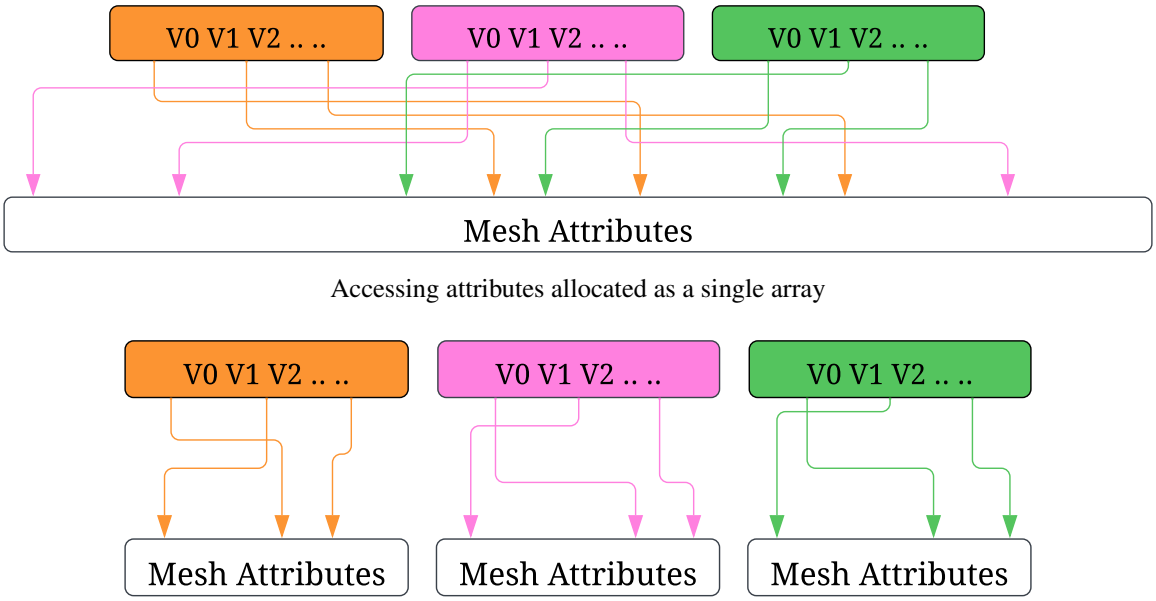


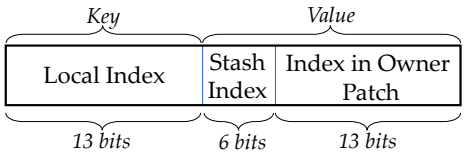
Figure 5.9: Allocating attributes as a single array requires mapping local indices to their global ones (top). Localized attribute allocation eliminates the need to map indices and leads to better caching and overall higher performance (bottom). Here we assume attributes associated with vertices. However, in our system, attributes could be associated with vertices, edges, or faces

by allocating them on a per-patch basis. With per-patch allocation, we eliminate the need for the local-to-global mapping (Figure 5.9). We essentially rely on the GPU's L1/L2 to cache accesses to the mesh attributes. Topology queries and updates are instead cached in shared memory since mesh queries and updates require extra temporary buffers that can be allocated in shared memory.

Ribbons: Ribbon elements require special treatment for dynamic changes since they duplicate mesh elements that reside in neighboring patches. For example, during a query, we do not return the ribbon elements themselves but instead the owner patch of the ribbon elements so that the user can subsequently access their attributes. To store the ribbon elements, we first classify the mesh element as either *owned* by the patch or a *ribbon* element (not owned by the patch). For any mesh element, we need to (1) check if it is classified as a ribbon element and (2) store the corresponding owner patch and its local index within the owner patch. During dynamic updates, an owned mesh element may become a ribbon and vice versa. So, the storage for

ribbon elements changes over time.

Ideally, we would like to allocate just enough memory for the ribbon elements to store their information without extra storage. Since RXMesh is a static data structure, we categorized mesh elements as owned and ribbon elements, and that status never changes during computation. Consequently, we assigned consecutive indices to ribbon elements and their storage was contiguous and compact. But in a dynamic scenario, an internal element may become a ribbon element after topology changes.



The key-value of our hash table

Given the above requirements on storing ribbon elements, we store them in a simple GPU-based dynamic cuckoo hash table [8]. Hash tables are an excellent choice to meet our requirements because (1) they allow compact data storage (i.e., their load factor can be as high as 0.9) and (2) they have constant-time insertion and deletion. First, every patch stores all its neighbor patches in a small array called a *patch stash*. Since a patch is surrounded by very few patches, the size of the patch stash is restricted to 64 (2^6) patches, and so we only need 6 bits to store this index. While we do not have a theoretical guarantee of the upper bound of the neighbor patch count, we have not found any realistic scenario (based on our experiments) where a patch is surrounded by more than 64 patches. We then use the mesh element index (represented using 13 bits) within the patch as a key in the hash table. The value stored per key is another 19-bit concatenation of the index in the patch stash (6 bit) and the local index in the owner patch (13 bit) (see the inset). The hash table allows us to add/remove mesh elements to/from the ribbon in constant time without excessive memory allocation, which is critical for performance given the limited shared memory resources and its impact on GPU occupancy. Finally, we also use a bitmask to check if an element is a ribbon or owned to save accesses into the hashtable when the mesh element is owned, i.e., optimizing for the common case.

Memory Allocation Since patches dynamically grow and shrink in size, we pre-allocate all necessary memory in advance to avoid costly memory allocation during application execution. Two factors control this pre-allocation: (1) the number of patches that will be added as a function of the initial number of patches, and (2) the maximum size a patch can grow to as a function

of its initial size. Both are user-defined runtime parameters to maximize flexibility depending on the application's behavior. For example, in static applications, these parameters are set to 1 as patches do not change.

If a patch grows beyond its predefined maximum size, we slice the patch into two patches. This ensures that the patch will always fit into shared memory. To slice a patch p into two patches, q and r, we create only one new patch q, while p remains the same with some mesh elements deleted. We achieve this by running 10 iterations of Lloyd's k-means clustering algorithm on p. Next, we copy the connectivity from p to q for the elements that belong to q and delete the unnecessary elements from p. The ribbon information of p is also copied to q, including additional ribbon elements at the interface between p and q. All these memory operations occur in shared memory before being written back to global memory, resulting in minimal overhead compared to the application's runtime. Finally, when a patch is sliced, we transfer all attributes for vertices, edges, and faces now owned by q from p to q.

5.3.2 Cavity Operations

As mentioned, our system first automatically assigns CUDA blocks to patches and then performs the cavity operations. From the user's perspective, cavity operations can be divided into three stages: (1) register a new cavity, (2) process the cavity, and (3) fill in the cavity. Internally, the first stage collects all the cavities that the user created on the given patch. The second stage ensures that there is a (sub)set of conflict-free cavities available for the next stage. The third stage finalizes the operation by (optionally) filling in the cavities before writing everything to global memory. Conflicting cavities will be attempted in subsequent iterations (Section 5.2.3).

Cavity Registration: Our system provides predefined *templates* that cover a wide range of cavity configurations (Figure 5.4). A template consists of a *seed* element and a local neighborhood that will be deleted. It is possible to add a user-defined template by specifying these two requirements. Note that deleting a mesh element leads to deleting all upward elements incident to it, e.g., deleting an edge leads to deleting its incident faces. To register a new cavity, the user calls a cavity template on a specific mesh element. Our system atomically increments the number of cavities associated with the patch and stores the seed's cavity ID in the shared memory.

Cavity Processing: We first detect intra-patch conflicting cavities before attempting to resolve inter-patch conflicts. We detect conflicting cavities within the patch by propagating the cavity ID from the seed to its adjacent/incident elements as described by the cavity template. Propagating information from an n-dimensional element to an m-dimensional element is a gather operation if $n < m$, and an atomic operation if $n > m$. For example, a face checks the cavity ID of its three edges if the edge is the seed (i.e., propagating information from edges to face) while a seed edge atomically sets the cavity ID of its two vertices (i.e., propagating information from edges to vertices). In both cases, we can detect if two cavities write to the same element. Then, we construct a graph where cavities represent the vertices of this graph and two vertices are connected with an edge if their corresponding cavities overlap. Now, in order to maximize the number of non-conflicting cavities that we can process in parallel, we compute Blelloch et al's greedy maximal independent set algorithm [11] on this graph in parallel. Then, we use an atomically updated bitmask in shared memory to indicate if the cavity is deactivated. This helps inform the user which cavity is successful. This approach guarantees that we process as many cavities concurrently as possible.

As we mentioned in Section 5.2.1, cavities that cross the patch boundaries must ensure that their changes are not visible to other patches. We do this by expanding one patch so that such cavities are fully contained in the patch. To do this, we first attempt to lock the neighbor patches that may be impacted by these cavities. If we cannot acquire the lock on these neighbor patches, we discard this patch and schedule it to be processed later. If we successfully acquire the lock, then we can read from the neighbor patch and change the ownership of some of its elements. If a mesh element (i.e., vertex, edge, or face) change its ownership from one patch to another, we also move all its attributes to the new owner patch. The whole block is engaged in this process: all threads within the block collaborate to expand the patch, which maximizes the memory throughput and reduces thread divergence.

A patch may have too many cavities along its boundaries and expanding the patch may require more memory than what we can store in shared memory. In such cases, we slice the patch into two patches. We schedule all patches that need to be sliced at the end of every update iteration.

Cavity Fill-in: For each active cavity, our system provides an iterator over the cavity boundary edges and vertices. Using this iterator, the user can add new mesh elements by connecting them to the cavity boundary edges or vertices. Internally, we create these new elements by appending to $\mathcal{FE}$ and $\mathcal{EV}$ connectivity information. Thus, during cavity fill-in, the user can query the cavity's old/deleted topology as well as access their deleted-element attributes. After committing the patch to global memory, the deleted topology can be re-written in subsequent update operations.

5.3.3 Queue-based Scheduler

Our system design requires a scheduler that can dynamically assign blocks to patches such that each patch is assigned at least once. If a patch fails to be processed (e.g., is not able to acquire the lock of a neighbor patch), the scheduler must schedule that block at a later time. The scheduler should also issue locking requests and maintain information about if a patch is locked.

We use a simple parallel array-based queue [40] to coordinate assigning patches to blocks. Every block declares a leader thread that tries to dequeue a patch from the queue. If the leader thread is successful in reading a patch, it communicates the assigned patch to the other threads via shared memory. Queue-based processing allows failed patches to be enqueued for future processing. Additionally, it improves load balance since blocks that complete their work can dequeue another patch to process. This simple design also maximizes GPU utilization since both control (the scheduler) and processing (patch computation) is local to the GPU.

Locking Algorithm: Locking patches in our system to allow mutual exclusion to read/update neighbor patches must satisfy two challenging requirements. First, we need to make sure that attempting to lock a patch does not lead to a deadlock. For example, using spin locks [41], patch p_0 may spin waiting to lock p_1 while p_1 is also spinning waiting to lock p_0. Second, we must guarantee forward progress in the case of contention, i.e., in a scenario where multiple blocks try to lock the same patch at the same time, at least one block must be able to lock the patch and make progress. In our implementation we allow the locking algorithm to trade off fairness in favor of quick response time since it might be beneficial to rollback updates (which is cheap) rather than waiting to acquire the lock of a neighbor patch, which would leave the SM idle for

```
class Lock {
    int* lock;   // initialized to FREE
    int* spin;   // initialized to MAX_INT

    __device__ bool acquire_lock(int threadID){
        int attempt = 0;
        while (atomicCAS(lock, FREE, LOCKED) == LOCKED) {
            if (attempt == MAX_ATTEMPT) {
                int other = atomicMin(spin, threadID);
                if (other < threadID) {
                    return false;
                }
                attempt = 0;
            }
            attempt++;
        }
        atomicExch(spin, threadID);
        return true;
    }

    __device__ void release_lock(){
        atomicExch(spin, MAX_INT);
        atomicExch(lock, FREE);
    }
};
```

Listing 5.1: Our device-side spin lock algorithm. Both kernels are called by a single thread in the block which then broadcasts the results to all threads in the block. This algorithm guarantees deadlock freedom since, after multiple attempts, threads back off if other threads try to acquire the same lock.

an extended period. Finally, we can and do optimize for a scenario with only modest contention since any patch is a neighbor of at most 64 other patches.

While many mature libraries implement locking mechanisms for multithreaded CPU applications (e.g., Intel Threading Building Blocks [81] or Boost [78]), there are no similar standards on the GPU and CUDA does not offer out-of-the-box locking mechanisms that can be used inside the kernel. This has motivated us to implement a locking mechanism customized to our workload. Our locking algorithm (Listing 5.1) is based on spin lock where a thread keeps attempting to acquire the lock (using `atomicCAS`) until it is successful. If not, the threads keep spinning (in a `while` loop) until the patch is available. To allow forward progress in case of

contention, the spinning threads should be aware if other threads are trying to lock the patch. If more than one thread tries to lock the same patch, only one of them should keep trying while others should quit to prevent deadlock. We do this by using `atomicMin` on a variable allocated for each patch that we call `spinner` initialized with a very large value. After a certain number of attempts to lock the patch (using `atomicCAS`), the thread `atomicMin` the `spinner` with the thread ID.

Only when the returned value is less than the thread ID does the thread exit, indicating a failure to lock the patch. This is because some other thread (with a lower ID) is also attempting to lock the patch at the same time, in which case the lower ID thread wins. When the winner thread releases the locks, it additionally resets the `spinner` to the initialized value. The `spinner` allows cheap communication between threads that are attempting to lock the same patch that is currently locked by another thread. To reduce contention on atomic operations, threads within a block elect a single thread to attempt locking the desired patch. Upon return, the result is broadcast to all threads in the block.

5.4 Applications

Here, we demonstrate the effectiveness of our design decisions on a set of common geometry processing applications. We first ensure that the changes in our data structure improve static application performance by comparing against RXMesh. Then, we assess the efficacy of our system through three dynamic applications, each targeting a specific aspect of our system's capabilities. The first application, Delaunay edge flip, maintains a constant mesh size but requires potential expansion of patches for conflict handling. This application serves as a benchmark to evaluate our system's ability to maximize parallelism and to determine the impact of altering patch layout on the overall performance. The second application, isotropic remeshing, challenges our system with fluctuating workloads, which include both increasing and decreasing mesh sizes, alongside other dynamic operations that maintain mesh consistency and static operations. Surface tracking, our third application, involves a spectrum of dynamic operators over an extended duration, challenging our system to sustain optimal performance over prolonged periods. Collectively, these applications offer a comprehensive insight into our system's

performance across a range of scenarios commonly encountered in dynamic mesh processing applications.

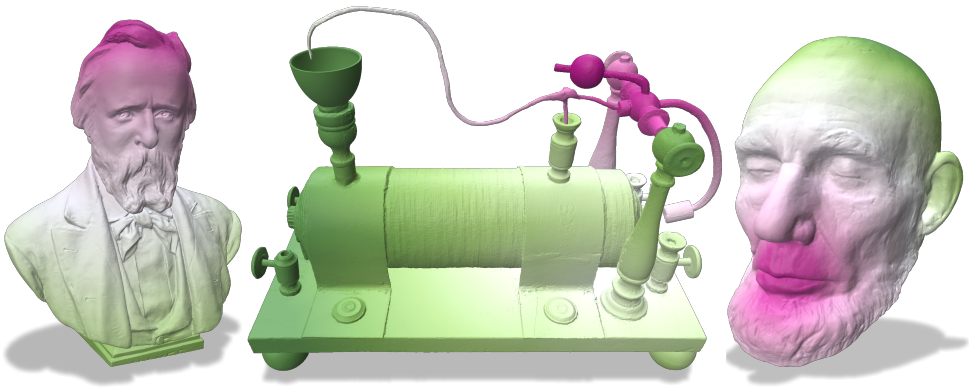
For dynamic applications, which lack a GPU comparison, we compare against WMTK [46], as it is the only modern system for dynamic triangle mesh processing that leverages the parallelism of multi-core CPU systems. In all experiments, input meshes are collected from the Smithsonian [90], ThreeDScans [57], and Thingi10K [101] repositories since they feature meshes with millions of faces. We conduct all experiments on an A100 GPU with 80 GB of memory using CUDA 12.2 on a machine featuring an AMD EPYC 9124 16-core 3.7 GHz Processor with 270 GB main memory. For all applications, we report only the application runtime—excluding any preprocessing time, e.g., our system's patch construction and WMTK's data structure initialization.

Parameters As mentioned in Section 5.2, we avoid inter-patch conflicts by ensuring that cavities do not imprint on other patches. If a cavity contains mesh elements that reside on other patches, we expand the patch and delete these elements from the neighboring patches. Thus, after transferring the patch from global memory to shared memory, the patch may increase in size. To prevent patches from becoming too large too frequently, we set the patch size to 256 faces during patch construction (Section 4.3.3), which is smaller than RXMesh's default patch size of 512 faces. This allows us to operate on a patch multiple times, potentially expanding it before it needs to be sliced. Since patch size is a runtime parameter, for purely static applications (e.g., computing geodesic distance), we revert to a 512 faces patch size as it provides optimal performance. Additionally, a patch is allowed to grow up to two times its initial size before slicing it.

In all applications, we use a CUDA block size of 512 threads. The number of blocks launched in a single kernel is automatically calculated based on the remaining number of patches in the queue. Similarly, we calculate the amount of shared memory needed to store patch information, i.e., connectivity information ($\mathcal{FE}$ and $\mathcal{EV}$) and the ribbon elements hashtable.

5.4.1 Comparison against RXMesh

We begin with a static geodesic distance computation (Figure 5.10) to compare our system performance against RXMesh. Here, we follow the same algorithm we discussed in Section 4.5.2



F = 7.7M # F = 16.2M # F = 6.1M

Figure 5.10: Examples of geodesic distance computation of large models

Table 5.1: Time and speedup of our system against RXMesh running geodesic distance computation [83].

# F ($\times 10^6$)	RXMesh (s)	Our (s)	Speedup
5.2	1.1	0.85	**1.32**
6.1	1.0	0.86	**1.17**
7.7	2.4	1.4	**1.65**
16.2	5.5	2.9	**1.90**
19.9	6.4	3.6	**1.75**

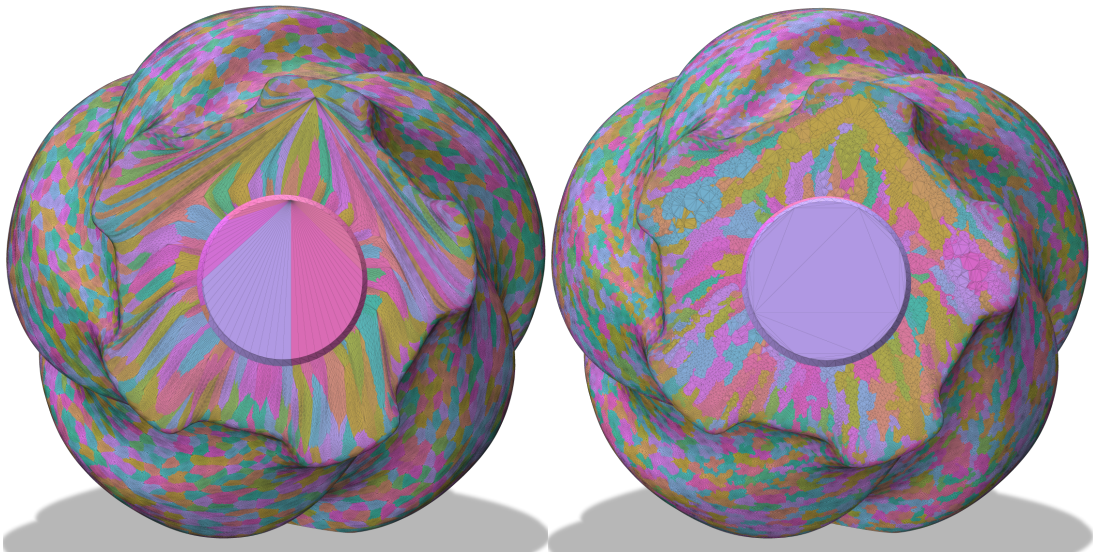
based on the minimalistic parallel algorithm for geodesic distance computation [83]. Our system outperforms RXMesh in this static application with an average geometric mean speedup of 1.53× (Table 5.1). The major difference in the query pipelines between our system and RXMesh is how ribbon element information is accessed. While RXMesh maps all elements to global indices, we use a hash table (within the shared memory) to map ribbon elements to their corresponding elements in the owner patch. Compared to RXMesh, we require more computation but obtain better locality, and this trade-off results in an overall performance speedup. In addition, our system localizes geometric information better since allocating geometric data is done per patch. Thus, in our system, accessing geometric data will result in better memory

coalescing.

Moreover, our data structure for *static* applications requires half the memory vs. RXMesh using the same patch size. We use a similar simplified manifold mesh for calculating memory footprint as we did in Section 4.4.1. For such a model, RXMesh requires 34 bytes/face. Using the Euler-Poincaré characteristic, our data structure stores for each patch the connectivity from face to edges ($\mathcal{FE}$) and from edges to vertices ($\mathcal{EV}$) which requires $3F_p$, where F_p is the average number of owned faces in a patch. We also store a bitmask for vertices, edges, and faces that indicate if the mesh element is owned and if it is active (which requires $0.75F_p$). Finally, we store the owner patch and local index within the owner patch for each not-owned mesh element in a hashtable as a 32-bit unsigned integer. This requires $\frac{12RF_p}{L}$, where R is the ratio of the ribbon elements, and L is the load factor of the hashtable. Thus, the total memory requirement in our data structure is $12.75 + \frac{12R}{L}$ bytes/face. Using the same patch size as in RXMesh, the ribbon ratio is $R \approx 0.4$. The load factor in our hashtable is 0.8. Thus, our data structure requires 18.75 bytes/face, which is $1.81\times$ less memory than RXMesh. However, when our system addresses *dynamic* applications, we must allocate (ahead of time) more memory to permit us to add more elements to patches and more patches to the mesh to prevent allocating memory while processing. Thus, for dynamic applications, our data structure may require more memory than the (static) RXMesh.

5.4.2 Delaunay Edge Flip

A Delaunay mesh is one where the sum of two opposite angles of an edge is less than $180°$. Flipping the edges of an input mesh to meet the Delaunay criterion is an easy way to improve the mesh quality, i.e., improving the minimum and maximum angles. To achieve the Delaunay criterion, we iteratively attempt to flip an edge if it is not a Delaunay edge until all non-Delaunay edges have been flipped. This algorithm is guaranteed to terminate for surface meshes [22]. Here, to create cavities, the user simply checks the two opposite angles of an edge. If their sum is greater than $180°$, then the edge and its two incident faces form a cavity. To fill in a cavity, the user connects the two opposite vertices and creates two new faces. Figure 5.11 shows the result of flipping non-Delaunay edges and how it improves the mesh quality. Table 5.2 compares our system implementation against WMTK's best configuration (32 threads, see Section 5.2.1). In



Non-delaunay input mesh Delaunay output mesh

Figure 5.11: Example of input/output of the Delaunay edge flip application on a model with $\sim$1M faces where color indicates the face patch ID.

Table 5.2: Time and speedup of our system on the Delaunay edge flip application against Wild Mesh Toolkit (WMTK) [46] using 32 CPU threads.

# F ($\times 10^6$)	WMTK (s)	Our (s)	Speedup
0.006	0.217	0.582	**0.373**
0.064	2.088	1.523	**1.372**
2.53	70.483	0.959	**73.513**
3.154	168.744	1.959	**86.129**
5.486	233.323	4.678	**49.876**
9.474	556.927	13.964	**39.882**
11.318	490.455	7.22	**67.932**
26.063	1444.971	31.241	**46.253**

Table 5.3: Assessing the impact of altering the patch layout by measuring time (in seconds) per iteration for the conjugate gradient-based Mean Curvature Flow computation before and after Delaunay edge flip. Slowdown is measured as the ratio between the time after Delaunay edge flip to the time before, i.e., lower is better.

# F ($\times 10^6$)	Time Before (s)	Time After (s)	Slowdown
0.006	0.134	0.153	1.142
0.064	0.18	0.187	1.039
2.53	3.435	3.547	1.033
3.154	4.097	6.036	1.473
5.486	6.698	6.836	1.021
9.474	11.747	15.931	1.356
11.985	14.301	14.515	1.015
26.063	38.274	40.065	1.047

both systems, we terminate when no further edges can be flipped to meet the Delaunay criterion. Our system performance is lower for smaller meshes due to limited parallelism in these models, which may be better processed serially. On the larger meshes that we target, our system achieves an order of magnitude speedup thanks to maximizing parallelism. Our performance advantage is due to increased parallelism (we can flip more edges concurrently) and due to the better memory locality in our data structure.

We use the Delaunay edge flip application to assess the impact of changing the patch layout. Moving mesh elements from one patch to another to avoid inter-patch conflicts and slicing patches (see Section 5.2.1) might diminish the quality of the patches, leading to imbalanced patches. Here, we measure this impact by timing a static application before and after the dynamic changes (Delaunay edge flip). Our static application here is mesh smoothing (see Section 4.5.1 for more details). We first measure the time of smoothing the input mesh. Then, we run the Delaunay edge flip application on the same input mesh and the same input vertex position (not the smoothing vertex position). After turning the mesh into a Delaunay one, we use the same input vertex position to re-run the smoothing again. Table 5.3 shows the time

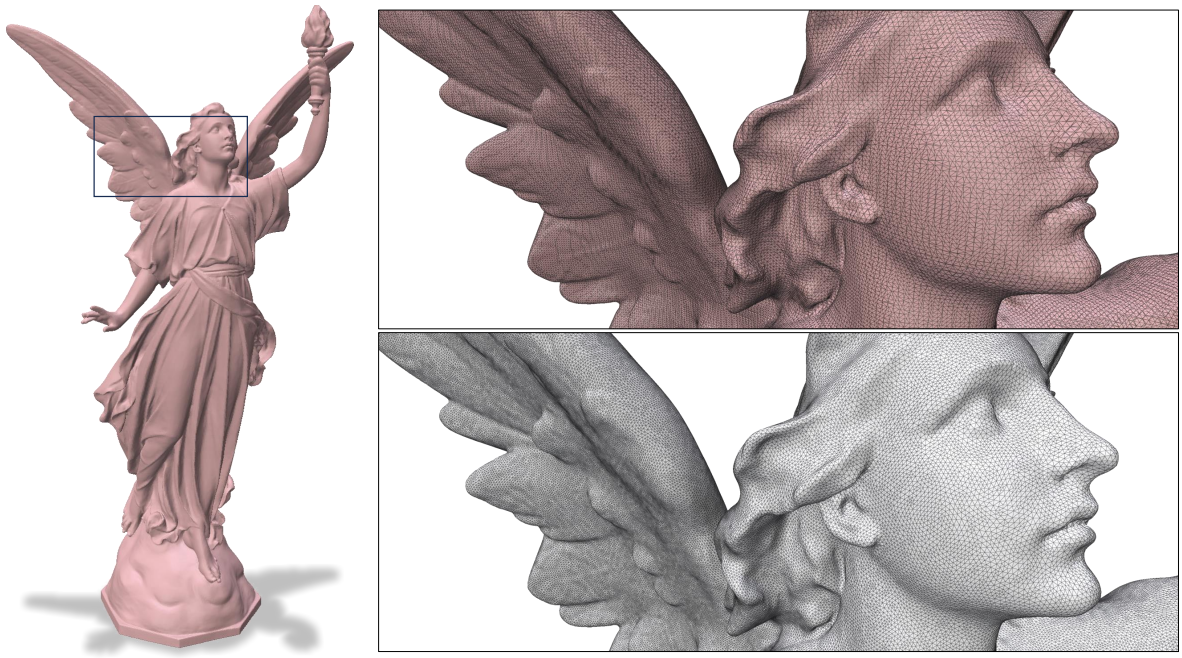


Figure 5.12: Example of input (top) and output (bottom) of our implementation of the uniform isotropic remeshing after three iterations.

per iteration for the non-Delaunay mesh (before dynamic changes) and the Delaunay mesh (after dynamic changes) along with the slowdown. The slowdown is calculated as $\frac{T_a}{T_b}$, where T_a is time after and T_b is time before the Delaunay flip. Here we report the time per iteration since the number of CG iterations changes due to changes in the mesh connectivity. However, the computation cost is the same per iteration since the mesh size (i.e., number of vertices) is the same for both the non-Delaunay and Delaunay mesh. While the patches undergo several changes in this application, this has a small impact on performance for the static application (i.e., local query operations) as the geometric mean of the slowdown is 1.101 with a maximum slowdown of 1.473. This suggests that while the patch quality might degrade due to dynamic changes, this has a very small impact on the runtime of the static operations. This slowdown is because patches might become imbalanced or degenerate after several dynamic modifications. In cases where the static mesh processing performance after dynamic changes is more important, it may be effective to re-patch the output of the dynamic application. We leave this for future exploration.

5.4.3 Isotropic Remeshing

Isotropic remeshing [13] improves the quality of an input mesh by making the output triangles as equilateral as possible (Figure 5.12). The algorithm consists of four phases; each does a full pass over the mesh before starting the next phase. The four phases are (1) split long edges, (2) collapse short edges, (3) equalize vertex valence via edge flip, and (4) vertex smoothing. The first two phases are guided by user input, specifically target edge length, which here we set to the average edge length of the input mesh. Unlike the Delaunay edge flip application, remeshing can either increase or decrease the mesh size while alternating between different phases. With these different dynamic operations, remeshing becomes challenging to run in parallel due to the many conflicting updates it generates. Implementing such an application in our system shows our system's flexibility in handling such a workload. In our tests, we run three iterations of the remeshing algorithm and the results of both our implementation and WMTK match in terms of the output mesh size and quality. Table 5.4 compares our implementation against the WMTK implementation; our system achieves a 1.85–8$\times$ speedup. Our system yields denser results. This is primarily attributed to the tendency for many edge split operations (which increases mesh size) to encounter conflicts in WMTK. This arises from WMTK's wider neighborhood locking approach around each edge-based operation, i.e., the locking of the one-ring of the edge's two end vertices. In contrast, our system allows edge splits to proceed concurrently as long as the edge's incident faces are in separate cavities. The slight variation observed in the final output mesh can be attributed to differences in operation scheduling between our system and WMTK.

5.4.4 Surface Tracking

Dynamic surface motion is prevalent in various computer graphics applications, e.g., motion through geometric flow or physical simulation where the surface serves as the interface in multi-phase flow simulation. Implicit methods for surface tracking may encounter numerical dissipation and often require very high resolution to capture fine surface details. Brochu and Bridson [16] introduced a robust *explicit* surface tracking method that involves collision detection to prevent mesh intersections. In essence, following each step of surface advection, the method enhances mesh quality before checking for collisions. Mesh improvement entails a combination

Table 5.4: Mesh statistics and timing for the uniform remeshing application for WMTK and our system. For each input, our system's output and WMTK's output, we report the number of faces (# F); minimum and maximum edge length as a function of the average edge length (L); minimum, maximum, and average vertex valence (V); timing in seconds for our system and WMTK (T); and our system speedup.

Input							WMTK							Ours							Speedup
# F ($\times 10^6$)	L		V			# F ($\times 10^6$)	L		V			T	# F ($\times 10^6$)	L		V			T	
	min	max	min	max	avg		min	max	min	max	avg			min	max	min	max	avg		
0.91	0.0033	7.6	3	14	6	0.80	0.274	2.024	3	8	6	59.2	1.07	0.268	2.9	3	9	6	17.4	**3.4**
1.4	0.00031	5.346	3	24	6	1.2	0.099	1.89	3	9	6	86.3	1.4	0.135	2.57	3	9	6	36.8	**2.34**
2.3	0.641	1.57	3	8	6	1.9	0.332	1.85	3	9	6	155.2	3.04	0.21	2.69	3	9	6	50.8	**3.05**
2.4	0.767	38.4	4	7	6	1.9	0.0399	9.12	3	10	6	117.7	2.5	0.0324	13.6	3	8	6	48.8	**2.41**
2.5	0.00037	1.8	3	12	6	1.87	0.158	1.862	3	9	6	138.8	3.9	0.2	2.1	3	8	6	74.7	**1.85**
11.9	0.311	1.341	3	9	6	9.6	0.312	1.784	3	9	6	759.14	11.3	0.301	2.65	3	9	6	94.4	**8.04**

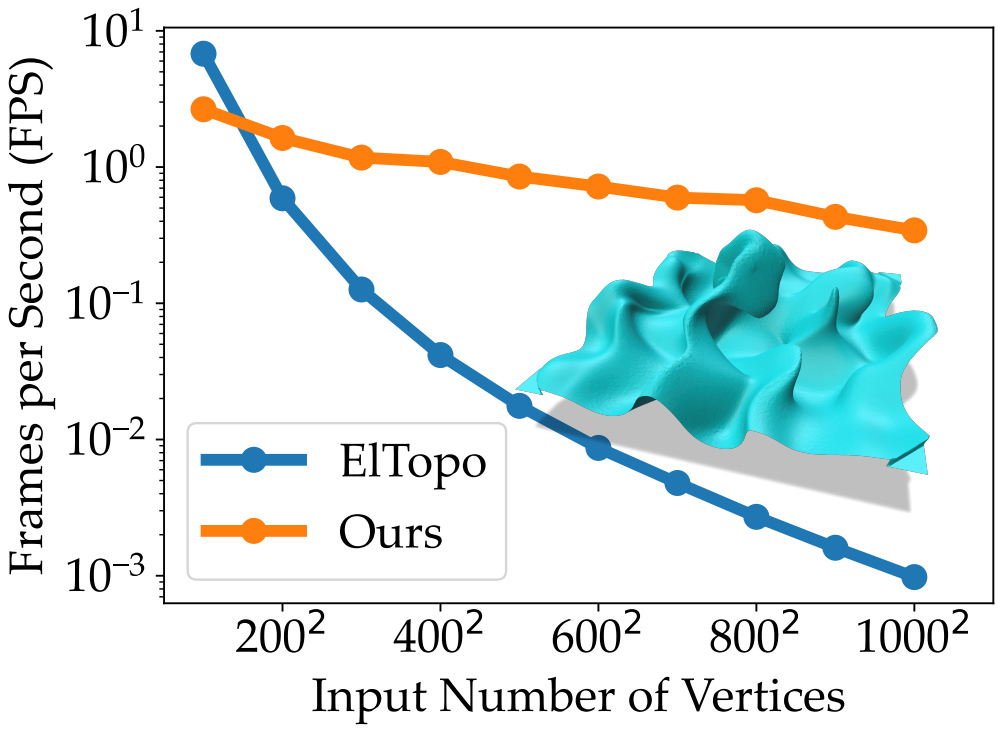


Figure 5.13: Scalability test of our system vs. El Topo [17] on surface tracking of curl noise advection of a 2D grid. Every frame is a single advection step followed by mesh improvement step.

of edge split, edge collapse, edge flip, and null-space vertex smoothing, each utilizing different criteria to maintain high-quality mesh during progressive surface evolution. For further details, readers are directed to the works of Brochu and Bridson [16] and the open-source El Topo software [17].

We implemented explicit surface tracking with mesh improvement but without collision detection. Our implementation faithfully mirrors the behavior of the El Topo software when collision detection and topology changes are disabled. We employed an example of surface advection using a curl noise velocity field on a 2D square grid embedded in 3D space. The grid's dimension is defined by the number of vertices along one direction. We run 100 iterations of surface tracking to illustrate how our system could sustain good performance over a large number of update iterations. Figure 5.13 shows a scalability comparison between our GPU implementation and El Topo's single-thread implementation. Notably, unlike WMTK, El Topo

does not provide a multithreaded implementation. While El Topo might exhibit faster performance for small meshes (≈20K faces), our GPU implementation surpasses it by more than two orders of magnitude for larger meshes (≈2M faces). In this particular example, over 99% of the runtime is attributed to mesh improvement, underscoring the advantages of GPU acceleration for dynamic changes. Such substantial speedup potentially paves the way for broader adoption of explicit surface tracking to mitigate the drawbacks of implicit methods.

5.4.5 Performance Discussion

Here we address two key aspects concerning the performance of our system. We begin by examining the common bottleneck we identified across all applications, followed by an exploration of failure cases or instances where our system falls short of optimal performance.

Bottleneck. Our system's design aims to confine all computation and memory accesses within the GPU's shared memory. We achieve this by initially reading patch information from global memory into shared memory, processing the patch within shared memory, and then writing it back to global memory. These accesses to global memory are coalesced and effectively utilize global memory bandwidth. However, other operations require accessing non-contiguous memory, e.g., reading neighbor patch topology information. These reads inevitably require global memory access. Due to the limited size of shared memory, we cannot read an entire neighbor patch into shared memory. Instead, threads load the necessary neighbor-patch data directly and in a non-coalesced manner since we only need to read the connectivity or individual elements (faces/edges). However, only when we must resolve inter-patch conflicts do we have the need to read neighbor patch information.

We evaluated the impact of resolving inter-patch conflicts which include locking and reading these neighbor patches. We measure this impact by deactivating any cavity triggering the reading of neighbor patch information in the Delaunay edge flip application on the mesh depicted in Figure 5.11. Our experiments showed that the cost of resolving neighbor patches accounted for 3/4 of the overall runtime. Although this latter scenario does not produce a Delaunay mesh, it sheds light on the location of an inherent bottleneck, i.e., uncoalesced global memory read due to non-local element access.

Limited Parallelism. Our system aims to maximize parallelism in dynamic mesh processing and assumes an abundance of concurrent operations in such applications. However, if this assumption does not hold true, our system may deliver subpar performance, resulting in serial processing of dynamic operations. Small meshes, for instance, simply do not exhibit enough parallelism to fill the GPU. Our results (see Table 5.2 and 5.4, as well as Figure 5.13) illustrate instances where systems with limited or no parallelism (e.g., WMTK and El Topo) outperform ours on small meshes. The applications we studied generally require operations spread across the entire mesh. If other applications are characterized by dynamic operations concentrated in a small region of the mesh, our system may have limited parallelism to leverage.

5.5 Summary

In this chapter, we present the first system for dynamic mesh processing entirely on the GPU, broadening the number of applications that can now run a dynamic workload on the GPU (e.g., graph data structures [7] and hash tables [6]). While our system focuses on dynamic mesh processing, it also improves the performance of the static case presented in Chapter 4. In our implementation and applications, we constrained ourselves to follow the description of the algorithms, which is often a serial description. For a few applications, this constraint does not impose any restriction on exploiting parallelism, e.g., Delaunay edge flips. However, since many algorithms depend on priority-based processing (e.g., mesh simplification [38], spherical parameterization [42], or Delaunay refinement [89]), this limits the amount of parallelism the applications expose, which subsequently limits the amount of parallelism that our system can exploit. Our system facilitates exploring relaxing the priority in geometry processing applications in favor of speedup on massively parallel hardware like the GPU.

Chapter 6

Conclusion

In this dissertation, we presented the argument that designing new data structures and programming models specifically optimized for GPU hardware can lead to significant speedup in processing geometric data. We demonstrated that attaining high performance necessitates leveraging mesh locality. Traditionally, relying on hardware caching has been the standard way to exploit locality. Our work shows that programmer-managed caching, which is concealed from the end user beneath the programming model, achieves superior performance. Exploiting locality in generic mesh processing poses a considerable challenge due to the dynamic nature of user queries and updates on various mesh elements (i.e., vertices, edges, faces). We proposed that employing patching strategies is the most effective method to leverage mesh locality and maintain consistent high performance across diverse operations and input mesh qualities. Our patch representation allows for several optimizations that are otherwise infeasible. We showed that in order to achieve high performance threads have to collaborate (either for static query operations or dynamic updates). We designed programming models that abstract this complexity and offer a straightforward conceptual framework for understanding highly parallel geometric data processing operations. The flexibility of these programming models allowed us to implement different applications while scheduling query/update operations with high cache efficiency.

Design Triumphs: The success of our systems hinges on a few key design decisions. The most important one is leveraging shared memory extensively and using it to create additional data (that is traditionally created and stored in global memory) that could be discarded once the

user is done with it. This allowed us to store compact data structures in global memory without restriction on the type of query operations we could perform. This design choice was later adapted in MeshTaichi [97]—a compiler for unstructured mesh (both triangle and tetrahedral mesh) operations based on the celebrated Taichi [43] language. More importantly, confining operations within shared memory allowed near zero-cost rollback for speculative processing for dynamic update operations. Beyond mesh processing, irregular dynamic problems that exhibit the same characteristics as mesh processing (i.e., an irregular computation that is amenable to localize its operations) could also benefit from this concept, e.g., dynamic graph analytics algorithms on high-diameter graphs (such as road networks and circuits) and other loosely connected networks appearing in applications like network security and topological data analysis.

Design Setbacks: Our goal was to develop a system for geometric data processing that provides a low entry bar for geometry processing practitioners to leverage the GPU for their applications. In retrospect, our choice of C++/CUDA as the foundational development framework was guided by its proven performance and widespread adoption in high-performance computing. While this decision leveraged the strengths of these languages, it also introduced challenges in usability that constrained its accessibility. The current trend in using simpler, higher-level languages (e.g., Python) in complex computational tasks (e.g., machine learning) sets the user expectations for ease of use, accessibility, and rapid prototyping capabilities, pushing the boundaries of what they seek in new tools and frameworks for their computational needs.

A more strategic approach would have been to develop a domain-specific language (DSL), akin to notable examples such as Simit [52], Ebb [10], Taichi [43], and Halide [82]. These DSLs have demonstrated their efficacy in abstracting complex computational patterns and hardware intricacies offering a more intuitive and streamlined development experience for the user. The use of modern compiler infrastructures (e.g., LLVM) significantly lowers the barrier to creating efficient, domain-specific compilers that can offer tailored optimizations to the needs of specific applications. Additionally, just-in-time (JIT) compilation could enhance flexibility, especially for dynamic applications requiring on-the-fly optimization based on runtime data. Combined with a DSL, it could improve development by enabling high-level abstractions and optimizations without losing performance.

6.1 Future Directions

My vision is to design fast, robust, and scalable systems for processing geometric data to empower geometry processing researchers with fundamental tools that efficiently harness modern hardware accelerators, streamlining their work by removing the complexities of low-level implementation. These systems are pivotal in enabling researchers to easily test their ideas against extensive geometric data sets, which are increasingly common in today's data-driven world. Testing on such massive scales ensures that the algorithms and applications developed are more robust and highly applicable to real-world scenarios. Beyond compilers and DSLs, there are still some fundamental geometric data processing capabilities that are challenging to perform on the GPU and require further research to realize them—potentially extending our current system.

Single and Multi-GPU Dynamic Tetrahedral Mesh Processing: To broaden the range of applications that we can efficiently run on the GPUs, our system could be extended to tetrahedral meshes which are fundamental in numerous domains, including engineering simulations, medical imaging, and computer graphics, due to their ability to accurately represent complex 3D shapes and volumes. Despite their importance, existing solutions for tetrahedral mesh processing (e.g., MeshTaichi [97]) are limited to static mesh processing on a single GPU and do not support dynamic topology changes. However, supporting tetrahedral meshes may stress out the GPU's shared memory—potentially reducing occupancy and overall performance. Nevertheless, the projected direction of GPU technology indicates an increase in shared memory per block, as demonstrated by the NVIDIA Hopper architecture, which offers up to 227 KiB shared memory per block, representing a 1.6× increase compared to the Ampere architecture. Another challenge is the fact that tetrahedral meshes often require a large amount of memory that may exceed the capacity of a single GPU, especially for large complex datasets. Our data structure design is conducive to multi-GPU distribution by partitioning the mesh into patches. However, our design was optimized for a single GPU, resulting in small patches that can fit into the GPU's shared memory. Such a design might not be ideal for multi-GPU environments, as it leads to numerous message exchanges for communication between multiple GPUs. Therefore, further research is required to develop a design that optimizes mesh partitioning for both single-GPU and multi-GPU processing scenarios. For maximum efficiency, this design will benefit from

overlapping part of the computation with the inter-GPU communication to hide the latency of the multi-GPU communication similar to what is done in structure grid [70].

Unstructured Mesh Neural Representation: Mesh compression is an alternative approach to overcome the memory requirements when it exceeds the capacity of a single GPU. This future direction would investigate the utilization of neural networks to minimize the memory requirements associated with unstructured meshes. Using neural networks for mesh compressions could also be beneficial for applications like mesh transmission for virtual and augmented reality headsets. Drawing inspiration from NeuralVDB [51], where a neural network is used to store leaf node information, they achieved a significant reduction in memory usage compared to VDB. In NeuralVDB, the tree hierarchy is represented using the conventional VDB data structure, while the values of the leaf nodes are encoded using a neural network. Similarly, we can use a similar strategy for unstructured mesh data, where the topology is stored using our data structure, and the geometry/attributes are represented through a neural network. The choice of our data structure for neural representation is advantageous due to 1) low memory footprint for topology information while efficiently supporting all topological mesh queries, and 2) partitioning the domain into patches where each can be encoded using a compact neural network. This parallels the VDB structure, where voxels are grouped into tiles/blocks. This concept can be extended to tetrahedral meshes, which typically demand more memory resources compared to surface meshes. One possible drawback of such a representation is the increased time complexity of neural network inference compared to direct access. However, it is expected to be faster than a full-mesh neural implicit representation [28] by efficiently pruning out queries that fall outside the scope of the patch.

Parallel Intrinsic Mesh Processing: Meshes in the wild are plagued with poorly shaped elements that make them ill-conditioned for numerical computations. *Intrinsic* mesh processing has recently pushed the limits of robust mesh processing making it possible to robustly process extremely poor-quality surface meshes for applications like geodesic distance, adaptive mesh refinement, parameterization [39], and simplification [60]. Intrinsic mesh processing is an example of multi-mesh where it decouples the triangulation used to describe the domain from the one used to implement the algorithm on that domain [86]. That way, we can circumvent the

pitfalls of altering the shape to improve its quality since the extrinsic triangulation is used to represent the domain faithfully while the intrinsic triangulation can change to meet other objectives, e.g., preconditions for finite-element simulation. While it can improve the robustness of such applications compared to conventional extrinsic mesh processing, no previous work has explored the potential of processing intrinsic triangulation on the GPU. Extending our data structure (or designing a new one) to support intrinsic mesh processing would be an impactful extension, as it will enable such a theoretically robust mesh processing framework on the GPU for the first time. However, working with two meshes that represent the same geometry adds another layer of complexity for parallel processing. Correspondence between the two (extrinsic and intrinsic) meshes needs to be maintained without sacrificing the parallel efficiency of working on each representation. The current patching approach we use may not be feasible since an extrinsic patch may not cover the same intrinsic area. Thus, building a new patching strategy that supports both representations would be one contribution of this work. Additionally, designing a programming model that allows the user to work on both representations simultaneously is another challenge that this direction should tackle.

Local and Global Mesh Updates on the GPU: Our data structure has been purposefully designed to facilitate efficient local static and dynamic mesh operations. This research direction explores the enhancement of our data structure by incorporating *global* information (e.g., using BVH) by building a tree hierarchy around bounding boxes encompassing the mesh patches. By implementing efficient mechanisms for both local and global updates, a diverse range of applications can be realized entirely on the GPU, e.g., contact simulation [35], surface tracking [16], and cloth simulation [75]. In these applications, local mesh updates should be propagated to the tree hierarchy to maintain its integrity. The primary focus of this work is to establish a tight integration between local and global mesh updates, achieving efficiency and speed without requiring a complete rebuild of both data structures. To handle tree updates, we can leverage existing techniques for fast BVH update/re-fit [54] and/or rebuild [48]. An important aspect of this research is the development of a programming model that abstracts away the complexities associated with managing global and local mesh updates.

REFERENCES

[1] Ahmed Abdelkader, Ahmed H. Mahmoud, Ahmad A. Rushdi, Scott A. Mitchell, John D. Owens, and Mohamed S. Ebeida. A constrained resampling strategy for mesh improvement. *Computer Graphics Forum*, 36(5):189–201, July 2017. doi: 10.1111/cgf.13256. URL http://escholarship.org/uc/item/5347s75h. Proceedings of the Symposium on Geometry Processing.

[2] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):154:1–154:9, December 2009. doi: 10.1145/1661412.1618500. URL https://escholarship.org/uc/item/445536d6.

[3] Oscar Antepara, Néstor Balcázar, and Assensi Oliva. Tetrahedral adaptive mesh refinement for two-phase flows using conservative level-set method. *International Journal for Numerical Methods in Fluids*, 93(2):481–503, February 2021. doi: 10.1002/fld.4893.

[4] Saman Ashkiani. *Parallel Algorithms and Dynamic Data Structures on the Graphics Processing Unit: a warp-centric approach*. PhD thesis, University of California, Davis, December 2017. URL https://escholarship.org/uc/item/5qd0r4ws.

[5] Saman Ashkiani, Andrew A. Davidson, Ulrich Meyer, and John D. Owens. GPU multisplit: an extended study of a parallel algorithm. *ACM Transactions on Parallel Computing*, 4(1):2:1–2:44, August 2017. doi: 10.1145/3108139. URL http://escholarship.org/uc/item/2kc8q23h.

[6] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. A dynamic hash table for the GPU. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2018, pages 419–429, May 2018. doi: 10.1109/IPDPS.2018.00052. URL https://escholarship.org/uc/item/2p48q0zg.

[7] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. Dynamic graphs on the GPU. In *Proceedings of the 34th IEEE International Parallel and Distributed Processing Symposium*, IPDPS 2020, pages 739–748, May 2020. doi: 10.1109/IPDPS47924.2020.00081.

[8] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, Martín Farach-Colton, and John D. Owens. Analyzing and implementing GPU hash tables. In *SIAM Symposium on Algorithmic Principles of Computer Systems*, APOCS23, pages 33–50, January 2023. doi: 10.1137/1.9781611977578.ch3. URL https://escholarship.org/uc/item/6cb1q6rz.

[9] Bruce G. Baumgart. Winged edge polyhedron representation. Technical Report STAN-CS-72-320, Stanford University Computer Science Department, Stanford, CA,

USA, October 1972. URL https://apps.dtic.mil/dtic/tr/fulltext/u2/755141.pdf.

[10] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Trans. Graph.*, 35(2), May 2016. ISSN 0730-0301. doi: 10.1145/2892632.

[11] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 308–317, New York, NY, USA, June 2012. Association for Computing Machinery. ISBN 9781450312134. doi: 10.1145/2312005.2312058.

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

[13] Mario Botsch and Leif Kobbelt. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, pages 185–192, New York, NY, USA, July 2004. Association for Computing Machinery. ISBN 3905673134. doi: 10.1145/1057432.1057457.

[14] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon Mesh Processing*. AK Peters / CRC Press, 1 edition, September 2010. doi: 10.1201/b10688. URL https://hal.inria.fr/inria-00538098.

[15] Erik Brisson. Representing geometric structures in d dimensions: Topology and order. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, pages 218–227, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913183. doi: 10.1145/73833.73858.

[16] Tyson Brochu and Robert Bridson. Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472–2493, 2009. doi: 10.1137/080737617.

[17] Tyson Brochu et al. El Topo: Robust topological operations for dynamic explicit surfaces, 2018. https://github.com/tysonbrochu/eltopo.

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

[19] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–11, December 1998. ISSN 1086-7651. doi: 10.1080/10867651.1998.10487494.

[20] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Rectangular multi-chart geometry images. In *Symposium on Geometry Processing*, SGP06, pages 181–190, June 2006. ISBN 3-905673-24-X. doi: 10.2312/SGP/SGP06/181-190.

[21] Zhenghai Chen and Tiow-Seng Tan. Computing three-dimensional constrained Delaunay refinement using the GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT 2019, pages 409–420, September 2019. doi: 10.1109/PACT.2019.00039.

[22] Siu-Wing Cheng and Tamal K Dey. Delaunay edge flips in dense surface triangulations. In *Proceedings of the 24th European Workshop on Computational Geometry*, EuroCG 2008, 2008. doi: 10.48550/arXiv.0712.1959.

[23] Nuttapong Chentanez, Matthias Müller, and Miles Macklin. GPU accelerated grid-free surface tracking. *Computers & Graphics*, 57:1–11, June 2016. ISSN 0097-8493. doi: 10.1016/j.cag.2016.03.002.

[24] Paolo Cignoni et al. VCGLib: The visualization and computer graphics library, 2023. URL `https://vcg.isti.cnr.it/vcglib//`.

[25] Luca Cirrottola and Algiane Froehly. Parallel unstructured mesh adaptation using iterative remeshing and repartitioning. Research Report RR-9307, INRIA Bordeaux, équipe CARDAMOM, November 2019. URL `https://inria.hal.science/hal-02386837`.

[26] David Cohen-Steiner, Pierre Alliez, and Mathieu Desbrun. Variational shape approximation. *ACM Transactions on Graphics*, 23(3):905–914, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015817.

[27] Narcís Coll and Marité Guerrieri. Parallel constrained Delaunay triangulation on the GPU. *International Journal of Geographical Information Science*, 31(7):1467–1484, July 2017. ISSN 1365-8816. doi: 10.1080/13658816.2017.1300804.

[28] Thomas Davies, Derek Nowrouzezahrai, and Alec Jacobson. On the effectiveness of weight-encoded neural implicit 3d shapes. Technical report, January 2021.

[29] Michael Dawson-Haggerty et al. trimesh, December 2019. URL `https://trimsh.org/`.

[30] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. 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, pages 317–324, USA, 1999. ACM Press/Addison-Wesley Publishing Co. ISBN 0201485605. doi: 10.1145/311535.311576.

[31] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve,

Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, page 12, November 2011. doi: 10.1145/2063384.2063396.

[32] Antonio DiCarlo, Alberto Paoluzzi, and Vadim Shapiro. Linear algebraic representation for topological structures. *Computer-Aided Design*, 46:269–274, 2014. ISSN 0010-4485. doi: 10.1016/j.cad.2013.08.044. 2013 SIAM Conference on Geometric and Physical Modeling.

[33] Ana Dodik, Oded Stein, Vincent Sitzmann, and Justin Solomon. Variational barycentric coordinates. *ACM Trans. Graph.*, 42(6), December 2023. ISSN 0730-0301. doi: 10.1145/3618403.

[34] Vladimir Dyedov, Navamita Ray, Daniel Einstein, Xiangmin Jiao, and Timothy J. Tautges. AHF: array-based half-facet data structure for mixed-dimensional and non-manifold meshes. *Engineering with Computers*, 31(3):389–404, July 2015. ISSN 1435-5663. doi: 10.1007/s00366-014-0378-6.

[35] Zachary Ferguson, Teseo Schneider, Danny M. Kaufman, and Daniele Panozzo. Intimestep remeshing for contacting elastodynamics. *ACM Trans. Graph.*, 42(4):145:1–145:15, August 2023. doi: 10.1145/3592428.

[36] Shachar Fleishman, Iddo Drori, and Daniel Cohen-Or. Bilateral mesh denoising. *ACM Transactions on Graphics*, 22(3):950–953, July 2003. ISSN 0730-0301. doi: 10.1145/882262.882368.

[37] Epic Games. Nanite virtualized geometry, December 2023. URL `https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine`.

[38] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 209–216, USA, August 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0897918967. doi: 10.1145/258734.258849.

[39] Mark Gillespie, Nicholas Sharp, and Keenan Crane. Integer coordinates for intrinsic geometry processing. *ACM Trans. Graph.*, 40(6), December 2021. ISSN 0730-0301. doi: 10.1145/3478513.3480522.

[40] Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983. ISSN 0164-0925. doi: 10.1145/69624.357206.

[41] Maurice Herlihy, Nir Shavit, and Michael Spear Victor Luchangco. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2021. ISBN 978-0-12-415950-1.

[42] X. Hu, X. Fu, and L. Liu. Advanced hierarchical spherical parameterizations. *IEEE Transactions on Visualization and Computer Graphics*, 24(6):1930–1941, June 2018. ISSN 1077-2626. doi: 10.1109/TVCG.2017.2704119.

[43] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics*, 38(6):201:1–201:16, November 2019. doi: 10.1145/3355089.3356506.

[44] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. Pumi: Parallel unstructured mesh infrastructure. *ACM Trans. Math. Softw.*, 42(3):17:1–17:28, May 2016. ISSN 0098-3500. doi: 10.1145/2814935.

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

[46] Zhongshi Jiang, Jiacheng Dai, Yixin Hu, Yunfan Zhou, Jeremie Dumas, Qingnan Zhou, Gurkirat Singh Bajwa, Denis Zorin, Daniele Panozzo, and Teseo Schneider. Declarative specification for unstructured mesh editing algorithms. *ACM Transactions on Graphics*, 41(6), November 2022. ISSN 0730-0301. doi: 10.1145/3550454.3555513.

[47] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, 16(3):64–77, May 1996. ISSN 1558-1756. doi: 10.1109/38.491187.

[48] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 89–99, New York, NY, USA, July 2013. Association for Computing Machinery. ISBN 9781450321358. doi: 10.1145/2492045.2492055.

[49] Bernhard Kerbl, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg, and Markus Steinberger. 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):29:1–29:16, August 2018. ISSN 2577-6193. doi: 10.1145/3233302.

[50] Lutz Kettner. Halfedge data structures. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019. URL `https://doc.cgal.org/4.14/Manual/packages.html#PkgHalfedgeDS`.

[51] Doyub Kim, Minjae Lee, and Ken Museth. NeuralVDB: High-resolution sparse volume representation using hierarchical neural networks. *ACM Trans. Graph.*, 43(2), February 2024. ISSN 0730-0301. doi: 10.1145/3641817.

[52] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2), March 2016. ISSN 0730-0301. doi: 10.1145/2866569.

[53] Naimin Koh, Wenjing Zhang, Jianmin Zheng, and Yiyu Cai. GPU-based multiple-choice scheme for mesh simplification. In *Proceedings of Computer Graphics International 2018*, CGI 2018, pages 195–200. ACM, June 2018. doi: 10.1145/3208159.3208195.

[54] Daniel Kopta, Thiago Ize, Josef Spjut, Erik Brunvand, Al Davis, and Andrew Kensler. Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 197–204, New York, NY, USA, March 2012. Association for Computing Machinery. ISBN 9781450311946. doi: 10.1145/2159616.2159649.

[55] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 42 of *PLDI '07*, pages 211–222, New York, NY, USA, June 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1273442.1250759.

[56] Bastian Kuth, Max Oberberger, Matthäus Chajdas, and Quirin Meyer. Edge-friend: Fast and deterministic Catmull-Clark subdivision surfaces. *Computer Graphics Forum*, 42 (8):8, August 2023. doi: 10.1111/cgf.14863.

[57] Oliver Laric. Three D Scans, 2023. `https://threedscans.com/`.

[58] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics*, 25(3):579–588, July 2006. doi: 10.1145/1141911.1141926.

[59] Hao Li, Takayuki Yamada, Pierre Jolivet, Kozo Furuta, Tsuguo Kondoh, Kazuhiro Izui, and Shinji Nishiwaki. Full-scale 3d structural topology optimization using adaptive mesh refinement based on the level-set method. *Finite Elements in Analysis and Design*, 194, October 2021. ISSN 0168-874X. doi: 10.1016/j.finel.2021.103561.

[60] Hsueh-Ti Derek Liu, Mark Gillespie, Benjamin Chislett, Nicholas Sharp, Alec Jacobson, and Keenan Crane. Surface simplification using intrinsic error metrics. *ACM Trans. Graph.*, 42(4):118:1–118:17, July 2023. ISSN 0730-0301. doi: 10.1145/3592403.

[61] Stuart P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982. doi: 10.1109/TIT.1982.1056489.

[62] A. Loseille, F. Alauzet, and V. Menier. Unique cavity-based operator and hierarchical domain partitioning for fast parallel generation of anisotropic meshes. *Computer-Aided Design*, 85:53–67, 2017. ISSN 0010-4485. doi: 10.1016/j.cad.2016.09.008. 24th International Meshing Roundtable Special Issue: Advances in Mesh Generation.

[63] Adrien Loseille and Rainald Löhner. Cavity-based operators for mesh adaptation. In *51st AIAA Aerospace Sciences Meeting*, number AIAA 2013-0152, January 2013. doi: 10.2514/6.2013-152.

[64] Adrien Loseille and Victorien Menier. Serial and parallel mesh modification through a unique cavity-based primitive. In Josep Sarrate and Matthew Staten, editors, *Proceedings of the 22nd International Meshing Roundtable*, pages 541–558. Springer International Publishing, 2014. ISBN 978-3-319-02335-9. doi: 10.1007/978-3-319-02335-9_30.

[65] Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. Dynamic mesh processing on the GPU. *Under review for SIGGRAPH Asia 2024*. URL https://escholarship.org/uc/item/1sm051d2.

[66] Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. RXMesh: A GPU mesh data structure. *ACM Transactions on Graphics*, 40(4):104:1–104:16, August 2021. doi: 10.1145/3450626.3459748. URL https://escholarship.org/uc/item/8r5848vp.

[67] M. Mäntylä. *Introduction to Solid Modeling*. W. H. Freeman & Co., New York, NY, USA, 1988. ISBN 0-88175-108-1.

[68] Nelson Max. Weights for computing vertex normals from facet normals. *Journal of Graphics Tools*, 4(2):1–6, March 1999. doi: 10.1080/10867651.1999.10487501.

[69] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys*, 48(2), October 2015. ISSN 0360-0300. doi: 10.1145/2818185.

[70] Massimiliano Meneghin, Ahmed H. Mahmoud, Pradeep Kumar Jayaraman, and Nigel J. W. Morris. Neon: A multi-GPU programming model for grid-based computations. In *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium*, pages 817–827, June 2022. doi: 10.1109/IPDPS53621.2022.00084. URL https://escholarship.org/uc/item/9fz7k633.

[71] D. Mlakar, M. Winter, P. Stadlbauer, H.-P. Seidel, M. Steinberger, and R. Zayer. Subdivision-specialized linear algebra kernels for static and dynamic mesh connectivity on the GPU. *Computer Graphics Forum*, 39(2):335–349, May 2020. doi: 10.1111/cgf.13934.

[72] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. Accelerating unstructured mesh point location with RT cores. *IEEE Transactions on Visualization and Computer Graphics*, 28(8):2852–2866, December 2020. doi: 10.1109/TVCG.2020.3042930.

[73] J. S. Mueller-Roemer, C. Altenhofen, and A. Stork. Ternary sparse matrix representation for volumetric mesh subdivision and processing on GPUs. *Computer Graphics Forum*, 36(5):59–69, 2017. doi: 10.1111/cgf.13245.

[74] Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. Spectral conformal parameterization. In *Proceedings of the Symposium on Geometry Processing*, SGP '08, pages 1487–1494. The Eurographics Association and Blackwell Publishing Ltd, 2008. doi: 10.5555/1731309.1731335.

[75] Rahul Narain, Armin Samii, and James F. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Transactions on Graphics*, 31(6):152:1–152:10, November 2012. doi: 10.1145/2366145.2366171.

[76] Maxim Naumov and Timothy Moon. Parallel spectral graph partitioning. Technical Report NVR-2016-001, NVIDIA Research, March 2016. `https://research.nvidia.com/publication/parallel-spectral-graph-partitioning`.

[77] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, March/April 2008. doi: 10.1145/1365490.1365500.

[78] The Boost organization. Boost C++ libraries, December 2023. URL `https://www.boost.org/`.

[79] Alexandros Papageorgiou and Nikos Platis. Triangular mesh simplification on the GPU. *The Visual Computer*, 31(2):235–244, November 2014. ISSN 1432-2315. doi: 10.1007/s00371-014-1039-x.

[80] Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. Adaptive tearing and cracking of thin sheets. *ACM Trans. Graph.*, 33(4), July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601132.

[81] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4), April 2008. doi: 10.5555/1352079.1352134.

[82] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530. ACM Press, June 2013. ISBN 9781450320146. doi: 10.1145/2491956.2462176.

[83] Luciano A. Romero Calla, Lizeth J. Fuentes Perez, and Anselmo A. Montenegro. A minimalistic approach for fast computation of geodesic distances on triangular meshes. *Computers & Graphics*, 84:77–92, 2019. ISSN 0097-8493. doi: 10.1016/j.cag.2019.08.014.

[84] H. Schäfer, B. Keinert, M. Nießner, and M. Stamminger. Local painting and deformation of meshes on the GPU. *Computer Graphics Forum*, 34(1):26–35, August 2014. ISSN 0167-7055. doi: 10.1111/cgf.12456.

[85] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, Department of Computer Science, 1997. `http://glaros.dtc.umn.edu/gkhome/node/87`.

[86] Nicholas Sharp, Yousuf Soliman, and Keenan Crane. Navigating intrinsic triangulations. *ACM Trans. Graph.*, 38(4):55:1–55:16, July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322979.

[87] Nicholas Sharp, Mark Gillespie, and Keenan Crane. Geometry processing with intrinsic triangulations. In *ACM SIGGRAPH 2021 Courses*, SIGGRAPH '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383615. doi: 10.1145/3450508.3464592.

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

[89] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1):21–74, 2002. ISSN 0925-7721. doi: 10.1016/S0925-7721(01)00047-5. 16th ACM Symposium on Computational Geometry.

[90] Smithsonian Institution Digitization Program Office. Smithsonian 3D digitization, 2023. `https://3d.si.edu/`.

[91] Robert F. Tobler and Stefan Maierhofer. A mesh data structure for rendering and subdivision. In *Proceedings of WSCG (International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision)*, pages 157–162, February 2006. URL `http://www.vrvis.at/publications/PB-VRVis-2006-007`.

[92] Christos Tsolakis, Polykarpos Thomadakis, and Nikos Chrisochoides. Tasking framework for adaptive speculative parallel mesh generation. *The Journal of Supercomputing*, 78(5):1–32, April 2022. doi: 10.1007/s11227-021-04158-9.

[93] Ingo Wald, Will Usher, Nate Morrical, Laura Lediaev, and Valerio Pascucci. RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh pointlocation. In *High-Performance Graphics - Short Papers*, 2019. doi: 10.2312/hpg.20191189.

[94] Bin Wang, Ingo Wald, Nate Morrical, Will Usher, Lin Mu, Karsten Thompson, and Richard Hughes. An GPU-accelerated particle tracking method for Eulerian–Lagrangian simulations using hardware ray tracing cores. *Computer Physics Communications*, 271: 108221, 2022. ISSN 0010-4655. doi: 10.1016/j.cpc.2021.108221.

[95] 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. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing*, 4(1):3:1–3:49, August 2017. doi: 10.1145/3108140.

[96] Ruben Wiersma, Ahmad Nasikun, Elmar Eisemann, and Klaus Hildebrandt. A fast geometric multigrid method for curved surfaces. In *ACM SIGGRAPH 2023 Conference Proceedings*, SIGGRAPH '23, New York, NY, USA, July 2023. Association for Computing Machinery. ISBN 9798400701597. doi: 10.1145/3588432.3591502.

[97] Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. MeshTaichi: A compiler for efficient mesh-based operations. *ACM Transactions on Graphics*, 41(6), November 2022. ISSN 0730-0301. doi: 10.1145/3550454.3555430.

[98] Yizhou Yu, Kun Zhou, Dong Xu, Xiaohan Shi, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Mesh editing with poisson-based gradient field manipulation. *ACM Transactions on Graphics*, 23(3):644–651, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015774.

[99] Rhaleb Zayer, Markus Steinberger, and Hans-Peter Seidel. A GPU-adapted structure for unstructured grids. In *Computer Graphics Forum (Proceedings of Eurographics 2017)*, volume 36, pages 495–507, May 2017. doi: 10.1111/cgf.13144.

[100] Wei Zhao, Shuming Gao, and Hongwei Lin. A robust hole-filling algorithm for triangular mesh. In *2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics*, pages 22–22, December 2007. doi: 10.1109/CADCG.2007.4407836.

[101] Qingnan Zhou and Alec Jacobson. Thingi10K: A dataset of 10,000 3d-printing models. *CoRR*, May 2016.

[102] Bo Zhu and Lixu Gu. A hybrid deformable model for real-time surgical simulation. *Computerized Medical Imaging and Graphics*, 36(5):356–365, July 2012. ISSN 0895-6111. doi: 10.1016/j.compmedimag.2012.03.001.