GPU-accelerated Certified Hausdorff Distance Between Triangle Meshes

HAOPENG FAN, Zhejiang University, China
MIN TANG*, Zhejiang University, Zhejiang Sci-Tech University, China
LEONARDO SACHT, Universidade Federal de Santa Catarina, Brazil
QIANG ZOU, State Key Lab of CAD and CG, Zhejiang University, China
RUOFENG TONG, Zhejiang University, China
PENG DU, State Key Lab of CAD and CG, Zhejiang University, China

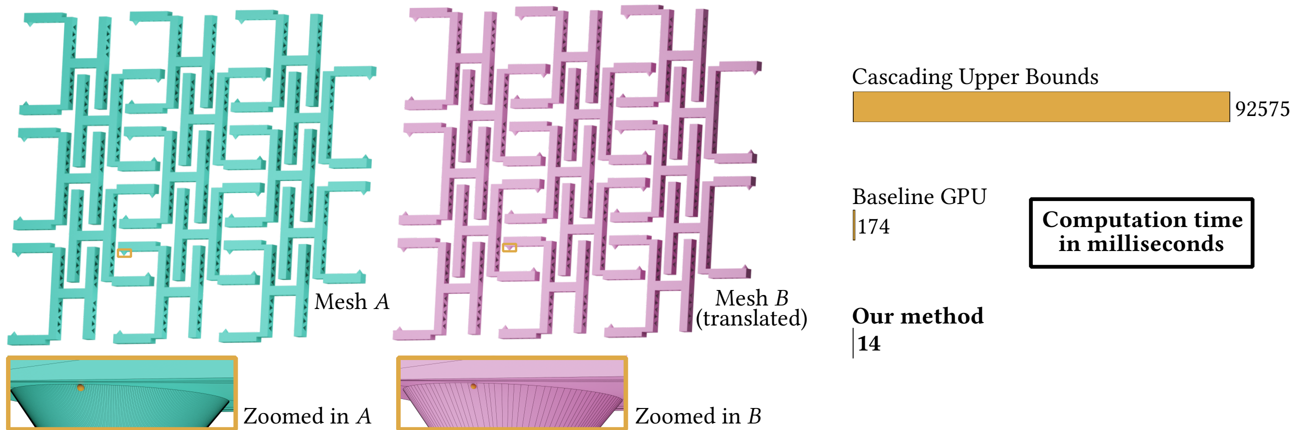

Fig. 1. Calculating the Hausdorff distance between similar objects is challenging. Zoomed-in views show the point on A that maximizes the distance to B and its closest point on B: the distance between these points is smaller than 0.0004% of the bounding-box diagonal. The state-of-the-art cascading-upper-bounds CPU baseline [Sacht and Jacobson 2024] takes 92,575 ms to reach the target tolerance; a Baseline GPU implementation (our direct parallel port of the cascading-upper-bounds pipeline) takes 174 ms; and our method takes 14 ms under the same settings.

Computing the directed Hausdorff distance between two triangle meshes is a fundamental operation in geometry processing and simulation. While existing certified branch-and-bound (B&B) methods are efficient for well-separated geometry, they can become prohibitively expensive on large models under tight tolerances and near-zero distance configurations where pruning is limited. We present a GPU-accelerated certified B&B algorithm that explicitly maintains enclosing lower and upper bounds on the directed Hausdorff distance and terminates once their normalized gap, measured with respect to the bounding-box diagonal of the source mesh, meets a user-prescribed tolerance. To map the inherently prioritized search to SIMT (single-instruction multiple-thread) hardware, we replace priority queues and recursion with a sorted, double-buffered wavefront pipeline built from bulk-parallel worklists for bound evaluation, culling, subdivision, and compaction. To mitigate loose bounds on thin primitives while preserving predictable stream behavior, we introduce a fixed-cardinality adaptive subdivision scheme that selectively applies double longest-edge bisection. To remain robust in deep-refinement regimes, we add a resource-aware deferral mechanism that enforces a device-capacity invariant by prioritizing candidates likely to be culled while postponing expensive ones. Finally, we improve numerical robustness under FP32 (single precision) via triangle-local coordinate transforms and other conservative numerical safeguards, and enhance coherence by spatially ordering the active set and traversing the BVH (bounding volume hierarchy) in triangle packets. Under the same stopping tolerance, experiments on an NVIDIA RTX 5090 show that our GPU solver remains numerically consistent with the FP64 CPU baseline, with normalized cross-platform deviation below 0.01% in over 99.9% of cases. Our method achieves millisecond-scale runtimes capable of supporting interactive frame rates, even on models with millions of triangles. Across the comparison set, it delivers throughput speedups of 836× on the Thingi10K/TetWild benchmark ($A \longrightarrow B$) and 709× on the Thingi10K/Decimation benchmark.

Code and data for this paper are available at https://github.com/fhp-transient/gpu-hausdorff.

CCS Concepts: • **Computing methodologies → Modeling and simulation**; **Massively parallel algorithms**.

Additional Key Words and Phrases: Hausdorff Distance, GPU Acceleration, Branch-and-Bound, Quality-Based Adaptive Subdivision

*Corresponding author, https://min-tang.github.io/home/GCHD/, This work was supported by “Pioneer” and " Leading Goose" R&D Program of Zhejiang Province under Grant No. 2025C01086.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Haopeng Fan, Min Tang, Leonardo Sacht, Qiang Zou, Ruofeng Tong, and Peng Du. 2018. GPU-accelerated Certified Hausdorff Distance Between Triangle Meshes. In . ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The Hausdorff distance (HD) captures the maximum geometric deviation between shapes, making it essential for applications demanding strict guarantees such as mesh simplification [Cignoni et al. 1998] and manufacturing tolerance verification [Binninger et al. 2021]. Unlike average-based measures, certified HD solvers can guarantee that the true maximum deviation is enclosed within explicit bounds. However, due to performance unpredictability in challenging configurations, certified HD has remained challenging to deploy in real-time pipelines, especially for near-zero configurations. We focus on the directed (one-sided) HD between triangle meshes, which forms the basis for the symmetric metric.

Mathematically, the directed HD is defined as the maximum over points on mesh A of the distance to the closest point on mesh B. Computing this quantity is challenging because the maximizer may lie anywhere on a continuous surface. While sampling-based approximations [Aspert et al. 2002; Cignoni et al. 1998] are fast, they generally fail to certify what was missed. For this reason, high-accuracy methods rely on branch-and-bound (B&B): maintaining a global interval $[\underline{h}, \overline{h}]$ such that $\underline{h} \leq h(A, B) \leq \overline{h}$, and iteratively refining surface subdomains until the gap meets a user-specified tolerance. In this paper, certified refers to interval-bounded, tolerance-terminated computation: throughout execution, the solver explicitly maintains a valid enclosing interval $[\underline{h}, \overline{h}]$ for the true directed Hausdorff distance and terminates only when the normalized bound gap $(\overline{h} - \underline{h})/d_A$ falls below the prescribed tolerance, where d_A denotes the diagonal length of the axis-aligned bounding box of the source mesh A. Certification therefore refers to the solver's internal bound guarantee, not to agreement between implementations. By contrast, when we compare our FP32 GPU solver against the FP64 CPU baseline, we report the difference between their final certified upper bounds as cross-platform deviation. CPU algorithms have steadily improved [Guthe et al. 2005; Tang et al. 2009], culminating in recent work that strengthens pruning via cascading upper bounds—progressively applying tighter (but more expensive) bounds to reject non-maximal regions [Sacht and Jacobson 2024].

Although CPU-based B&B solvers are highly effective for general inputs, the near-zero regime ($h(A, B) \approx 0$, Figure 1) remains a critical bottleneck. In these cases, local upper bounds barely exceed the global lower bound, rendering conservative pruning ineffective and forcing deep refinement. This exponential growth in active subdomains pushes runtimes from seconds to minutes on high-resolution meshes, preventing their use in real-time pipelines.

GPUs appear to be a natural fit for accelerating the massive geometric workload in such cases, yet existing B&B formulations map poorly onto SIMT hardware. State-of-the-art B&B relies on priority-queue-driven recursive refinement, producing irregular control flow, pointer-chasing dependencies, and frequent global synchronization. On GPUs, this translates into severe warp divergence and load imbalance. Moreover, in near-zero regimes, the active set can expand

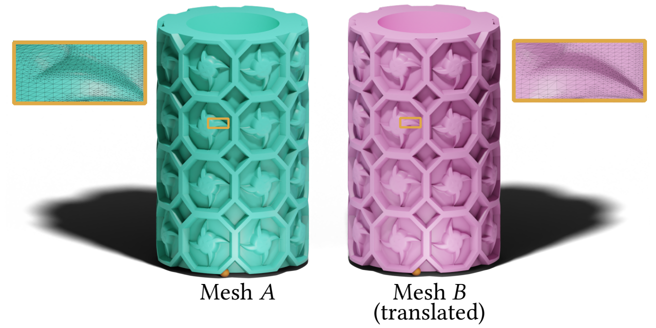

Fig. 2. Mesh A has 2.274 M faces and mesh B has 1.137 M. Our method terminates with a certified interval satisfying the stopping tolerance $\varepsilon = 10^{-6}$ for this near-zero Hausdorff distance case. The points on the bottom are the maximizer on A and its closest point on B; the close-up views illustrate the high resolution of both meshes.

rapidly, risking device memory exhaustion. Consequently, many GPU approaches typically revert to sampling-based approximations or restrict inputs to parametric surfaces [Krishnamurthy et al. 2011], sacrificing the explicit bound tracking on unstructured meshes that makes B&B attractive.

1.1 Main Results

This paper demonstrates that B&B-style bound tracking and high-throughput GPU execution are not incompatible. Our key observation is that the computational core of HD B&B is not the priority queue, but the massive evaluation of local bounds and refinement. We therefore replace serial, recursion-based logic with a sorted wavefront architecture: we process a spatially coherent active set in bulk using double-buffered worklists, fusing bound updates with culling and subdivision. To make this practical in the near-zero regime, we introduce (i) a numerically robust bounding strategy using triangle-local coordinate systems to preserve FP32 precision, and (ii) a priority-based deferral mechanism that prioritizes the pruning of converging primitives while deferring expensive candidates to mitigate device memory pressure.

Performance. We evaluated our method on an NVIDIA RTX 5090. For a fair comparison, we use the same stopping tolerance for the state-of-the-art CPU implementation [Sacht and Jacobson 2024] and apply both solvers to identical mesh pairs. On models with millions of triangles (see Figure 2), our method achieves millisecond-scale runtimes suitable for real-time scenarios, delivering throughput speedups (ratio of total time) of 836 × on the comparison set for the Thingi10K/TetWild benchmark ($A \longrightarrow B$), and 709 × for the Thingi10K/Decimation benchmark. Our GPU solver rigorously satisfies the same certification criterion as the CPU baseline, terminating only when the normalized bound gap $(\overline{h} - \underline{h})/d_A$ falls below the prescribed 10^{-6} tolerance. Separately, to assess numerical consistency across implementations, we compare the final certified upper bounds returned by our FP32 GPU solver and the FP64 CPU baseline. The normalized cross-platform deviation remains below 0.01% in over 99.9% of cases, with worst-case values of 1.10×10^{-3} (0.11%)

on the Thingi10K/TetWild benchmark and 1.40×10^{-2} (1.4%) on the Thingi10K/Decimation benchmark, while retaining explicit interval bounds and tolerance-based termination.

The principal contributions of our work include:

- **Sorted Wavefront B&B for Hausdorff Distance.** We reformulate cascading-upper-bound B&B as a recursion-free pipeline. By sorting the active set via a composite key (encoding priority, subdivision mode, and spatial location), we maximize execution coherence and minimize warp divergence without relying on global priority queues.
- **Fixed-Cardinality Adaptive Subdivision.** We decouple topological quality from memory management with a specialized subdivision scheme that guarantees a fixed $1 \rightarrow 4$ output cardinality while handling thin primitives (via double longest-edge bisection). This enables efficient, atomic-free stream compaction and predictable memory usage.
- **Resource-Aware Memory Regulation.** We introduce a priority-based deferral mechanism that dynamically enforces a device-capacity invariant. By prioritizing the refinement of primitives likely to be culled quickly, we mitigate the risk of memory exhaustion in deep-refinement regimes ($h(A, B) \approx 0$), enabling the processing of large-scale meshes where unregulated breadth-first search would fail.
- **Precision-Robust Geometric Evaluation.** We integrate numerical safeguards for FP32 geometric evaluation—most notably triangle-local coordinate transformations and conservative bound evaluation—to preserve the same certification semantics as the CPU baseline while remaining numerically close to its FP64 final certified outputs.

2 RELATED WORK

Hausdorff distance (HD) computation is a fundamental component in geometry processing and quality assessment. Early practical tools, such as Metro [Cignoni et al. 1998] and MESH [Aspert et al. 2002], rely on discrete sampling: they densely sample the surface of a source mesh and project samples onto a target mesh to estimate the maximum closest-point distance. For the directed HD on a continuous surface, the maximum over a finite sample set constitutes a lower bound on the true maximum, but it does not provide a certified bound gap or a tolerance-based stopping guarantee, and the estimate may vary with sampling density and distribution.

To obtain tolerance-tracked results on triangle meshes, subsequent work shifted toward B&B formulations that explicitly maintain global bounds $[\underline{h}, \overline{h}]$ and refine the search domain until the bound gap meets a user-specified tolerance. Guthe et al. [Guthe et al. 2005] introduced octree-based spatial indexing with error-guided adaptive sampling for mesh comparison. Tang et al. [Tang et al. 2009] formalized the B&B framework for mesh HD, leveraging BVH-based distance queries and hierarchy-driven refinement to achieve interactive performance.

Motivated by near-zero bottlenecks, later CPU methods focused on designing tighter yet economical upper bounds and on mitigating memory pressure. Kang et al. [Kang et al. 2018] avoid storing explicit primitive pairs in the queue and accelerate projection using auxiliary spatial structures (e.g., uniform grids), substantially reducing memory pressure in challenging cases. Zheng et al. [Zheng et al. 2022] propose an "economic" upper bound that uses a small set of representative samples and inexpensive point–triangle distance queries, aiming to preserve pruning effectiveness while lowering per-test cost.

The current CPU state of the art for unstructured triangle soups is the cascading upper bounds framework of Sacht and Jacobson [Sacht and Jacobson 2024]. To balance tightness and computational cost, it evaluates a sequence of increasingly expensive bounds $u_1, \ldots, u_4$, enabling early-out on easy regions while reserving more expensive geometric tests for difficult cases. This upgrade-on-demand design, however, naturally induces heterogeneous per-triangle workloads and deep recursion, which is a poor fit for SIMT execution in its original form and makes direct GPU ports prone to warp divergence and load imbalance.

On the GPU side, high efficiency has often been achieved by exploiting additional structure. For freeform surfaces (e.g., NURBS), the regularity of the parametric domain enables grid-like evaluation and more uniform control flow: prior work leverages parameter-domain discretizations and coherence to implement data-parallel pruning and traversal on GPUs [Hanniel et al. 2012; Krishnamurthy et al. 2011], while related CPU approaches [Elber and Grandine 2008; Kim et al. 2010, 2013] exploit parameterization and continuity to handle near-zero cases efficiently. These techniques rely on global parameter-domain structure and do not directly transfer to general triangle meshes, where traversal is driven by irregular BVHs and cascading upper bounds introduce further branching heterogeneity.

Recent accelerations also target discretized domains rather than continuous triangle meshes. RT-HDIST [Kim et al. 2025] leverages hardware ray-tracing cores to accelerate Hausdorff distance on point sets by reformulating the problem as nearest-neighbor search with voxel-space clustering. However, extending such methods to meshes typically requires surface sampling, reverting to an approximation without continuous, tolerance-tracked bound reasoning. In the volumetric setting, DisTorch [Rony and Kervadec 2025] accelerates distance-based metrics on voxel-grid boundaries for 3D segmentation evaluation, but it operates in label and voxel space rather than on continuous geometric bounds for triangle meshes.

Broader GPU-based proximity queries have also been explored: Lauterbach et al. [Lauterbach et al. 2010] demonstrated hierarchical collision and distance queries on the GPU, while Krishnamurthy et al. [Krishnamurthy et al. 2009] accelerated geometric queries for CAD models. For exact HD computation on point sets, Taha and Hanbury [Taha and Hanbury 2015] proposed efficient algorithms, and Zhang et al. [Zhang et al. 2017] addressed 3D point sets with octree-based methods. In the freeform domain, Alt and Scharf [Alt and Scharf 2008] studied HD between curved objects, and subsequent work [Bartoň et al. 2010; Chen et al. 2010; Son et al. 2021] refined precision and efficiency for various curve and surface representations.

In summary, CPU research has developed efficient bound-tracked B&B pipelines for triangle meshes, culminating in cascading upper bounds, whereas existing GPU work largely relies on parametric regularity or discretized representations (points and voxels). This leaves

a gap for unstructured triangle meshes: a GPU cascading-upper-bound B&B that preserves explicit bound tracking and tolerance-based termination while mitigating SIMT divergence and memory exhaustion. To our knowledge, we fill this gap by reformulating cascading-upper-bound B&B as a sorted wavefront pipeline with fixed-cardinality subdivision and resource-aware execution.

3 ALGORITHM PIPELINE

This section formalizes the directed (one-sided) Hausdorff distance problem and introduces our GPU-native sorted-wavefront architecture. State-of-the-art certified solvers on CPU [Sacht and Jacobson 2024] rely on a recursive Depth-First or Best-First Search B&B pipeline: they maintain a global lower bound $\underline{h}$, store active subdomains in a scalar priority queue ordered by local upper bounds, and iteratively pop, evaluate, cull, and subdivide candidates while updating $\underline{h}$. While this minimizes memory footprint, it maps poorly to GPUs due to pointer chasing, deep per-thread recursion stacks, and highly irregular control flow. In contrast, we reformulate B&B as a breadth-first, stream-processed pipeline built around double-buffered worklists. Our design maximizes throughput via bulk-parallel kernels while strictly enforcing a device-capacity invariant to keep the wavefront within device memory in the near-zero regime.

3.1 Problem Formulation

Given two triangle meshes $A = \{T_i^A\}$ and $B = \{T_j^B\}$, we denote by $d(p, B) = \min_{q \in B} \|p - q\|_2$ the Euclidean distance from a point p to mesh B. The directed Hausdorff distance is

$$h(A, B) = \max_{p \in A} d(p, B) = \max_{p \in A} \min_{q \in B} \|p - q\|_2. \quad (1)$$

Directly optimizing this objective over continuous surfaces is prohibitive; we therefore adopt a certified B&B strategy that maintains an enclosing interval $[\underline{h}, \overline{h}]$ such that $\underline{h} \le h(A, B) \le \overline{h}$.

We define the global lower bound $\underline{h}$ as the maximum distance from a discrete sample set on A (comprising initial vertices and subdivision samples) to B, providing a constructive witness. Let $\mathcal{W} \subset A$ be the current active worklist of triangles. For any triangle $T \in \mathcal{W}$, we define an admissible local upper bound $u(T)$ as a conservatively computed scalar that strictly upper-bounds the true maximum distance from T to B, such that $u(T) \ge \max_{p \in T} d(p, B)$. The global upper bound $\overline{h}$ is then defined as the maximum of these local bounds over $\mathcal{W}$: $\overline{h} = \max_{T \in \mathcal{W}} u(T)$.

Starting with $\mathcal{W}$ initialized to contain all triangles of A, we iteratively refine the worklist to tighten $[\underline{h}, \overline{h}]$ until the certified normalized gap

$$\frac{\overline{h} - \underline{h}}{d_A} \le \varepsilon \quad (2)$$

falls below a user tolerance ε, where d_A is the diagonal length of the axis-aligned bounding box of A. This normalized gap upper-bounds the absolute error, and we report this metric throughout the paper.

At each iteration, we compute admissible upper bounds $u(T)$ for active triangles and cull those with $u(T) < \underline{h}$. When the stopping criterion holds, the interval $[\underline{h}, \overline{h}]$ is certified, and $\overline{h}$ may be reported as a conservative final estimate.

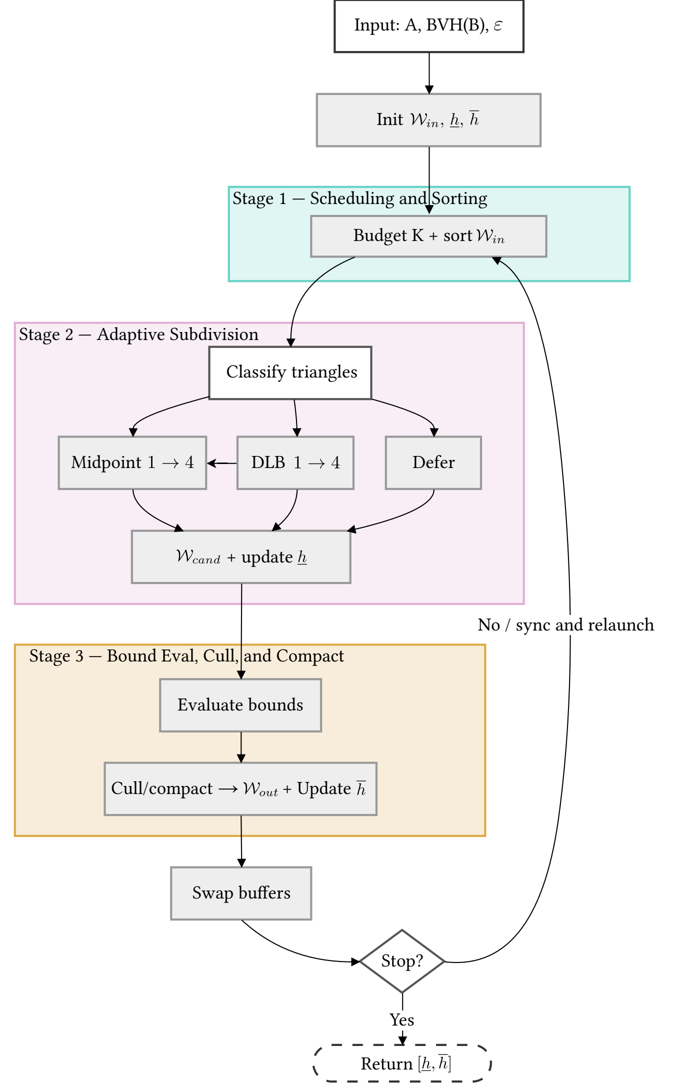

Fig. 3. Overview of our sorted-wavefront pipeline with double-buffered worklists.

3.2 Wavefront Architecture

B&B algorithms for Hausdorff distance are challenging to parallelize on GPUs due to irregular control flow and pointer-chasing dependencies. We restructure the search into a sorted wavefront pipeline using double-buffered Structure-of-Arrays (SoA) worklists, enabling coalesced access and bulk-parallel passes. Each wavefront iteration consists of three stages (Figure 3 and Algorithm 1).

Pipeline overview. Figure 3 summarizes the control flow of our GPU-native solver. The active set is maintained in double-buffered worklists and processed in three bulk-parallel stages: scheduling and sorting, adaptive subdivision or deferral, and bound evaluation with culling and compaction. Between iterations, the host reads back the updated global bounds and active-set counters, checks the

Algorithm 1 GPU-Accelerated Sorted-Wavefront Pipeline (Overview)

Require: Meshes A, B; tolerance ε; device capacity $M_{\max}$
Ensure: Certified interval $[\underline{h}, \overline{h}]$ with $(\overline{h} - \underline{h})/d_A \leq \varepsilon$
1: $\mathcal{W}_{in} \leftarrow$ Triangles(A); $\mathcal{W}_{out} \leftarrow \varnothing$
2: $\underline{h} \leftarrow$ InitialSampling(A, B); $\overline{h} \leftarrow +\infty$
3: **while** $(\overline{h} - \underline{h})/d_A > \varepsilon$ **and** $\mathcal{W}_{in} \neq \varnothing$ **do**
// Stage 1: Memory-Bounded Scheduling and Sorting
4: Calculate subdivision budget K based on capacity $M_{\max}$
5: $\tau \leftarrow$ SelectThreshold($\mathcal{W}_{in}, K$) ▷ Determine priority cutoff
6: **Sort** $\mathcal{W}_{in}$ by Key (Priority ‖ Mode ‖ Spatial)
// Stage 2: Adaptive Subdivision Streams (Blind Dispatch)
7: $\mathcal{W}_{cand} \leftarrow$ ParallelExpand($\mathcal{W}_{in}, \tau$)
▷ Apply subdivision rules or deferral; update $\underline{h}$
// Stage 3: Bound Evaluation, Culling, and Compaction
8: $\mathcal{W}_{out} \leftarrow$ ParallelFilter($\mathcal{W}_{cand}, \underline{h}$)
▷ Evaluate $u(T)$, cull, and update $\overline{h}$
9: **Swap**($\mathcal{W}_{in}, \mathcal{W}_{out}$); $\mathcal{W}_{out} \leftarrow \varnothing$
10: **end while**
11: **return** $\overline{h}$

stopping criterion, and relaunches the next wavefront pass if the certified gap is still above the prescribed tolerance.

BVH Preprocessing. Before execution, we construct a linear Bounding Volume Hierarchy (BVH) for mesh B on the CPU. The hierarchy is flattened into a pointer-less array layout and transferred to GPU memory.

Stage 1: Memory-Bounded Scheduling and Sorting. We compute a subdivision budget, select a threshold τ to classify triangles as priority and deferred, and sort by composite key for locality (Section 5).

Stage 2: Adaptive Subdivision Streams (Blind Dispatch). Stream-specialized kernels expand priority items (with midpoint subdivision or double longest-edge bisection) and pass through deferred ones, using branchless dispatch (launching kernels without CPU-side branching) to avoid host-device synchronization. This produces $\mathcal{W}_{cand}$ and updates $\underline{h}$ from newly generated samples; subdivision details are in Section 4.1.

Stage 3: Bound Evaluation, Culling, and Compaction. We evaluate tight admissible bounds $u(T)$ for candidates (Section 4.2), cull against $\underline{h}$, and compact survivors into $\mathcal{W}_{out}$. We update $\overline{h}$ in-kernel and use warp-aggregated writes for compaction (Section 5.2). Finally, we swap the double buffers ($\mathcal{W}_{in} \leftrightarrow \mathcal{W}_{out}$) to prepare the active wavefront for the subsequent iteration.

4 GPU-OPTIMIZED BOUND EVALUATION

We architect upper-bound cascade evaluation for high throughput GPU execution. This section details (i) a quality-aware subdivision strategy that maintains a fixed $1 \rightarrow 4$ output cardinality to ensure predictable memory usage (Section 4.1), and (ii) a divergence- and memory-aware scheduling of the bound cascade with early-exit pruning (Section 4.2).

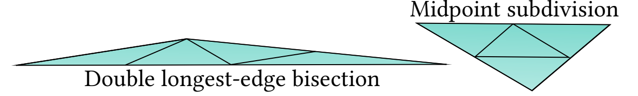

Fig. 4. Very thin triangles are subdivided using a double longest-edge bisection while better-shaped ones are subdivided using midpoint subdivision.

4.1 Quality-Based Adaptive Subdivision

Standard midpoint subdivision ($1 \rightarrow 4$, Figure 4-right) inherits the aspect ratio of parent triangles. For thin primitives (characterized by small area and large perimeter, Figure 4-left), upper-bound computations are dominated by the longest edge $l_{\max}$. Consequently, the bound tightness improves slowly under standard midpoint refinement, limiting culling effectiveness. To address this, we introduce Double Longest-Edge Bisection (DLB) for thin triangles.

We identify such primitives using a lightweight quality metric. Let T have vertices $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$. The quality is defined as the ratio of the triangle's altitude to its longest edge:

$$q(T) = \frac{\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\|}{l_{\max}^2} = \frac{2 \cdot \mathrm{Area}(T)}{l_{\max}^2} = \frac{h_\perp}{l_{\max}}, \quad (3)$$

where $h_\perp$ denotes the altitude perpendicular to the longest edge $l_{\max}$.

Triangles with $q(T) < 0.06$ are classified as thin and undergo specialized refinement. This empirical threshold targets extreme aspect ratios selectively, avoiding unnecessary computational overhead on well-shaped geometry, where standard midpoint subdivision efficiently reduces the bounding diameter while preserving the element's aspect ratio.

For thin triangles, we apply double longest-edge bisection: bisect along the longest edge, then bisect each child along its longest edge, yielding four grandchildren (Figure 4-left).

Crucially, this process generates exactly four output primitives, matching the cardinality of standard midpoint subdivision. This fixed cardinality enables predictable memory management with constant-stride output offsets, eliminating the need for per-parent atomic counters or variable-sized allocation logic.

4.2 Divergence-Aware Cascading Execution

We implement the four upper bounds (u_1–u_4) originally derived by Sacht and Jacobson [Sacht and Jacobson 2024]. For completeness and clarity, the detailed mathematical formulations of these bounds are provided in Appendix A. To map this logic efficiently onto SIMT hardware, we restructure the evaluation pipeline to (i) maximize warp-uniform instruction flow and (ii) reduce stalls caused by indirect global-memory accesses. We organize the bounds into two distinct phases: a high-throughput, predominantly uniform filtering phase (combining u_1, u_2, and a partial evaluation h_a) and a rigorous, divergence-heavy refinement phase (u_3, u_4).

Phase I: Warp-Uniform Filtering. We first evaluate (u_1, u_2) as a low-overhead arithmetic sieve with no additional global fetches. A primitive T is immediately culled if $\min(u_1(T), u_2(T)) < \underline{h}$. Survivors proceed to h_a, a warp-uniform partial evaluation of u_3 (Appendix A) that computes point-to-triangle distances without bisector construction. Again, candidates are culled if $h_a(T) < \underline{h}$. Although

h_a loads candidate geometry, the uniform instruction trace allows effective latency hiding before entering the divergent phase.

Phase II: Divergent Bisector-Based Refinement. Only primitives that survive Phase I enter the full bisector-based bounds (u_3, u_4). As detailed in Appendix A, these bounds involve complex geometric case classification (e.g., clipping against bisector planes and conditional fragment-to-triangle distance evaluation), which naturally incurs substantial control-flow divergence. By aggressively culling via the uniform Phase I, we ensure that only a small fraction of the active wavefront is exposed to this divergence penalty.

Sync-Free Early Exit. Throughout the cascaded evaluation, threads maintain the current best upper bound u_{curr} and test $u_{\text{curr}} < \underline{h}$ after each stage. If satisfied, the thread terminates its evaluation immediately. Crucially, this early-exit mechanism evaluates bounds independently per thread without requiring explicit warp- or block-level synchronization barriers. While SIMT execution dictates that early-exiting threads must wait for the longest-running thread in their warp, masked threads safely bypass the costly indirect memory fetches and complex arithmetic of subsequent bounds. Furthermore, because the active work buffer is spatially sorted via Morton keys, threads within the same warp process geometrically clustered primitives. This spatial coherence maximizes the probability that all threads in a warp satisfy the culling conditions simultaneously, allowing the entire warp to retire early and effectively mitigating the intra-warp divergence penalty. Surviving primitives are then compacted only once via warp-aggregated operations when writing to the output buffer.

5 PARALLEL EXECUTION ON GPUS

Unlike CPU-based methods that typically employ Depth-First Search (DFS) to minimize memory footprint, we use a level-synchronous wavefront (BFS-like) strategy to saturate massive parallelism, augmented with a capacity-aware priority policy (Section 5.3). We implement this via a double-buffered worklist mechanism, where active primitives are processed in parallel and remaining tasks are compacted into a contiguous output buffer for the subsequent iteration. While this level-wise traversal maximizes instruction throughput, it introduces challenges regarding memory capacity and warp divergence. We address these through the following architectural optimizations.

5.1 Coherent Traversal

To mitigate the latency of indirect memory accesses (e.g., fetching vertex coordinates via indices), we maintain geometric attributes in flat, coalesced buffers. This organization facilitates intra-thread packet traversal: instead of treating triangle vertices as independent query points, each thread treats the three vertices $T = \{v_0, v_1, v_2\}$ of its assigned active triangle as a local packet. We manage this packet using a private, thread-local stack and a register-based bitmask to track valid samples. This setup allows for joint node testing—checking the BVH bounding box against all three vertices simultaneously—and skipping branches that cannot improve the distance for any active vertex.

Crucially, while the traversal stack is private to each thread, spatial sorting (Section 5.2) ensures adjacent warp threads process spatially proximal triangles. Combined with the SoA layout, this ensures threads fetch adjacent BVH nodes and geometry data, maximizing L2 cache hit rates and coherence. Algorithm 2 in Appendix B provides pseudocode for this traversal, including the near-first heuristic used to accelerate culling.

Numerical Robustness. To mitigate catastrophic cancellation errors when processing meshes with large world-space coordinates, all geometric predicates and bound computations (u_1, u_2, u_3, u_4) are performed in a triangle-local coordinate system. We translate the origin to the parent triangle's first vertex (v_0) on-the-fly during data fetching. This translation is an isometry and strictly preserves distances, ensuring that the computed bounds remain valid and conservative while maximizing floating-point precision.

5.2 Spatial Sorting and Stream Compaction

Morton Code Sorting. To mitigate the divergence penalty described in Section 4, the active buffer is spatially sorted at the beginning of each iteration. To enable a single-pass sort that respects both memory priority and spatial locality, we construct a 32-bit composite sort key Key for each primitive:

$$\text{Key} = (b_{\text{defer}} \ll 31) \vee (b_{\text{mode}} \ll 30) \vee (M_{30} \wedge \text{0x3FFFFFFF}), \quad (4)$$

where b_{defer} is the priority flag (1 if deferred), b_{mode} indicates the subdivision routine (Midpoint vs. Thin and DLB), and M_{30} is the 30-bit quantization of the Morton code derived from the triangle centroid. This sorting ensures that warps process spatially adjacent primitives with identical subdivision logic, significantly improving L2 cache hit rates and reducing branch divergence.

Warp-Aggregated Compaction and Fused Updates. Efficiently packing surviving triangles into the output buffer is critical. Leveraging the fixed $1 \rightarrow 4$ output cardinality (Section 4.1), we employ a warp-level aggregation strategy to minimize atomic contention: active threads identify survivors via ballot intrinsics; lane 0 reserves a contiguous segment with one global atomic; survivors then scatter to their allocated positions.

Simultaneously, global bounds ($\underline{h}$ and $\overline{h}$) are updated via fused reductions within the same kernels. By utilizing warp-level shuffle intrinsics before issuing global atomics, we strictly synchronize the global search window with wavefront evolution, avoiding the overhead of separate reduction passes.

5.3 Priority-Based Memory Regulation

A critical limitation of BFS on GPUs is the risk of unbounded wavefront growth, particularly in near-zero regimes where pruning is initially ineffective. To strictly adhere to the device memory budget $M_{\max}$, we implement a priority-based deferral mechanism that dynamically enforces a capacity invariant.

At the start of each iteration, we calculate a subdivision budget K based on remaining memory. We then determine a priority cutoff τ by performing a parallel radix sort on the upper bounds of the active set and selecting the element at rank K. During the subdivision stage, primitives with tight bounds $(u(T) \leq \tau)$ are refined immediately, as they are likely to be culled quickly. Conversely, expensive

candidates with loose bounds ($u(T) > \tau$) are flagged as deferred: they bypass subdivision and are copied directly to the output buffer via stream compaction. Empirically, the deferral rate dynamically adapts to memory pressure: it remains close to zero when primitives are quickly culled, but scales to the vast majority of the active set whenever the wavefront hits the capacity ceiling.

This strategy effectively morphs the traversal from pure BFS to a resource-constrained best-first search under pressure, preventing allocation failures while guaranteeing forward progress.

6 RESULTS

The careful design and implementation of our method enables it to excel when $h(A, B) \approx 0$ and/or both meshes have large triangle counts. Figure 1 shows an example where B is the result of decimating A by a 0.5 factor. The meshes substantially overlap but are rendered separately for clarity, as is done for all results in this paper. Our method takes only 14 ms using the tolerance $\varepsilon = 10^{-6}$. Figure 2 presents another near-zero case, but now both meshes have millions of triangles. Our method takes 25.5 ms for $\varepsilon = 10^{-6}$. The remainder of this section presents a detailed performance analysis on benchmarks derived from the Thingi10K dataset.

6.1 Experimental Setup

Hardware and software. We evaluate scaling across three NVIDIA consumer GPUs: RTX 3060 (8GB VRAM), RTX 4070 (8GB VRAM), and RTX 5090 (32GB VRAM). CPU-SOTA is executed on an Intel Core i7-14700HX (2.10 GHz base) with 32GB system RAM. All experiments run on a Windows 11 Pro 64-bit workstation. Our GPU solver is implemented in CUDA (Toolkit 13.0) and uses single-precision (FP32) arithmetic; we build BVHs using AABB-based hierarchies as a preprocessing step. Unless otherwise stated, all ablation experiments are conducted on the GeForce RTX 4070 (8GB).

Benchmarks. We perform a large-scale evaluation on a combined suite of 19,807 mesh pairs, mirroring the datasets used by the state-of-the-art cascading-upper-bounds CPU baseline [Sacht and Jacobson 2024]. Specifically, the Thingi10K/TetWild benchmark contains **9,836** mesh pairs ($A \to B$, Figure 5), pairing each Thingi10K surface [Zhou and Jacobson 2016] with the boundary extracted from its TetWild tetrahedralization [Hu et al. 2018]. The Thingi10K/Decimation benchmark contains **9,971** pairs ($A \to A_{0.5}$, Figures 1 and 2), pairing each mesh with its 50% decimated counterpart and stressing near-zero-distance configurations that trigger deep refinement. Two additional results from these benchmarks are provided in Appendix C.

Baselines and protocol. We compare against (1) CPU-SOTA, the official C++ implementation of the cascading-upper-bounds CPU method [Sacht and Jacobson 2024], and (2) a Baseline GPU port that runs the same bound cascade in parallel but without our sorting, packet traversal, adaptive subdivision, or memory-aware deferral. The main performance comparisons in this paper report only the iterative B&B loop (Algorithm 1) in order to isolate the throughput of the certified solver itself under a consistent measurement scope across methods. At the same time, end-to-end cost is relevant in practice, so we additionally report a separate timing breakdown of our implementation in Table 3. This breakdown separates CPU-side loading and parsing, CPU preprocessing (BVH construction), and GPU initialization and transfer; these one-time costs are reported for transparency but are not included in the main solver-time speedup tables. We sanitize inputs by removing unreferenced vertices to avoid spurious lower-bound inflation. CPU runs use a 180 s per-pair timeout. For GPU runs, we enforce a 60 s budget; cases that do not meet the stopping criterion $(\overline{h} - \underline{h})/d_A \leq 10^{-6}$ within this limit are marked as timeouts, while those exceeding device memory capacity are classified as Out-of-memory (OOM) failures. Unless otherwise stated, all statistics are computed on the intersection set of instances where the CPU baseline succeeds within 180 s and all tested GPU variants successfully converge without timeout or OOM.

Table 1. GPU vs. CPU on Thingi10K→TetWild ($A \to B$, 9,766 pairs). CPU Median Time: 2134.52 ms. Mean-Ratio× is computed as $\sum t_{\mathrm{CPU}} / \sum t_{\mathrm{GPU}}$.

GPU	Median (ms)	Mean-Ratio×	Geo-Mean×	p90×	p99×
RTX 3060	7.82	527.91	174.73	911.67	2279.13
RTX 4070	5.89	750.24	235.37	1375.91	3698.36
RTX 5090	5.82	835.63	244.98	1638.01	5931.00

Correctness. We distinguish between certification and cross-platform deviation. Certification is an internal property provided by our solver: throughout execution, we maintain a valid interval $[\underline{h}, \overline{h}]$ enclosing the true directed Hausdorff distance, and we terminate only when the normalized bound gap satisfies $(\overline{h} - \underline{h})/d_A \leq 10^{-6}$. By contrast, cross-platform deviation is an external comparison between implementations, measured here as $|\overline{h}_{\mathrm{GPU}} - \overline{h}_{\mathrm{CPU}}|/d_A$ between the final certified upper bounds returned by our FP32 GPU solver and the FP64 CPU baseline. Across the comparison set, the mean normalized deviation is 2.54e-7 on TetWild and 1.73e-7 on Decimation; the p99 deviation is 9.78e-7 and 9.60e-7, respectively. The worst-case normalized deviations are 1.10e-3 (0.11%) on TetWild and 1.40e-2 (1.4%) on Decimation, each caused by a single pathological outlier. Crucially, for more than 99.9% of cases, the cross-platform deviation remains below 0.01%, confirming that our robust FP32 implementation remains numerically close to the FP64 CPU reference while still meeting the prescribed certification criterion.

Sanity-check with an analytically known distance. We additionally consider a simple translated-patch example with a closed-form directed Hausdorff distance. Let A be a square patch on the plane $z = 0$, triangulated into two triangles, and let $B = A + (0, 0, \delta)$ with $\delta > 0$. Then for every point $p = (x, y, 0) \in A$, the point $(x, y, \delta) \in B$ is its closest point on B, so $d(p, B) = \delta$ for all $p \in A$ and hence $h(A, B) = \delta$. Both our FP32 GPU solver and the FP64 CPU baseline maintain certified intervals containing the exact value δ and terminate once $(\overline{h} - \underline{h})/d_A \leq 10^{-6}$. For general mesh pairs, a closed-form reference value is typically unavailable, so this example is included only as a transparent sanity-check rather than as the basis of certification.

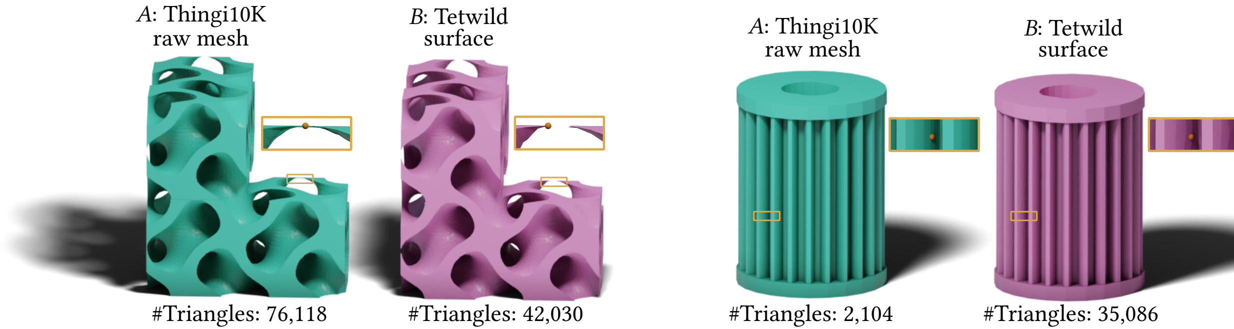

Fig. 5. Two pairs of meshes from the Thingi10K and TetWild benchmark. Close-ups show the maximizers of the distance on A and their closest points on B. Pairs in this benchmark have near-zero Hausdorff distance.

Table 2. GPU vs. CPU on Thingi10K→Decimation ($A \to A_{0.5}$, 9, 660 pairs). CPU Median Time: 229.46 ms. Mean-Ratio× is computed as $\sum t_{\text{CPU}} / \sum t_{\text{GPU}}$.

GPU	Median (ms)	Mean-Ratio×	Geo-Mean×	p90×	p99×
RTX 3060	4.57	224.19	39.43	395.59	1570.56
RTX 4070	3.76	306.74	48.70	577.22	2567.74
RTX 5090	4.44	709.30	47.13	827.63	4369.79

Table 3. Median end-to-end timing breakdown of our implementation on the comparison set. All times are in milliseconds. Setup = Load + CPU pre (BVH) + GPU init.; Total = Setup + Solve.

Bench.	N	Load (ms)	CPU pre (ms)	GPU init. (ms)	Setup (ms)	Solve (ms)	Total (ms)
TetWild	9,766	21.89	2.71	137.34	162.55	6.21	168.76
Decim.	9,660	10.21	0.77	135.13	145.98	4.04	150.02

6.2 End-to-End Performance

Tables 1 and 2 summarize end-to-end performance on the comparison set. We report Median runtime (ms) as a robust "typical-case" indicator, together with Geometric Mean and tail speedups (p90 and p99) to characterize variability. To quantify aggregate throughput, we report a throughput-oriented mean speedup with respect to the CPU-method (Mean-Ratio×) defined as the ratio of total runtimes:

$$\text{Mean-Ratio} = \frac{\sum_i t_{\text{CPU}}^{(i)}}{\sum_i t_{\text{GPU}}^{(i)}}.$$

Unlike averaging per-instance speedups, this metric directly reflects end-to-end throughput improvements on the benchmark suite.

Thingi10K/TetWild (Table 1). On this comparison set, CPU-SOTA has a median runtime of 2134.52 ms. On RTX 5090, our method achieves a median runtime of 5.82 ms, corresponding to a median speedup of 382.92× and a throughput speedup of 835.63×. These results show that the sorted wavefront pipeline can efficiently map certified B&B to SIMT execution despite irregular BVH traversal and cascading upper bound logic.

Thingi10K/Decimation (Table 2). Although CPU-SOTA is faster on median (229.46 ms) due to the overhead of GPU kernel launches on trivial cases, near-zero distances force deep refinement and stress branching and memory behavior. Our method maintains high throughput, reaching 709.30× on RTX 5090, while maintaining certification and substantially improving tail latency on pathological instances.

6.3 Ablation Study

We ablate key components of our GPU pipeline to quantify their individual contributions. Unless otherwise noted, we report a throughput-oriented slowdown relative to the full system (Ours), computed as $\sum_i t_{\text{variant}}^{(i)} / \sum_i t_{\text{ours}}^{(i)}$ over the intersection set of instances where both runs successfully converge within the time budget and do not OOM. All ablations in this section are measured on a single GeForce RTX 4070 (8GB) GPU.

Throughput-critical Components: Spatial Sorting and Packet Traversal. We first ablate two throughput-critical design choices: (i) spatial sorting of the active work buffer (Section 5.2), and (ii) packet BVH traversal that evaluates the three triangle vertices as a bundled query (Section 5.1).

Impact of Spatial Sorting. As shown in Figure 6, removing spatial sorting ("w/o sorting") causes a modest 1.09× slowdown on TetWild but a substantial 1.55× slowdown on Decimation. Decimation requires deeper refinement in near-zero configurations, producing a fragmented wavefront. Without Morton-code sorting, neighboring GPU threads process spatially distant triangles, severely reducing memory locality and throughput.

Impact of Packet Traversal. Disabling packet traversal ("w/o packet") and treating vertices as independent queries consistently degrades performance, yielding a 1.46× slowdown on TetWild and 1.49× on Decimation. Packet traversal reduces redundant BVH node visits and arithmetic operations, providing consistent benefits across all datasets.

Subdivision Strategy: Midpoint-only vs. Mixed. We additionally ablate our mixed subdivision strategy (Section 4.1) by comparing it against a midpoint-only rule. As shown in Figure 7, the mixed strategy incurs negligible overhead on average, with mean throughput being statistically indistinguishable from the simpler midpoint

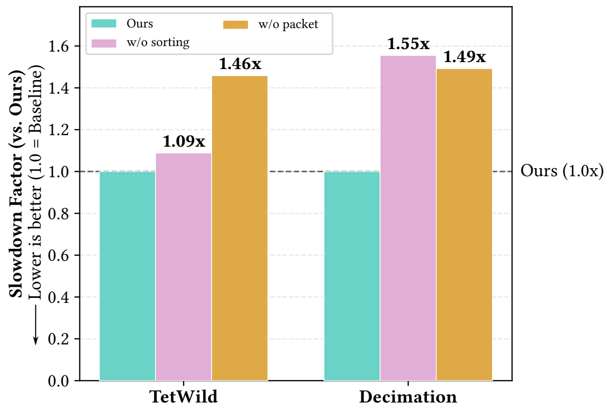

Fig. 6. Slowdown relative to the full pipeline (Ours) for variants without spatial sorting and without packet traversal. Sorting is particularly important for handling irregular refinement (Decimation), while packet traversal provides consistent architectural benefits across workloads.

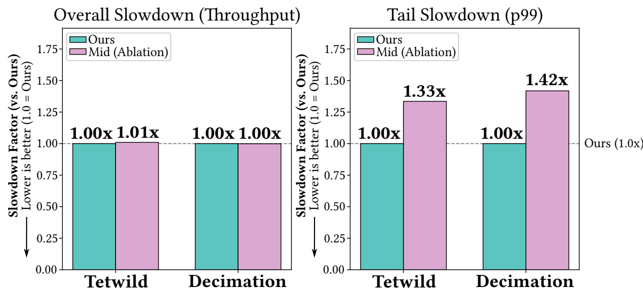

Fig. 7. Throughput-oriented slowdown (left) and tail slowdown at p99 (right) relative to the full system (mixed subdivision, Ours). Midpoint-only matches mixed subdivision on average throughput but is substantially worse in the tail, indicating that thin-triangle refinement mainly improves robustness on the hardest near-zero instances.

approach. However, its impact is pronounced in the tail latency. The thin-triangle refinement acts as a critical auxiliary mechanism for pathological geometry—specifically, cases where a single long edge spans multiple target primitives. In these scenarios, standard midpoint subdivision converges slowly, whereas our mixed strategy rapidly tightens bounds to resolve the bottleneck without inducing additional memory pressure.

Subdivision Priority and Memory Regulation. We investigate memory stability on the 8GB RTX 4070. To maximize the number of resident triangles, our worklist stores only minimal state (vertex positions and closest-triangle indices), recomputing intermediate data on demand. Even with this compact representation, unchecked growth leads to memory exhaustion. Disabling deferral (unbounded BFS) results in 110 and 203 OOM failures on the full TetWild and Decimation benchmarks, respectively, confirming that regulating wavefront size is mandatory for consumer hardware. Furthermore, policy choice is critical under this constraint: compared to a largest-u-first approach, our smallest-u-first priority reduces OOMs from

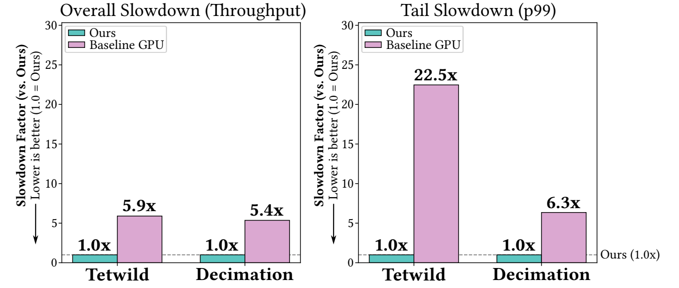

Fig. 8. Slowdown of the Baseline GPU implementation relative to our method. Our approach is 5.9× faster on average for TetWild and 5.4× faster for Decimation, with the gap widening significantly in the tail (e.g., reaching a 22.5× slowdown for the Baseline at p99 on TetWild).

Table 4. **Top 5 Speedups vs. Baseline GPU (TetWild Benchmark).** Our architectural optimizations deliver up to 140× speedups on pathological instances where the Baseline approach suffers from poor locality and divergence.

Model ID	Ours (ms)	Baseline (ms)	Speedup
1087144	45.64	6,418.41	**140.64×**
47850	8.78	1,213.43	**138.14×**
1088054	39.94	5,493.23	**137.55×**
1088217	39.92	5,229.30	**130.98×**
1088218	38.42	4,899.78	**127.54×**

54 to 4 on the Decimation benchmark. This validates that our strategy successfully maximizes the effective reach of limited VRAM, enabling the certification of deep-refinement cases that would otherwise fail due to capacity exhaustion.

6.4 Impact of Architectural Optimizations

We quantify the impact of our architectural innovations (sorted execution, adaptive subdivision, and double-buffered scheduling) by comparing our full pipeline with the Baseline GPU implementation described in Section 6.1. As shown in Figure 8, our method delivers substantial throughput gains, with mean speedups of 5.9× on TetWild and 5.4× on Decimation. Table 4 further reports peak performance on pathological geometry, where our optimizations mitigate memory divergence and achieve speedups above 140×.

7 CONCLUSION

We presented a GPU-accelerated, tolerance-certified solver for the directed Hausdorff distance between triangle meshes. Our central idea is to replace the irregular priority-queue recursion of CPU B&B pipelines with a sorted wavefront that processes the active set in bulk using double-buffered worklists. Combined with fixed-cardinality adaptive subdivision, packetized BVH traversal, and a resource-aware deferral strategy under memory pressure, this reformulation preserves explicit global lower and upper bounds and tolerance-based termination while mapping naturally to SIMT execution.

On the large-scale Thingi10K/TetWild and Thingi10K/Decimation benchmarks, our method satisfies the same certification criterion as the state-of-the-art cascading-upper-bounds CPU solver under the same stopping tolerance and achieves millisecond-scale runtimes on modern GPUs. In addition, the final certified upper bounds returned by our FP32 GPU implementation remain numerically close to those of the FP64 CPU baseline, while delivering substantial throughput gains of up to 835.63× on the Thingi10K/TetWild benchmark and 709.30× on the Thingi10K/Decimation benchmark on RTX 5090.

Limitations. First, our current iterative scheduler relies on host-side convergence checks between kernel launches. This introduces intermittent synchronization overheads ("stop-and-go"), preventing the GPU from being fully saturated by a continuous compute stream. Second, to maximize throughput, our wavefront explicitly stores full vertex coordinates (9 floats per triangle). While effective, this limits the maximum solvable problem size on commodity GPUs; a more compact representation (e.g., storing only parent indices and subdivision codes) could reduce the memory footprint by an order of magnitude but remains an engineering challenge. Third, while our triangle-local coordinate system mitigates floating-point cancellation, extreme aspect ratios or near-degenerate inputs may still require mixed-precision (FP64) arithmetic to guarantee strict certification, which implies a trade-off between throughput and numerical robustness.

Future work. With performance no longer the primary bottleneck, promising directions include (i) fully device-resident control flow (e.g., CUDA Graphs) to reduce CPU–GPU synchronization overhead, (ii) implicit wavefront encodings (paths or codes instead of explicit geometry) to scale to larger meshes, (iii) faster symmetric Hausdorff computation by warm-starting the reverse direction with bounds from the forward pass, and (iv) extensions to dynamic or deformable meshes, where incremental BVH refitting or lazy reconstruction strategies could amortize hierarchy update costs across frames.

Ultimately, by bringing certified distance evaluation into the millisecond regime, our method opens the door for using the Hausdorff distance in latency-sensitive geometry-processing scenarios where uncertified approximations were previously the only viable option. Examples include interactive mesh optimization and tolerance-verification workflows, where fast and reliable geometric deviation bounds can be valuable under tight computational budgets.

REFERENCES

Helmut Alt and Ludmila Scharf. 2008. Computing the Hausdorff distance between curved objects. *International Journal of Computational Geometry & Applications* 18, 04 (2008), 307–320.

Nicolas Aspert, Diego Santa-Cruz, and Touradj Ebrahimi. 2002. Mesh: Measuring errors between surfaces using the Hausdorff distance. In *Proceedings. IEEE international conference on multimedia and expo*, Vol. 1. IEEE, 705–708.

Michael Bartoň, Iddo Hanniel, Gershon Elber, and Myung-Soo Kim. 2010. Precise Hausdorff distance computation between polygonal meshes. *Computer Aided Geometric Design* 27, 8 (2010), 580–591.

Alexandre Binninger, Floor Verhoeven, Philipp Herholz, and Olga Sorkine-Hornung. 2021. Developable approximation via Gauss image thinning. *Computer Graphics Forum (proceedings of SGP 2021)* 40, 5 (2021), 289–300.

Xiao-Diao Chen, Weiyin Ma, Gang Xu, and Jean-Claude Paul. 2010. Computing the Hausdorff distance between two B-spline curves. *Computer-Aided Design* 42, 12 (2010), 1197–1206.

Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. 1998. Metro: measuring error on simplified surfaces. In *Computer graphics forum*, Vol. 17. Wiley Online Library, 167–174.

Gershon Elber and Tom Grandine. 2008. Hausdorff and minimal distances between parametric freeforms in and. In *International conference on geometric modeling and processing*. Springer, 191–204.

Michael Guthe, Pavel Borodin, and Reinhard Klein. 2005. Fast and accurate Hausdorff distance calculation between meshes. In *Proc. WSCG*. 41–48.

Iddo Hanniel, Adarsh Krishnamurthy, and Sara McMains. 2012. Computing the Hausdorff distance between NURBS surfaces using numerical iteration on the GPU. *Graphical Models* 74, 4 (2012), 255–264.

Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. 2018. Tetrahedral meshing in the wild. *ACM Trans. Graph.* 37, 4 (2018), 60.

Yunku Kang, Min-Ho Kyung, Seung-Hyun Yoon, and Myung-Soo Kim. 2018. Fast and robust Hausdorff distance computation from triangle mesh to quad mesh in near-zero cases. *Computer Aided Geometric Design* 62 (2018), 91–103.

Yong-Joon Kim, Young-Taek Oh, Seung-Hyun Yoon, Myung-Soo Kim, and Gershon Elber. 2010. Precise Hausdorff distance computation for planar freeform curves using biarcs and depth buffer. *The Visual Computer* 26, 6 (2010), 1007–1016.

Yong-Joon Kim, Young-Taek Oh, Seung-Hyun Yoon, Myung-Soo Kim, and Gershon Elber. 2013. Efficient Hausdorff distance computation for freeform geometric models in close proximity. *Computer-Aided Design* 45, 2 (2013), 270–276.

Young Woo Kim, Jaehong Lee, and Duksu Kim. 2025. RT-HDIST: Ray-Tracing Core-based Hausdorff Distance Computation. In *Computer Graphics Forum*, Vol. 44. Wiley Online Library, e70229.

Adarsh Krishnamurthy, Sara McMains, and Kirk Halle. 2009. Accelerating geometric queries using the GPU. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. 199–210.

Adarsh Krishnamurthy, Sara McMains, and Iddo Hanniel. 2011. GPU-accelerated Hausdorff distance computation between dynamic deformable NURBS surfaces. *Computer-Aided Design* 43, 11 (2011), 1370–1379.

Christian Lauterbach, Qi Mo, and Dinesh Manocha. 2010. gProximity: hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 419–428.

Jérôme Rony and Hoel Kervadec. 2025. DisTorch: A fast GPU implementation of 3D Hausdorff distance. In *Medical Imaging with Deep Learning-Short Papers*.

Leonardo Sacht and Alec Jacobson. 2024. Cascading upper bounds for triangle soup Pompeiu-Hausdorff distance. In *Computer Graphics Forum*, Vol. 43. Wiley Online Library, e15129.

Sang-Hyun Son, Myung-Soo Kim, and Gershon Elber. 2021. Precise Hausdorff distance computation for freeform surfaces based on computations with osculating toroidal patches. *Computer Aided Geometric Design* 86 (2021), 101967.

Abdel Aziz Taha and Allan Hanbury. 2015. An efficient algorithm for calculating the exact Hausdorff distance. *IEEE transactions on pattern analysis and machine intelligence* 37, 11 (2015), 2153–2163.

Min Tang, Minkyoung Lee, and Young J Kim. 2009. Interactive Hausdorff distance computation for general polygonal models. *ACM Transactions on Graphics (TOG)* 28, 3 (2009), 1–9.

Dejun Zhang, Fazhi He, Soonhung Han, Lu Zou, Yiqi Wu, and Yilin Chen. 2017. An efficient approach to directly compute the exact Hausdorff distance for 3D point sets. *Integrated Computer-Aided Engineering* 24, 3 (2017), 261–277.

Yicun Zheng, Haoran Sun, Xinguo Liu, Hujun Bao, and Jin Huang. 2022. Economic upper bound estimation in Hausdorff distance computation for triangle meshes. In *Computer Graphics Forum*, Vol. 41. Wiley Online Library, 46–56.

Qingnan Zhou and Alec Jacobson. 2016. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797* (2016).

A SUMMARY OF CASCADING UPPER BOUNDS

We summarize the four upper bounds (u_1, u_2, u_3, u_4) for the directed Hausdorff distance from a query triangle $T \subset A$ to mesh B, based on the cascading framework of Sacht and Jacobson [Sacht and Jacobson 2024]. While we provide the core geometric intuition and exact mathematical formulations below to keep our document self-contained, we refer interested readers to the original work [Sacht and Jacobson 2024] for the corresponding visual diagrams and comprehensive proofs.

Notation. Let T have vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ with edge lengths $e_i = \|\mathbf{v}_{i+1} - \mathbf{v}_{i+2}\|$ (indices mod 3). For each vertex $\mathbf{v}_i$, let $\mathbf{q}_i \in B$ denote its closest point on B, such that $d(\mathbf{v}_i, B) = \|\mathbf{v}_i - \mathbf{q}_i\|$. Let $S_i \subset B$ be the primitive triangle containing $\mathbf{q}_i$.

Exact Case. If all three projections $\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3$ lie on the same primitive triangle $S \subset B$, the distance function is convex over T, and the exact Hausdorff distance is simply the maximum vertex distance:

$$h(T, B) = \max_{i \in \{1,2,3\}} \|\mathbf{v}_i - \mathbf{q}_i\|. \tag{5}$$

Bound 1: Anchor Bound (u_1). This bound exploits the 1-Lipschitz continuity of the distance function. For any point $\mathbf{p} \in T$, we have $d(\mathbf{p}, B) \le d(\mathbf{v}_i, B) + \|\mathbf{p} - \mathbf{v}_i\|$. Taking the minimum over all possible anchor vertices $\mathbf{v}_i$:

$$u_1(T) = \min_{i=1}^{3} \left(\|\mathbf{v}_i - \mathbf{q}_i\| + \max_{j \ne i} \|\mathbf{v}_i - \mathbf{v}_j\| \right). \tag{6}$$

Bound 2: Smallest Enclosing Ball (u_2). This bound uses the radius g of the smallest enclosing ball of T. For acute triangles, this ball is the circumcircle; for obtuse triangles, it is the circle having the longest edge as its diameter:

$$u_2(T) = g + \max_{i=1}^{3} \|\mathbf{v}_i - \mathbf{q}_i\|, \tag{7}$$

where

$$g = \begin{cases} R = \dfrac{e_1 e_2 e_3}{4|T|} & \text{if } T \text{ is acute,} \\ \dfrac{1}{2} \max_i e_i & \text{otherwise.} \end{cases} \tag{8}$$

Here $|T|$ denotes the area of T.

Bound 3: Piecewise-Linear Bisector (u_3). Following Kang et al. [Kang et al. 2018], this bound partitions T based on the Voronoi regions induced by the planes that support the candidate triangles $\{S_i\}$.

- **Case 1** (S_1, S_2 share an edge): The bisector plane between the supporting planes of S_1 and S_2 clips T into a triangle R and a quadrilateral Q:

$$u_3(T) = \max\left\{ h(R, S_1),\ h(Q, S_2) \right\}. \tag{9}$$

- **Case 2** (Otherwise): T is subdivided into three quadrilaterals Q_1, Q_2, Q_3 using edge midpoints and the centroid. The bound combines these partitioned estimates with direct triangle-to-triangle distances:

$$u_3(T) = \min\left\{ \max_{k=1}^{3} h(Q_k, S_k),\ \min_{k=1}^{3} h(T, S_k) \right\}. \tag{10}$$

Partial Bound (h_a). In our GPU implementation (Section 4.2), we define h_a as a warp-uniform relaxation of u_3 that bypasses the bisector construction and clipping:

$$h_a(T) = \min_{i=1}^{3} h(T, S_i) = \min_{i=1}^{3} \max_{j \in \{1,2,3\}} d(\mathbf{v}_j, S_i). \tag{11}$$

This bound tests whether any single candidate triangle S_i is spatially close to the entirety of T, enabling aggressive pruning without control-flow divergence.

Bound 4: Point-Point Bisector (u_4). For thin triangles where vertex projections $\{\mathbf{q}_i\}$ span disjoint regions of B, bounds u_1–u_3 may become loose. Bound u_4 instead considers the Voronoi partition induced by the projection points themselves.

Without loss of generality, assume the shortest edge of T connects $\mathbf{v}_2$ and $\mathbf{v}_3$, with $\mathbf{v}_1$ as the opposite vertex. We define:

$$u_4(T) = \min\left\{ h(T, \{\mathbf{q}_1, \mathbf{q}_2\}),\ h(T, \{\mathbf{q}_1, \mathbf{q}_3\}) \right\}, \tag{12}$$

where $h(T, \{\mathbf{q}_a, \mathbf{q}_b\})$ denotes the maximum distance from any point in T to the nearer of $\mathbf{q}_a$ and $\mathbf{q}_b$. This maximum is attained either at a vertex of T or at the intersection of an edge of T with the bisector plane of $\mathbf{q}_a$ and $\mathbf{q}_b$.

B INTRA-THREAD PACKET TRAVERSAL PSEUDOCODE

This appendix lists the pseudocode (Algorithm 2) for the intra-thread triangle packet BVH traversal used in Section 5.1. Each thread processes the three vertices of one active triangle as a small packet, maintaining a local stack of BVH nodes together with a 3-bit mask indicating which vertices may still be improved. The near-first ordering prioritizes the child with smaller box distance, while the per-vertex masks allow pruning branches that cannot decrease the current minima. The routine outputs updated per-vertex squared-distance minima $d^2_{\min,i}$ that are later consumed by the bound-evaluation stages.

C OTHER RESULTS

Figures 9 and 10 show two additional results to illustrate the Thingi10K-derived benchmarks. Figure 9 uses a mesh pair from the Thingi10K/Decimation benchmark (Model ID: 97674), where mesh A is the original Thingi10K model and mesh B is its 50% decimated version. The CPU baseline requires 169,399.8 ms, whereas **our method** runs in 11.2 ms on an RTX 5090, yielding a **15,187×** speedup. Figure 10 uses a mesh pair from the Thingi10K/TetWild benchmark (Model ID: 1087144), where mesh A is the raw Thingi10K model and mesh B is the boundary surface extracted from a TetWild tetrahedralization. The CPU baseline exceeds the 180 s timeout, while **our method** completes in 15.09 ms on an RTX 5090, achieving more than **11,928×** speedup.

Algorithm 2 Intra-Thread Triangle Packet Traversal

Require: Triangle $T = \{v_0, v_1, v_2\}$, BVH root N_{root}
Ensure: Updated minima $d^2_{\min,0}, d^2_{\min,1}, d^2_{\min,2}$

```
1:  Init: Stack S ← {(N_root, 111_2)};
2:  while S ≠ ∅ do
3:      (N, mask) ← S.Pop()
4:      if mask == 0 then continue
5:      end if
6:      if N is Leaf then
7:          Fetch T_target from SoA
8:          d²_min,i ← min(d²_min,i, Dist²(v_i, T_target))    ∀i ∈ mask
9:      else
10:         Let N_L, N_R be children
11:         mask_L/R ← {i ∈ mask | DistBox²(v_i, N_L/R) < d²_min,i}
12:         if mask_L ≠ 0 and mask_R ≠ 0 then
            // Near-first heuristic
13:             Order children C_near, C_far by box distance
14:             S.Push(C_far, mask_far)
15:             S.Push(C_near, mask_near)
16:         else if mask_L ≠ 0 then
17:             S.Push(N_L, mask_L)
18:         else if mask_R ≠ 0 then
19:             S.Push(N_R, mask_R)
20:         end if
21:     end if
22: end while
```

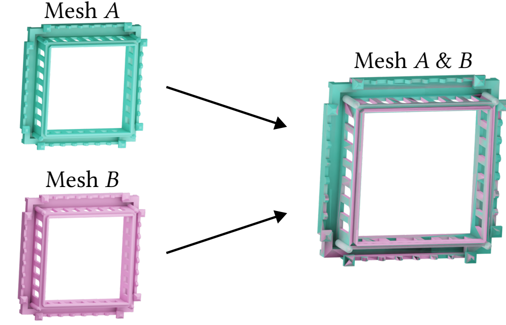

Fig. 9. Thingi10K and Decimation benchmark (Model ID: 97674). Mesh A is the original Thingi10K model and mesh B is its 50% decimated counterpart. The CPU baseline takes 169,399.8 ms, whereas **our method** runs in 11.2 ms on an RTX 5090, yielding a **15,187**× speedup.

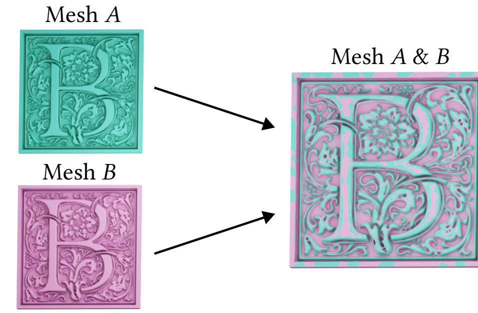

Fig. 10. Thingi10K and TetWild benchmark (Model ID: 1087144). Mesh A is the raw Thingi10K model and mesh B is the boundary surface extracted from the tetrahedral mesh generated by TetWild. The CPU baseline exceeds the 180 s timeout, while **our method** completes in 15.09 ms on an RTX 5090, achieving more than **11,928**× speedup.