Dynamic Mesh Processing on the GPU

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

We propose a system for dynamic triangle mesh processing entirely on the GPU. Our system offers an efficient data structure that allows fast updates of the underlying mesh connectivity and attributes. Our data structure partitions the mesh into small patches which allows processing all dynamic updates for each patch within the GPU's fast shared memory. This allows us to rely on *speculative processing* for conflict handling, which has low rollback cost while maximizing parallelism and reducing the cost of locking. Our system also introduces a new programming model for dynamic mesh processing. The programming model offers concise semantics for dynamic updates, relieving the user from having to worry about conflicting updates in the context of parallel execution. Our programming model relies on the *cavity operator*, which is a general mesh update operator that formulates any dynamic operation as an element reinsertion by removing a set of mesh elements and inserting others in the created void. We used our system to implement Delaunay edge flips and isotropic remeshing applications on the GPU. Our system achieves a 3–18x speedup on large models compared to multithreaded CPU solutions. Despite our additional dynamic features, our data structure also outperforms state-of-the-art GPU static data structures in terms of speed and memory requirements.

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

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

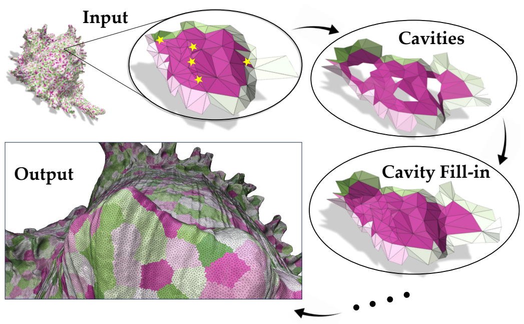


Fig. 1. Our system casts local, dynamic mesh operations as *cavitiy operators* which provide intuitive semantics for mesh updates (e.g., edge collapse here, during one iteration of isotropic remeshing). With the cavity operator, the user defines mesh updates by creating cavities and then fills in these cavities with new elements while our system handles potential conflicts. Additionally, we have designed a data structure for updating triangle meshes that primarily uses the GPU's shared memory for better locality. Our approach significantly speeds up dynamic mesh processing compared to multi-threaded CPU solutions and also results in better performance on static mesh processing tasks over existing state-of-the-art GPU solutions.

1 INTRODUCTION

The field of 3D geometric data processing, traditionally applied to simulation, visualization, and computer-aided design (CAD), is witnessing a surge in interest thanks to the demand for systems that manipulate unstructured meshes. Geometry processing applications include shape analysis and synthesis, computational design, virtual reality, and 3D printing. Moreover, the growing influences of machine learning and data-driven algorithmic design have led to breakthrough developments in the field as well as applications to computer vision and AI-driven design of virtual assets. Despite the growing influence of geometric data processing and recent coupling to machine learning tools, most geometry processing algorithms are implemented using serial processes on the CPU.

In computational modeling and simulation, there is an ongoing demand for *dynamic* mesh processing. The importance of having meshes that can adapt in real-time, altering their structures in response to the stimuli of the operation or simulation, can be seen in many applications. 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 [Antepara et al. 2021]. Similarly, in materials science, simulations that deal with crack propagation or material failures often hinge on the ability of the mesh to refine around the evolving crack tip, ensuring that the details of the propagation pathway are well-represented [Pfaff et al. 2014]. In topology optimization [Li et al. 2021], 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 dynamic mesh processing include real-time interactive applications like surgical simulation [Zhu and Gu 2012] and cloth manipulation [Narain et al. 2012]. This situation explains the multitude of libraries for dynamic mesh processing (e.g., CGAL [Kettner 2019], OpenMesh [Botsch et al. 2002], and VCGlib [Cignoni et al. 2023]) that have significantly lowered the entry bar to facilitate efficient geometric data processing.

However, existing libraries are predominantly single-core CPU-based. With the increase in size of geometric data, CPU/serial solutions for processing geometric data are no longer sufficient to meet the needs of performance and interactivity. One notable exception is the Wild Meshing Toolkit (WMTK) [Jiang et al. 2022], which leverages the parallelism of multi-threaded CPU systems. However, even multicore CPUs cannot leverage the full parallelism in mesh processing applications. Complex highly-detailed meshes contain millions of mesh elements, while the most powerful CPU offers only a few hundreds of parallel threads (e.g., AMD EPYC 9004 Series offers 128 cores leading to 256 threads with hyperthreading [Advanced Micro Devices, Inc. (AMD) 2023]). Thus, the limited parallel processing power in multithreaded CPUs is insufficient for handling the increasingly complex and large-scale meshes required in modern

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

computational applications, necessitating the exploration of more powerful and parallel computing architectures.

The inherent data-parallel nature of mesh processing makes it perfectly suited for execution on the GPU. GPUs offer a vast number of processing cores that can concurrently execute computations, allowing for significant acceleration in mesh processing tasks. 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. This principled approach has been successfully applied in at least two recently introduced systems: RXMesh [Mahmoud et al. 2021] and MeshTaichi [Yu et al. 2022]. These systems offer general-purpose solutions for mesh processing by designing efficient data structures and generic programming models that cover a large set of applications. However, these systems are limited to static mesh processing where the mesh topology does not change.

Current solutions for dynamic mesh processing on the GPU are application-specific. Examples of these applications include surface tracking [Chentanez et al. 2016], mesh simplification [Koh et al. 2018; Papageorgiou and Platis 2014], mesh subdivision [Kuth et al. 2023], and Delaunay refinement [Chen and Tan 2019]. Solutions within these applications do not generalize well to other applications; e.g., a GPU data structure that is tailored for mesh simplification cannot be used for refinement as each poses different challenges. 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.

While processing unstructured meshes has ample latent parallelism across the millions of geometric elements in a detailed mesh, that processing involves complex dependencies, synchronization, and many levels of memory reference indirections, potentially leading to inefficient utilization of massively parallel hardware. More specifically, building a dynamic unstructured mesh processing on the GPU requires tackling the following challenges:

(1) **Locality**: The majority of dynamic mesh operations change a local neighborhood in the mesh. 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.

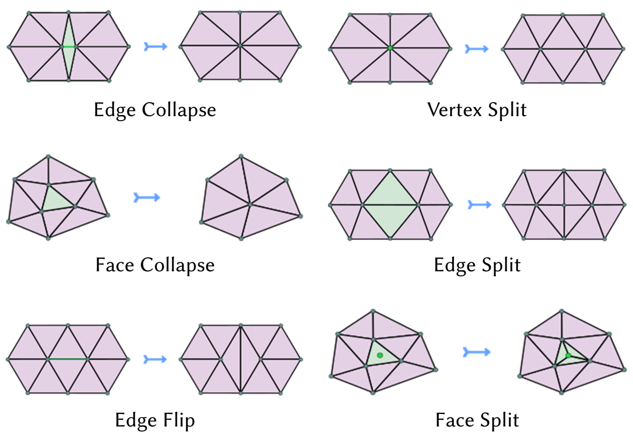


Fig. 2. Examples of dynamic triangle mesh local operators

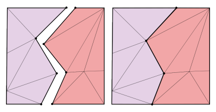
In this paper, we propose a new dynamic mesh processing system that operates entirely on the GPU. Our system leverages the GPU's parallelism for high-performance generic dynamic triangle mesh updates and operations by tackling the above challenges. In summary, our system achieves the following design 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 delivers better performance to state-of-the-art GPU static mesh systems, e.g., RXMesh [Mahmoud et al. 2021].
(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 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 2x less memory than RXMesh [Mahmoud et al. 2021]. This compact data structure requires less bookkeeping and exhibits greater locality, both leading to higher performance.

Using our system, we implemented two dynamic geometry processing applications i.e., Delaunay edge flip and isotropic remeshing. In comparison with the state-of-the-art multithreaded CPU framework, our system speedup is between 3–18x on large meshes with millions of faces. Besides memory efficiency, our data structure outperforms state-of-the-art GPU static mesh data structure with a geometric mean speedup of 1.5x.


Mesh zippering

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 [Mlakar et al. 2020], nor does it support operations with non-topologically local are of impact for update operations, e.g., mesh "zippering" [Brochu and Bridson 2009] (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. Finally, while we have implemented our system using CUDA and use CUDA terminology throughout, the concepts presented are general and are applicable to any GPU architecture and GPU programming language. We will release our code as an open source upon acceptance.

2 RELATED WORK

2.1 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 in computer graphics. 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 [Baumgart 1972], 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 [Baumgart 1972] stores adjacency information, enabling efficient navigation across the mesh by linking faces and vertices to edges. The Halfedge data structure [Mäntylä 1988]—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 [Kettner 2019]. The Quad-edge structure [Guibas and Stolfi 1985] extends this concept by efficiently representing the topology of non-manifold surfaces. The Cell-tuple [Brisson 1989] is used for higher-dimensional meshes, providing a more flexible representation for complex geometries. Recently, Linear Algebraic Representation (LAR) [DiCarlo et al. 2014] 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. In this work we adopt LAR, in part, as described by Mahmoud et al. [2021] in RXMesh due to its compactness and suitability for the GPU, but with modifications that further reduce memory use over RXMesh by 50% (see Appendix A).

2.2 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 mulicore 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, deal with attributes, and to maintain correspondence between the geometric representation and its discretized mesh representation [Cirrottola and Froehly 2019; Ibanez et al. 2016].

In an effort to leverage existing codes, Cirrottola and Froehly [2019] design 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. As we see in Section 4, we do not repartition our mesh, but rather modify our patch boundaries as necessary to ensure all mesh operations occur within a single patch.

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. [2022] 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 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. [2022] 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 [Ibanez et al. 2016] 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.

2.3 GPU Mesh Data Structure

Only recently, GPU mesh data structures started gaining traction to make the best use of the modern GPU. The goal of these data structures is to provide the same level of ease as CPU-based data structures while also exploiting the GPU massive parallelism. Mesh Matrix [Zayer et al. 2017] is built upon the foundation of the LAR representation for surface meshes. In Mesh Matrix, the relationship between faces and vertices is captured through a sparse matrix, This method achieves a compact form by utilizing a singular array complemented by an *action map*, a concise local structure detailing vertex interactions. This configuration of Mesh Matrix forgoes the necessity of generating intermediate data. RXMesh [Mahmoud et al. 2021] subdivides the mesh into *patches* and prioritizes efficient utilization of the GPU's memory hierarchy by confining most of the computation in the GPU's shared memory. RXMesh extends each patch with ghost-cells, called "ribbons", to improve data locality when accessing out-of-patch neighbor mesh elements.

2.4 Domain Specific Languages (DSL)

DSL and compiler techniques can be used to improve portability across different architectures. For example, MeshTaichi [Yu et al. 2022] takes the idea of partitioning the mesh and implements a compiler and DSL for mesh-based operations with an intuitive programming model. The user writes a single code that is deployed on either GPU or CPU. By inspecting the user code during compilation, MeshTaichi can precompute the "wanted" queries during compile time, allowing them to leverage the GPU's shared memory to cache mesh attributes. Liszt [DeVito et al. 2011] 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, SMPs, and GPUs. Ebb [Bernstein et al. 2016] 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 could deliver state-of-the-art results for static meshes, 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 [Kulkarni et al. 2007].

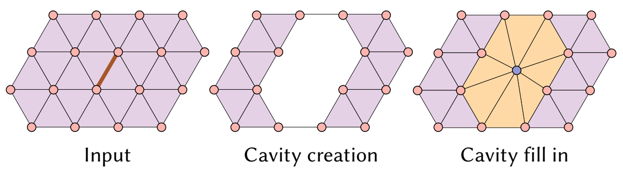


Fig. 3. An example of edge collapse, cast as a cavity-based operator.

2.5 Dynamic Mesh Processing Programming Model

While DSLs are restricted to static mesh processing, different runtime libraries expose different programming models for manipulating meshes. The most widely used approach for exposing mesh manipulation is through *local operators*, e.g., edge flip [Botsch et al. 2010, 2002; Cignoni et al. 2023; Dawson-Haggerty et al. 2019; Kettner 2019]. 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 [Loseille and Löhner 2013] 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 [Abdelkader et al. 2017] 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 PROGRAMMING MODEL

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

- 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
- Propagate dynamic topological changes to mesh attributes

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 [Botsch et al. 2010] and CGAL [Kettner 2019]. 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 [Botsch et al. 2010], CGAL [Kettner 2019], OpenMesh [Botsch et al. 2002], WMTK [Jiang et al. 2022], libigl [Jacobson et al. 2018]) and found no consistency in the set

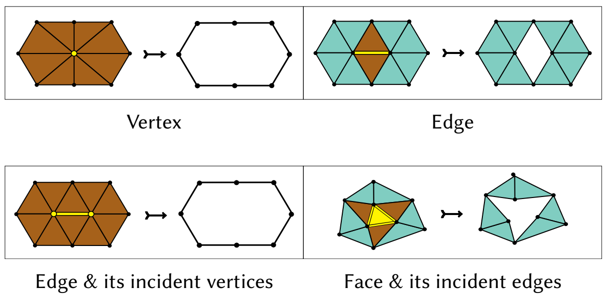
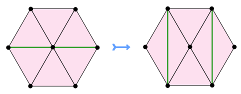


Fig. 4. Examples of cavity templates provided by our system. The seed for each cavity is highlighted in yellow and the deleted neighborhood is shown in brown.

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 [Jiang et al. 2022] 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. [2022] 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 different local operators where the locking region is user-defined. For example, not all dynamic mesh libraries offer 1⟶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⟶3 triangle split operator, users might assume that they do not need to lock the vertices of the split 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* [Loseille and Menier 2014] 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 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 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). The cavity-based operator provides the right ingredients for an extensible framework, 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 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 4 shows a subset of templates that our system offers. Note that using these templates is entirely optional.

After the underlying system resolves conflicts, it returns a set of conflict-free cavities to the user. This set 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.

4 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 3) while achieving our design goals (Section 1). Here we discuss the design principles we followed in designing the data structure (Section 4.1) and the scheduler (Section 4.2). Finally, we discuss how the whole system works (Section 4.3).

4.1 Data Structure

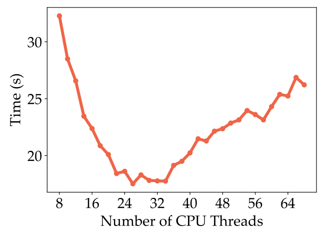
Maximize Locality. In a design where all mesh data is stored in global GPU memory, mesh-based operations are mostly out-of-cache. Topological query operations involve multiple levels of memory indirection, frustrating attempts at exploiting locality. Geometric information (i.e., mesh attributes) is hard to coalesce when neighbor attributes are accessed, leading to irregular memory accesses. To mitigate this problem, state-of-the-art unstructured GPU mesh data structures [Mahmoud et al. 2021; Yu et al. 2022] rely on 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. In this work, we follow a similar approach, 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 5.1 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* [Kulkarni et al. 2007], as we will discuss in Section 4.2. 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 [Kulkarni et al. 2007]).

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 5.2).

Cavities that cross a patch boundary pose a challenge since detecting and resolving their 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, 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 our initial 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), and can be easily coalesced. In contrast, our initial approach required writing topological information (i.e., face and edge connectivity).

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 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. The inset shows the result of running the Delaunay Edge Flip application (Section 6.1) on a model with 2M faces on a 64-Core AMD EPYC 7742 while varying the number of threads (i.e., strong scaling).



Strong scaling of WMTK [Jiang et al. 2022] on Delaunay Edge Flip application

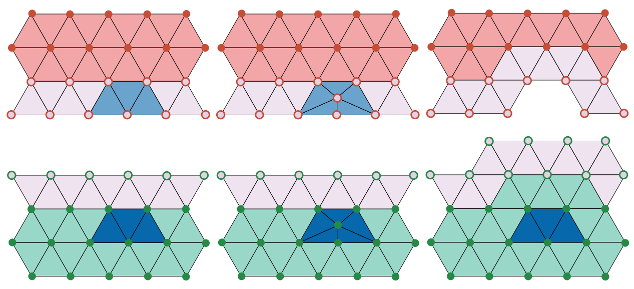


Fig. 5. Illustrating inter-patch conflicts between two patches (top q, bottom p), where ribbon elements are shown in a lighter color. Note the bottom row of the top patch (the ribbon) represents the same mesh elements as the top row of the bottom patch. Potential conflicts occur when a cavity (bottom 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 don't need to write cavity fill-in in q. Our solution also maximizes parallelism, as we lock q for a shorter time, allowing other blocks to process q afterward and concurrently while we perform cavity fill-in on p.

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.

4.2 Scheduler

In the below 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. Static GPU mesh processing systems rely on the GPU hardware scheduler, i.e., they 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.

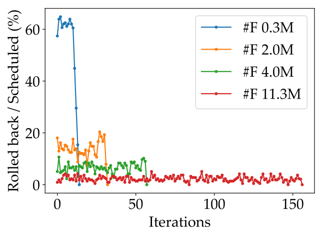
(1) Serialization. One potential scheduler-based solution to conflicts is to serialize conflicting updates. Such a solution might be fine for limited-parallelism environments such as a desktop multi-threaded CPU, where a relatively few number of patches can be executed concurrently. However, a modern 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 [Loseille et al. 2017] 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. 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. Speculative processing was characterized as "the only plausible approach to parallelizing many, if not most, irregular applications" [Kulkarni et al. 2007]. 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 detecting conflicts, which happens if a cavity crosses the patch boundary and thus the neighbor patch should not be processed concurrently. However, this cost

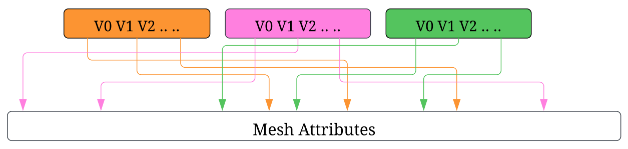


is inevitable and would be paid for other alternative designs as well. The second is the cost of rolling back. In our design, this cost is low, as it only requires discarding changes in shared memory (Section 4.1) for conflicting updates without any impact on the global state of the data structure. The 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. The inset shows the ratio of discarded patches vs. the number of scheduled patches for different input meshes. Here, we assume the worst-case scenario where every patch has a dependency on all its neighbor patches. With the smallest meshes, the cost is high (∼50%) due to limited parallelism since the number of SMs in this case is higher than the number of patches. As the mesh size increases, the fraction of wasted work becomes negligible (less than 2%).

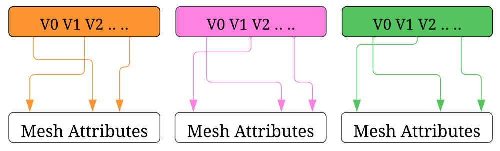
4.3 Putting it All Together

Finally, we discuss the high-level architecture of our system and how we put the above design principles into practice. We defer implementation details to the next section. The main controller in our system is the scheduler, which runs in a loop. On each iteration in the loop, the scheduler launches as many blocks as possible to maximize GPU occupancy, while avoiding processing two neighbor patches at the same time. This strategy minimizes the cost of speculation. The loop terminates when all patches in the mesh have been processed.

Within each iteration, the scheduler assigns blocks to patches. Once a block is assigned to a patch, threads within the block call a user-defined (or system-defined for the predefined cavity templates) function to create cavities within the patch. Such a function may involve both query operations and accessing the mesh attributes. Next, our system resolves intra-patch conflicts before attempting to resolve inter-patch conflicts. By imposing the constraint that a mesh element belongs to a single cavity, our system resolves intra-patch conflicts and proceeds with a subset of conflict-free cavities within the patch. The user can attempt running the cavities that do not proceed later. Then, our system resolves the inter-patch conflicts by checking if there are cavities that have an imprint on neighbor patches. If at least one cavity has an imprint, the whole block cooperates to expand the patch. Using the whole block to compute here allows coalesced accesses and reduces thread divergence. Expanding a patch requires locking the neighbor patches before reading from them. If locking fails, then the patch is discarded. Our system will reschedule the discarded patches in subsequent iterations. Otherwise, if the patch successfully locks its neighbor patches, we proceed with expanding the patch and write this information in shared memory. After a successful expansion, we unlock the neighbor patches, allowing them to be processed by other blocks. Finally, the block calls the user-specified cavity fill-in before finally committing the patch to global memory.



(a) RXMesh's approach: accessing attributes allocated as a single array



(b) Our approach: accessing attributes allocated per patch

Fig. 6. 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).

5 IMPLEMENTATION DETAILS

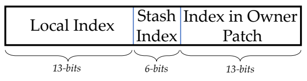
5.1 Patch Data Structure

Mesh Topology. As mentioned in Section 4.1, 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 ($\mathcal{EV}$). Since the patch is small, we 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 Mahmoud et al. [2021] for more details). In addition, 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. RXMesh stores mesh attributes as a single array in global memory. To access this array, RXMesh requires mapping 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, inspired by MeshTaichi. We localize mesh attributes by allocating them on a per-patch basis. With per-patch allocation, we eliminate the need for the local-to-global mapping (Figure 6). 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, it divides mesh elements into owned and ribbon elements, and that status never changes during computation. Consequently, ribbon elements can be assigned consecutive indices and their storage can be 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 [Awad et al. 2023]. 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 about the upper bound of 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 access into the hashtable when the mesh element is owned, i.e., optimizing for the common case.

5.2 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 4.3).

Cavity Registration. Our system provides predefined *templates* that cover a wide range of cavity configurations (Figure 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 [Blelloch et al. 2012] 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 4.1, 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. 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 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 [Gottlieb et al. 1983] 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 [Herlihy et al. 2021], 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 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 [Pheatt 2008] or Boost [organization 2023]), 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. Thus, we implemented a simple spinlock locking algorithm with a backoff strategy [Herlihy et al. 2021] that achieves our design goals. To reduce contention on the atomic operations used in the spin lock, 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.

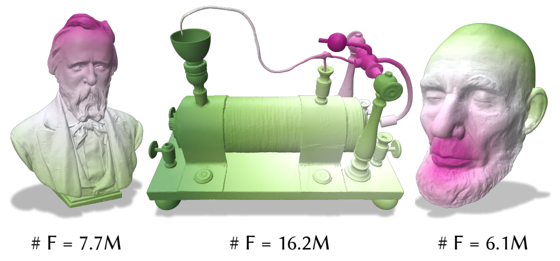


Fig. 7. Examples of geodesic distance computation of large models

Table 1. Time and speedup of our system against RXMesh running geodesic distance computation [Romero Calla et al. 2019].

# 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**

6 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 the static applications performance by comparing against RXMesh [Mahmoud et al. 2021]. For other dynamic applications, we compare against WMTK [Jiang et al. 2022] 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 [2023] and ThreeDScans [Laric 2023] repositories since they feature meshes with millions of faces. We conduct all experiments on an A100 GPU with 40 GB of memory using CUDA 12.2 from an NVIDIA DGX machine featuring an AMD EPYC 7742 64-core 2.25 GHz Processor with 1 TB main memory.

Comparison against static-only system. We begin with a static geodesic distance computation (Figure 7) to compare our system performance against RXMesh. Geodesic distance refers to the shortest path between a source vertex and all other vertices, traversing across the surface of a given mesh. For this computation, we employ a minimalistic parallel algorithm [Romero Calla et al. 2019], which approximates geodesic distances. This algorithm operates by sequentially computing geodesic distance, starting from vertices nearest to the source and progressively reaching those farther away. The algorithms first compute the topological level sets surrounding the source vertex, essentially calculating the number of *hops* required to reach each vertex. Subsequently, on the GPU, we process these vertices in batches, based on their topological distance, to calculate their geodesic distances and associated errors, all performed in parallel. This process involves activating a new group of vertices at each level and deactivating those that have completed their calculations,

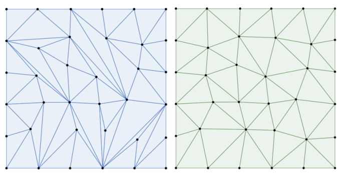
Table 2. Time and speedup of our system on the Delaunay edge flip application against Wild Mesh Toolkit (WMTK) [Jiang et al. 2022] using 32 CPU threads.

# F ($\times 10^6$)	WMTK (s)	Our (s)	Speedup
0.015	0.24	0.99	**0.24**
0.064	0.76	1.31	**0.57**
0.58	5.13	1.43	**3.6**
0.6	4.64	1.51	**3.1**
1.46	13.71	1.19	**11.4**
6.1	81.79	4.97	**16.4**
8.5	175.57	9.98	**17.6**
19.8	200.142	10.72	**18.6**

allowing for data propagation through the mesh. Our system outperforms RXMesh in this static application with an average geometric mean speedup of 1.53x (Table 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 tradeoff 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. In addition, our data structure for *static* applications requires half the memory vs. RXMesh (Appendix A) using the same patch size. In contrast, 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.

Next, we assess the efficacy of our system through two distinct dynamic applications, each targeting a specific aspect of our system's capabilities. The first application, Delaunay edge flip, maintains a consistent 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 sizes 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. Collectively, these applications offer a comprehensive insight into our system's performance across a range of scenarios commonly encountered in dynamic mesh processing applications.

6.1 Delaunay Edge Flip


Input & output of 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 (see the inset). 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 [Cheng and Dey 2008]. 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. Table 2 compares our system implementation against WMTK's best configuration (32 threads, see Section 4.1). 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 (here, those that exceed 0.5M faces), 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.

6.2 Isotropic Remeshing

Isotropic remeshing [Botsch and Kobbelt 2004] improves the quality of an input mesh by making the output triangles as equilateral as possible (Figure 1). 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. Thus, 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 3 compares our implementation against WMTK implementation; our system achieves a 1.5–6x speedup on meshes of at least 0.5M elements.

7 CONCLUSION AND FUTURE WORK

In this paper, 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 [Awad et al. 2020] and hash tables [Ashkiani et al. 2018]). While our system focuses on dynamic mesh processing, it also improves the performance of the static case. In our implementation

Table 3. Comparing the performance of our system against WMTK using three iterations of isotropic remeshing [Botsch and Kobbelt 2004] .

Input # F ($\times 10^6$)	Output # F ($\times 10^6$)	WMTK (s)	Ours (s)	Speedup
0.064	0.046	4.19	16.34	**0.25**
0.6	0.45	41.3	24.9	**1.65**
0.91	0.79	73.6	13.9	**5.29**
1.4	1.2	105.12	35.6	**2.94**
3.5	2.7	309.5	51.3	**6.03**
6.1	4.5	483.3	111.7	**4.32**

and application, 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 [Garland and Heckbert 1997], spherical parameterization [Hu et al. 2018], or Delaunay refinement [Shewchuk 2002]), 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. We plan to explore this trade-off in future work. Such a tradeoff was previously explored for multicore systems, e.g., Delaunay refinement [Pingali et al. 2011], where it was shown that in practice Delaunay refinement does not have to follow the priority as described in the serial implementation. We plan to expand this study into more geometric data processing applications specifically on the GPU.

REFERENCES

Ahmed Abdelkader, Ahmed H. Mahmoud, Ahmad A. Rushdi, Scott A. Mitchell, John D. Owens, and Mohamed S. Ebeida. 2017. A Constrained Resampling Strategy for Mesh Improvement. *Computer Graphics Forum* 36, 5 (July 2017), 189–201. https://doi.org/10.1111/cgf.13256 Proceedings of the Symposium on Geometry Processing.

Advanced Micro Devices, Inc. (AMD). 2023. *AMD EPYC™ 9004 Series Server Processors.* https://www.amd.com/content/dam/amd/en/documents/products/epyc/epyc-9004-series-processors-data-sheet.pdf.

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

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

Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, Martín Farach-Colton, and John D. Owens. 2023. Analyzing and Implementing GPU Hash Tables. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS23).* 33–50. https://doi.org/10.1137/1.9781611977578.ch3

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

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

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

Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. 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* (Pittsburgh, Pennsylvania, USA) *(SPAA '12).* Association for Computing Machinery, New York, NY, USA, 308–317. https://doi.org/10.1145/2312005.2312058

Mario Botsch and Leif Kobbelt. 2004. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing* (Nice, France) *(SGP '04).* Association for Computing Machinery, New York, NY, USA, 185–192. https://doi.org/10.1145/1057432.1057457

Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. 2010. *Polygon Mesh Processing.* AK Peters / CRC Press. 250 pages. https://hal.inria.fr/inria-00538098

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

Erik Brisson. 1989. Representing Geometric Structures in d Dimensions: Topology and Order. In *Proceedings of the Fifth Annual Symposium on Computational Geometry* (Saarbruchen, West Germany) *(SCG '89).* Association for Computing Machinery, New York, NY, USA, 218–227. https://doi.org/10.1145/73833.73858

Tyson Brochu and Robert Bridson. 2009. Robust Topological Operations for Dynamic Explicit Surfaces. *SIAM Journal on Scientific Computing* 31, 4 (2009), 2472–2493. https://doi.org/10.1137/080737617

Zhenghai Chen and Tiow-Seng Tan. 2019. Computing Three-Dimensional Constrained Delaunay Refinement Using the GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT 2019).* 409–420. https://doi.org/10.1109/PACT.2019.00039

Siu-Wing Cheng and Tamal K Dey. 2008. Delaunay edge flips in dense surface triangulations. In *Proceedings of the 24th European Workshop on Computational Geometry (EuroCG 2008).* https://arxiv.org/abs/0712.1959

Nuttapong Chentanez, Matthias Müller, and Miles Macklin. 2016. GPU accelerated grid-free surface tracking. *Computers & Graphics* 57 (June 2016), 1–11. https://doi.org/10.1016/j.cag.2016.03.002

Paolo Cignoni et al. 2023. VCGLib: The Visualization and Computer Graphics Library. https://vcg.isti.cnr.it/vcglib//

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

Michael Dawson-Haggerty et al. 2019. trimesh. https://trimsh.org/

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. 2011. 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).* 12. https://doi.org/10.1145/2063384.2063396

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

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

Allan Gottlieb, Boris D. Lubachevsky, and Larry Rudolph. 1983. Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors. *ACM Transactions on Programming Languages and Systems* 5, 2 (April 1983), 164–189. https://doi.org/10.1145/69624.357206

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

Maurice Herlihy, Nir Shavit, and Michael Spear Victor Luchangco. 2021. *The Art of Multiprocessor Programming.* Morgan Kaufmann.

X. Hu, X. Fu, and L. Liu. 2018. Advanced Hierarchical Spherical Parameterizations. *IEEE Transactions on Visualization and Computer Graphics* 24, 6 (June 2018), 1930–1941. https://doi.org/10.1109/TVCG.2017.2704119

Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. 2016. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Trans. Math. Softw.* 42, 3, Article 17 (May 2016), 28 pages. https://doi.org/10.1145/2814935

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

Zhongshi Jiang, Jiacheng Dai, Yixin Hu, Yunfan Zhou, Jeremie Dumas, Qingnan Zhou, Gurkirat Singh Bajwa, Denis Zorin, Daniele Panozzo, and Teseo Schneider. 2022. Declarative Specification for Unstructured Mesh Editing Algorithms. *ACM Transactions on Graphics* 41, 6, Article 251 (Nov. 2022), 14 pages. https://doi.org/10.1145/3550454.3555513

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

Naimin Koh, Wenjing Zhang, Jianmin Zheng, and Yiyu Cai. 2018. GPU-based Multiple-Choice Scheme for Mesh Simplification. In *Proceedings of Computer Graphics International 2018 (CGI 2018)*. ACM, 195–200. https://doi.org/10.1145/3208159.3208195

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07, Vol. 42)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/1273442.1250759

Bastian Kuth, Max Oberberger, Matthäus Chajdas, and Quirin Meyer. 2023. Edge-Friend: Fast and Deterministic Catmull-Clark Subdivision Surfaces. *Computer Graphics Forum* 42, 8 (Aug. 2023), 8. https://doi.org/10.1111/cgf.14863

Oliver Laric. 2023. *Three D Scans*. https://threedscans.com/.

Hao Li, Takayuki Yamada, Pierre Jolivet, Kozo Furuta, Tsuguo Kondoh, Kazuhiro Izui, and Shinji Nishiwaki. 2021. Full-scale 3D structural topology optimization using adaptive mesh refinement based on the level-set method. *Finite Elements in Analysis and Design* 194 (Oct. 2021). https://doi.org/10.1016/j.finel.2021.103561

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

Adrien Loseille and Rainald Löhner. 2013. Cavity-Based Operators for Mesh Adaptation. In *51st AIAA Aerospace Sciences Meeting*. 8 pages. https://doi.org/10.2514/6.2013-152

Adrien Loseille and Victorien Menier. 2014. Serial and Parallel Mesh Modification Through a Unique Cavity-Based Primitive. In *Proceedings of the 22nd International Meshing Roundtable*, Josep Sarrate and Matthew Staten (Eds.). Springer International Publishing, 541–558. https://doi.org/10.1007/978-3-319-02335-9_30

Ahmed H. Mahmoud, Serban D. Porumbescu, and John D. Owens. 2021. RXMesh: A GPU Mesh Data Structure. *ACM Transactions on Graphics* 40, 4, Article 104 (Aug. 2021), 16 pages. https://doi.org/10.1145/3450626.3459748

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

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

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

The Boost organization. 2023. Boost C++ Libraries. https://www.boost.org/

Alexandros Papageorgiou and Nikos Platis. 2014. Triangular mesh simplification on the GPU. *The Visual Computer* 31, 2 (Nov. 2014), 235–244. https://doi.org/10.1007/s00371-014-1039-x

Tobias Pfaff, Rahul Narain, Juan Miguel de Joya, and James F. O'Brien. 2014. Adaptive Tearing and Cracking of Thin Sheets. *ACM Trans. Graph.* 33, 4, Article 110 (July 2014), 9 pages. https://doi.org/10.1145/2601097.2601132

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

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 12–25. https://doi.org/10.1145/1993498.1993501

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

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

Smithsonian Institution Digitization Program Office. 2023. *Smithsonian 3D Digitization*. https://3d.si.edu/.

Christos Tsolakis, Polykarpos Thomadakis, and Nikos Chrisochoides. 2022. Tasking framework for adaptive speculative parallel mesh generation. *The Journal of Supercomputing* 78, 5 (2022), 1–32.

Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-based Operations. *ACM Transactions on Graphics* 41, 6, Article 252 (Nov. 2022), 17 pages. https://doi.org/10.1145/3550454.3555430

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

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

A MEMORY FOOTPRINT:

We use a similar simplified manifold mesh for calculating memory footprint as done in RXMesh [Mahmoud et al. 2021]. For such a model, RXMesh requires 37.4 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 1.994x less memory than RXMesh.