gDist: Efficient Distance Computation between 3D Meshes on GPU

PENG FAN, Zhejiang University, China
WEI WANG, Zhejiang University, China
RUOFENG TONG, Zhejiang University, China
HAILONG LI, Shenzhen Poisson Software Co., Ltd., China
MIN TANG*, Zhejiang University, Zhejiang Sci-Tech University, China

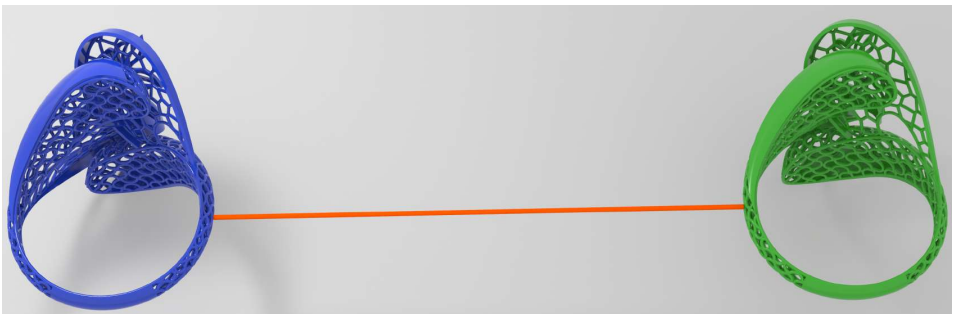

Fig. 1. **Benchmark Rings:** For the pair of rings, each consisting of 7.5M triangles, our algorithm can calculate the minimum distance between them (as indicated by the orange line) within 0.38 milliseconds on an NVIDIA GeForce 4090. As compared to an optimized CPU implementation (PQP [Larsen et al. 2014]), we observe 54X speedups for this specific frame and an average speedup of up to 26X across a sequence of rotation frames.

Computing maximum/minimum distances between 3D meshes is crucial for various applications, i.e., robotics, CAD, VR/AR, etc. In this work, we introduce a highly parallel algorithm (gDist) optimized for Graphics Processing Units (GPUs), which is capable of computing the distance between two meshes with over 15 million triangles in less than 0.4 milliseconds (Fig. 1). By testing on benchmarks with varying characteristics, the algorithm achieves remarkable speedups over prior CPU-based and GPU-based algorithms on a commodity GPU (NVIDIA GeForce RTX 4090). Notably, the algorithm consistently maintains high-speed performance, even in challenging scenarios that pose difficulties for prior algorithms.

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

Additional Key Words and Phrases: Bounding Volume Hierarchy (BVH), Graphics Processing Units (GPUs), Distance Query, Proximity Query Package (PQP)

*Corresponding author, https://min-tang.github.io/home/gDist/, This work was supported by the Leading Goose R&D Program of Zhejiang under Grant No. 2024C01103.

Authors' addresses: Peng Fan, Zhejiang University, China, yingtix2021@163.com; Wei Wang, Zhejiang University, China, wangweiydyk@zju.edu.cn; Ruofeng Tong, Zhejiang University, China, trf@zju.edu.cn; Hailong Li, Shenzhen Poisson Software Co., Ltd., China, lihailong@poissonsoft.com; Min Tang, Zhejiang University, Zhejiang Sci-Tech University, China, tang_m@zju.edu.cn.

ACM XXXX-XXXX/2024/8-ART
https://doi.org/10.1145/3680528.3687619

ACM Reference Format:
Peng Fan, Wei Wang, Ruofeng Tong, Hailong Li, and Min Tang. 2024. gDist: Efficient Distance Computation between 3D Meshes on GPU. 1, 1 (August 2024), 11 pages. https://doi.org/10.1145/3680528.3687619

1 INTRODUCTION

The precise computation of maximum/minimum distance between 3D models is crucial for a spectrum of applications, including robotics, computer-aided design (CAD), virtual reality (VR), and augmented reality (AR). Within CAD systems, distance computation is a key component for tasks such as interference checking and collision detection [Larsen et al. 2014; Shellshear and Ytterlid 2014]. By computing distances between models, CAD systems can rapidly discern potential conflicts, thereby amplifying the precision and dependability of designs. The practical ramifications of this technology extend broadly across diverse domains, encompassing manufacturing, engineering, architecture, physics simulation, and virtual reality. The perennial challenge lies in the optimization of distance computation efficiency, a conundrum persistently grappled with by both industrial and academic communities.

In recent years, Graphics Processing Units (GPUs) have emerged as formidable many-core parallel processors, extensively applied to accelerate various computationally intensive tasks, including surface tessellation [Xiong et al. 2023], mass property computation [Krishnamurthy et al. 2008], collision detection [Tang et al. 2011], ray tracing [Meister et al. 2020], and physically-based simulation [Li et al. 2020; Wang et al. 2020]. However, notwithstanding these strides, the development of efficient GPU algorithms specifically tailored for distance computation remains an challenge.

1.1 Main Results

This paper addresses the efficiency challenge of computing maximum/minimum distances between 3D mesh models. We introduce a meticulously designed parallel algorithm optimized for GPUs, showcasing both efficiency and accuracy. Our algorithm demonstrates the capability to compute the distance between two meshes, totally consisting of over 15 million triangles in less than 0.4 milliseconds. Moreover, it achieves remarkable speedups over prior CPU-based and GPU-based algorithms on a commodity GPU (NVIDIA GeForce RTX 4090). The principal contributions of our work include:

- **Fine-grained parallel execution:** Our algorithm employs a fine-grained parallel design, achieving intricate load balancing and fully exploiting the parallel computational potential of GPUs for optimal performance.
- **Effective AABB-based culling:** Through a novel AABB distance bounding formula, our algorithm introduces a more effective culling method, significantly reducing the computation load and enhancing overall performance.
- **Maximum and minimum distance computing:** Leveraging the unified culling algorithm and task decomposition strategy, our algorithm adeptly handles both maximum and minimum distance computing between 3D mesh models.

These algorithms have been implemented on various GPUs and rigorously evaluated on complex benchmarks. As compared to prior approaches, our algorithm presents the following advantages:

- **Generality:** Our approach exhibits insensitivity to underlying motions, relative size, or distance between models. It accommodates both rigid and deformable models and computes both maximum and minimum distances.
- **Faster performance:** As compared to prior algorithms [Larsen et al. 2014; Pan et al. 2012; Shellshear and Ytterlid 2014], our approach demonstrates approximately one to three orders of magnitude speedups.
- **Lower memory overhead:** For scenes with 15M triangles, the memory overhead is less than 980 Mbytes, allowing our method to run efficiently on GPUs with limited capabilities.

2 RELATED WORK

There are many efficient algorithms [Cameron 1997; den Bergen 1999; Gilbert et al. 1988; Lin and Canny 1991] for determining the closest points between two convex objects. Quinlan [1994] proposed the utilization of a hierarchy of convex bounding volumes for the representation of general non-convex objects. A comprehensive survey on collision detection and distance computation is presented in [Lin et al. 2017].

Larsen et al. [1999] introduced a general paradigm for computing the minimum distance between non-convex objects represented in the bounding volume hierarchies (BVH) of sphere-swept bounding volumes. Their method maintains a priority queue of pairs of BVH nodes, where each BVH node serves as the root for a subtree, representing a volume bounding a subset of the object. Johnson et al. [2005; 1998] adopted a similar approach to compute the minimum distance between two B-spline surfaces. Chang et al. [2011] and Kim et al. [2011] proposed algorithms for distance computation between Bézier surfaces and B-spline surfaces, respectively. Son et al. [2020] empolyed toroidal patches for minimum distance computation for solids of revolution.

The proximity query package (PQP) [Larsen et al. 1999] is considered one of the most efficient algorithms for distance query between 3D mesh models, utilizing rectangular swept sphere (RSS) trees as the acceleration data structure. FCPW [Sawhney 2023] and Embree [Áfra et al. 2016] represent attempts at further acceleration, demonstrating key performance enhancements concerning point-to-mesh distance queries. Shellshear and Ytterlid [2014] employed SSE instructions for acceleration on CPUs, resulting in about a 10% reduction in overall query time. Pan et al. [2012] released an open-source package for collision detection and distance queries, primarily used in robot simulation.

Most algorithms for distance queries between 3D models are based on bounding volume hierarchies (BVHs) [Larsen et al. 1999; Lauterbach et al. 2010; Shellshear and Ytterlid 2014]. As models undergo deformation, these hierarchies are either updated or reconstructed. In order to accelerate the computations on BVH, several parallel algorithms utilizing multiple cores on a CPU or GPU have been proposed [Fan et al. 2011; Heo et al. 2010; Kim et al. 2009; Pabst et al. 2010; Tang et al. 2010; Weller et al. 2017; Zhang and Kim 2014]. Many BVH-based parallel algorithms maintain or update a BVTT (Bounding Volume Traversal Tree) front to accelerate computations [Chitalu et al. 2020; Klosowski et al. 1998; Li and Chen 1998; Tang et al. 2011, 2010; Wang et al. 2018; Zhang and Kim 2014]. These parallel algorithms maintain a BVTT front for parallel collision checking, which can incur a substantial memory overhead.

Recently, Zong et al. [2023] introduced an efficient algorithm for point-to-mesh distance queries. However, extending this approach to distance queries between two meshes proves to be challenging.

3 ALGORITHM PIPELINE

Mathematically, given two 3D models, P and Q in $\mathbb{R}^3$, the maximum and minimum distances between them can be defined as [Lin et al. 2017]:

$$D_{min} = \min_{p\in P}\min_{q\in Q} \ \mathrm{Dist}(p,q), \tag{1}$$

$$D_{max} = \max_{p\in P}\max_{q\in Q} \ \mathrm{Dist}(p,q), \tag{2}$$

where Dist is the distance function, which can be any distance norm. Most standard applications employ the Euclidean distance norm. D_{min} and D_{max} denote the separation distance and the spanning distance between the two models, respectively.

We construct Axis-Aligned Bounding Box (AABB) hierarchies [van den Bergen 1997] for the given models, utilizing them as acceleration data structures. In Section 7.3, we design a data structure called f12-BVH (a full binary tree with 1 or 2 triangles on each leaf node) which is an optimized BVH tailored to improve GPU performance. The f is from "full", 1 and 2 are from the "1 or 2 triangles. Additionally, we maintain BVTT fronts for fine-grained task decomposition and a reduction in the number of pairwise bounding volume (BV) tests [Chitalu et al. 2020; Tang et al. 2011; Wang et al. 2018].

The algorithm's overarching structure is delineated in Algorithm 1. The procedure begins by initializing two buffers (buffer_1 and buffer_2), which represent the BVTT fronts, along with an initial estimate for

Algorithm 1 Parallel Minimum Distance Computing

Input: Two BVHs: $root_A, root_B$
Output: The minimum disance: D_{min}
1: Initialize $buffer_1, buffer_2$ // BVTT fronts
2: $lower, upper \leftarrow$ calculateDistanceBounds$(root_A, root_B)$
3: $buffer_1 \leftarrow root_A, root_B, lower, upper$ // Update the front
4: $upper_{global} \leftarrow upper$
5: **while** Not Reach Leafs **do**
6: $step \leftarrow$ calculateAdaptiveDepth$(buffer_1^{length})$
7: $buffer_2 \leftarrow$ expandBvtt$(buffer_1, upper_{global}, step)$
8: swapBuffer$(buffer_1, buffer_2)$
9: **end while**
10: $D_{min} \leftarrow upper_{global}$
11: **return** D_{min}

the upper and lower bounds of the minimum distance between two AABBs (lines 1-2, Section 4).

buffer$_1$ exclusively contains the BVTT front composed of the two root nodes of the BVHs, and the estimate upper$_{\text{global}}$ is set as the upper bound of the minimum distance between the BVs of the two root nodes (lines 3-4). The algorithm then iteratively expands the BVTT in the buffer until all BVH nodes within the BVTT are leaf nodes (lines 5-9, Section 5), ultimately returning upper$_{\text{global}}$ as the resultant D_{min} (lines 10-11).

In each iteration, an adaptive expansion depth step is computed based on the current buffer size (line 6, Section 6). Subsequently, all BVTT nodes in the existing buffer are expanded, producing a set of new BVTT nodes stored in another buffer (line 7). The two BVTT buffers are then switched using a double-buffer mechanism (line 8).

Symmetrically, the algorithm can be modified to calculate the maximum distances between 3D mesh models, as detailed in the supplementary material. Sections 4 through to 6 will be dedicated exclusively to minimum distance computation. Similar methodologies can be derived for the maximum distance computation.

4 ENHANCED DISTANCE BOUNDS FOR AABBS

A key component of our algorithm is the utilization of AABB distance bounds for culling, as demonstrated in Algorithm 1. Here, we derive enhanced distance bounds for AABBs, which are superior to conventional bounds.

For a given AABB pair, i.e., V_a and V_b, where:

$$V_a = \{P | A_i^{min} \leq P_i \leq A_i^{max}, i = 1, 2, 3\} \tag{3}$$

$$V_b = \{P | B_i^{min} \leq P_i \leq B_i^{max}, i = 1, 2, 3\} \tag{4}$$

The conventional bounds can be defined as follows [Krishnamurthy et al. 2011]. The lower bound of the distance between them, d_{min}^L can be defined as:

$$d_{min}^L = \sqrt{\sum_{i=1,2,3} f_{min}(A_i^{min}, A_i^{max}, B_i^{min}, B_i^{max})^2}, \tag{5}$$

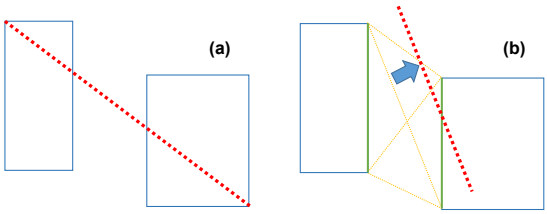

Fig. 2. **Distance Bounds for AABBs:** Compared to the conventional bound computation (a), our new method returns a tighter bound (b). The upper bounds for the minimum distance are highlighted in red.

where function f returns the projected distance between two AABBs on a specific axis, i.e.:

$$f_{min}(A_i^{min}, A_i^{max}, B_i^{min}, B_i^{max}) = \begin{cases} 0 & \text{if the projections overlap} \\ min\{|A_i^{min} - B_i^{max}|, |A_i^{max} - B_i^{min}|\} & \text{otherwise} \end{cases} \tag{6}$$

Symmetrically, the upper bound of the distance between the two AABBs can be defined as:

$$d_{max}^U = \sqrt{\sum_{i=1,2,3} f_{max}(A_i^{min}, A_i^{max}, B_i^{min}, B_i^{max})^2}, \tag{7}$$

where:

$$f_{max}(A_i^{min}, A_i^{max}, B_i^{min}, B_i^{max}) = max\{|A_i^{min} - B_i^{max}|, |A_i^{max} - B_i^{min}|\} \tag{8}$$

A conventional upper bound of the minimum distance between AABBs, $\hat{d}_{min}^U$, is determined by the fastest points contained by them, which is equal to the upper bound of the maximum distance:

$$\hat{d}_{min}^U = max_{P \in V_a, Q \in V_b} ||PQ|| = d_{max}^U \tag{9}$$

Although these conventional bounds are widely used, we see potential for further enhancements to increase culling efficiency. We define an AABB as 'tight' only if a vertex of its encompassing model aligns with a point on each of its six bounding rectangles.

If V_a and V_b are tight, we obtain the following enhanced bounds:

$$d_{min}^U = min\{max_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}, \tag{10}$$

$$d_{max}^L = max\{min_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}, \tag{11}$$

where S represents the six bounding rectangles of each AABB, and $i = 1, 2, ..., 6, j = 1, 2, ...6$. A comparison between $\hat{d}_{min}^U$ (left) and d_{min}^U (right) is demonstrated in Fig. 2.

It can be proven that during the maintenance of a BVH, if all its leaf nodes' AABBs are tight, then all the nodes' AABBs are tight. The overestimation of the upper bound for the minimum distance and the underestimation of the lower bound for the maximum distance do not affect the correctness of the algorithm. Hence, it is not necessary to calculate all 36 pairs of rectangular comparisons. However, in practice, due to the well-hidden cost during the computation process, we choose to compute all 36 pairs of rectangular comparisons.

It is noteworthy that in some applications, it is a common practice to slightly enlarge the bounding box to improve the robustness of

Algorithm 2 Expand BVTT Buffer

Input: $buffer_{input}, upper_{global}, step$
Output: $buffer_{output}$
1: Initialize $buffer_{output}$
2: **for** each $BVTT \in buffer_{input}$ **do**
3: $children_A \leftarrow$ getChildren$(BVTT_A, step)$
4: $children_B \leftarrow$ getChildren$(BVTT_B, step)$
5: **for** each $child_A \in children_A$ **do**
6: **for** each $child_B \in children_B$ **do**
7: $lower, upper \leftarrow$ calculateDistanceBounds$(child_A, child_B)$
8: **if** $lower < upper_{global}$ **then**
9: $bufferd_{output} \leftarrow child_A, child_B, lower$
10: $upper_{global} \leftarrow \min(upper_{global}, upper)$
11: **end if**
12: **end for**
13: **end for**
14: **end for**
15: **return** $buffer_{output}$

the algorithm. However, in this case, employing a similar approach would loosen the bounding box, resulting in underestimated d^U_{min} or overestimated d^L_{max}, leading to incorrect culling.

5 BVTT EXPANSION

Each BVTT node encapsulates two nodes from the BVH trees of two objects, signifying that this BVTT is designed to compute the distance between the objects contained in these two nodes. Initially, there exists a single BVTT node comprising the root nodes of two BVH trees. Typically, a BVTT node can be expanded into four new BVTT nodes, with these four nodes originating from permutations and combinations of the left and right subtrees of the nodes from the two BVH trees.

The procedure for expanding the BVTT in the buffer is outlined in Algorithm 2. It traverses all BVTT nodes in the input buffer and, guided by the adaptive depth step (explained in Section 6), generates descendants for the two BVH nodes contained in the BVTT (lines 3-4). Subsequently, it iterates through all combinations of these descendants (lines 5-13). For each pair, it computes the upper and lower bounds of the minimum distance between the bounding boxes. If the current lower bound of the minimum distance for this pair of bounding boxes surpasses the ongoing (global) estimate of the upper bound of the minimum distance, it implies that this pair of bounding boxes can be disregarded without further expansion. Otherwise, it needs to be appended to the output BVTT buffer (line 9), and the upper bound of the minimum distance for this pair of bounding boxes is used to update the global estimate of the upper bound of the minimum distance (line 10). An illustrative example of BVTT expansion is presented in Fig. 3.

6 ADAPTIVE EXPANSION DEPTH COMPUTATION

In the context of distance computation, sequential algorithms typically employ a heuristic depth-first search (DFS) strategy that swiftly identifies closely located primitives, potentially terminating upon

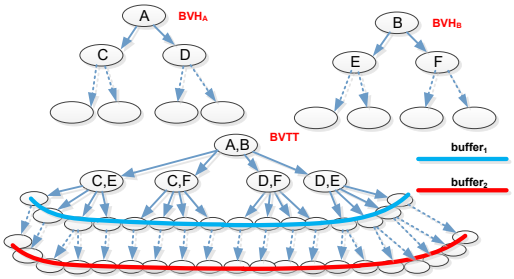

Fig. 3. **BVTT expansion:** We traverse all BVTT nodes in the input front buffer$_1$ and, based on the input adaptive depth step, obtain the expanded BVTT front buffer$_2$.

reaching a leaf node. To illustrate, consider an extreme case where the sequential algorithm discovers the optimal solution upon first reaching a leaf node, enabling immediate termination. In contrast, parallel algorithms based on Breadth-First Search (BFS) typically iterate one layer at a time, with each thread corresponding to a node. When the sequential algorithm rapidly finds the correct answer through heuristic depth-first search, it implies effective pruning for parallel algorithms. In such scenarios, the buffer contains only a small number of BVTT nodes, leading to underutilization of GPU hardware resources, as each iteration involves launching only a limited number of threads to process these BVTT nodes. Moreover, the traditional strategy's extensive unfolding requires frequent CPU setting adjustments based on the GPU's previous computation results, resulting in substantial CPU-GPU synchronization and a consequent drop in algorithm performance.

A direct solution is to expand multiple layers at once. In [Lauterbach et al. 2010], the author experimented with converting BVH into an octree, essentially expanding three layers in BVH. However, adopting a fixed number of expansion layers is suboptimal, as it may lead to either too few or too many BVTT nodes during unfolding. If there are too few nodes, the aforementioned issues persist. Conversely, if there are too many nodes, expanding multiple layers at once might compromise pruning efficiency. A BVTT node, which could have been pruned after the first layer, might be expanded to the third layer, resulting in wasted space and time.

In essence, a dynamic adjustment of the number of layers expanded is required based on the number of nodes. When the buffer contains n BVTT nodes, the adaptive unfolding algorithm aims for a constant C. Specifically, it seeks the maximum expansion depth k such that: $2^{2k}n < C$. Since our f12-BVH is a full binary tree, expanding the n BVTT nodes by k levels will result in fewer than C BVTT nodes. With the depth k, the BVTT nodes are expanded in parallel on the GPU. In practice, C is typically set to 1024×256 for optimal performance, and k is chosen to be 5, corresponding to five expansion levels.

7 PARALLEL DISTANCE COMPUTATION ON GPU

7.1 Parallel Culling

In scenarios where the quantity of BVTT nodes grows exponentially in the outlined algorithm, the challenge arises in controlling their

number by culling some of them. This culling process hinges on estimating the maximum value for the minimum distance between bounding boxes.

In each iteration, we calculate the minimum maximum distance between the bounding boxes of the two BVH nodes in every BVTT node obtained. Subsequently, we execute a reduction operation to identify the minimum among these maximum values. This minimum value is then used as the estimate for the minimum distance. All BVTT nodes with bounding box distances exceeding this estimated distance can be efficiently eliminated in the subsequent iteration. This culling mechanism ensures a controlled and optimized progression of BVTT nodes throughout the algorithm.

7.2 Throughput Consideration

In the context of our parallel algorithm, a pertinent question arises: How can we harness the full parallel high-throughput advantages of the GPU when the number of BVTT nodes is low?

Traditional traversal methods often assign one thread per existing BVTT node, with each thread responsible for the expansion of a single BVTT node. While this approach is standard, especially in distance calculations with the elimination algorithm described earlier, it tends to underutilize the GPU's resources as only a limited number of threads are processed in each kernel call.

Our algorithm introduces a novel strategy by allocating one BVTT node to multiple threads. In this design, each thread corresponds to a newly generated node after expanding the associated BVTT node. For instance, when expanding one layer, if both BVH nodes have left and right children, four possible combinations arise. Consequently, four threads process one BVTT node, with each thread potentially generating a new BVTT. While expanding only one layer is insufficient for optimal GPU resource utilization, the algorithm dynamically determines the number of layers to expand based on the current BVTT node count. For a given number n of BVTT nodes and expanding k layers, the maximum number of newly generated nodes is at most $2^{2k}n$. The algorithm dynamically adjusts k to ensure the generated node quantity falls within an appropriate range. Please refer to Section 6 for the calculation of parameter k.

This parallel approach enables the algorithm to maximize GPU resource utilization, launching the maximum number of threads each time a BVTT node is expanded. Particularly in simpler tasks, this strategy reduces the number of expansion iterations, enhancing overall algorithm efficiency.

7.3 F12-BVH

To implement the described algorithm on the GPU, adjustments to the BVH structure are necessary. The commonly used lbvh [Karras 2012; Lauterbach et al. 2009] proves suboptimal for such operations due to the non-trivial task of finding the $k-th$ descendant, requiring sequential searches based on left and right child pointers. This complicates determining which pair of descendants from two BVH nodes in a BVTT node should be processed by a given thread.

To address this, we propose f12-BVH, designed for enhanced GPU performance, as depicted in Fig. 4. Constructed on the BV sequence arranged by Morton code [Lauterbach et al. 2009], we treat adjacent primitives as one in cases where the number of BVs

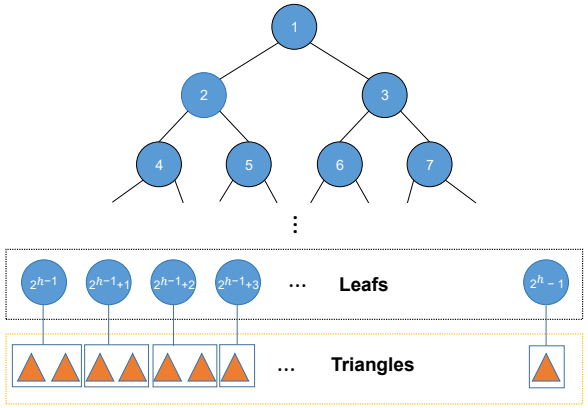

Fig. 4. **F12-BVH:** We propose f12-BVH, a full binary tree with 1 or 2 triangles on each leaf node, designed for enhanced GPU performance.

is not exactly a power of two, adjusting the number of primitives to a power of two. While this may slightly impact BVH quality, the advantages outweigh the trade-offs. The construction of the f12-BVH can be facilitated using a greedy algorithm, as detailed in the supplementary material.

Our algorithm does not rely on any assumption about the triangle order. While our algorithm may not perform as well on extremely poor spatial partitioning. We ensure high efficiency even under less favorable conditions, as demonstrated in benchmarks such as the truck, which has dramatic variations in triangle distribution and size, and the intersection, where the triangles of the two objects are closely packed together.

Accessing the $k-th$ descendant of BVH nodes in a full binary tree is straightforward using bitwise operations. This structure also allows us to determine, based on the thread ID, which newly generated BVTT node the current thread should handle corresponding to the BVTT expansion. In a scenario with n BVTT nodes to be expanded to k layers, the thread with ID t should handle the $\lfloor \frac{t}{2^{2k}} \rfloor - th$ BVTT. The lower $2k$ bits of t encode which descendants of the BVH nodes in BVTT will form a new BVTT node.

A full binary tree can be implicitly stored, and BVH nodes can contain only bounding box information. Our f12-BVH can completely omit any structure-related information, impacting BVH quality but offering advantages. In a full binary tree, no node lacks a left or right child, making each thread perform identical tasks. Threads obtain bounding boxes of two descendants based on the thread ID, calculate the distance between bounding boxes, and decide whether to output the bounding box. Specifically, each thread initially accesses the relevant BVTT. It's noteworthy that adjacent threads often access the same or neighboring BVTT positions, creating coalesced memory access on GPUs. Following this, the threads proceed to access the bounding boxes of the two descendants. In practical terms, the number of layers k expanded at once is frequently substantial. Consequently, adjacent threads predominantly process neighboring nodes of BVH nodes contained within the same BVTT. This characteristic contributes to an efficient and parallelized execution on

the GPU. As the full binary tree is naturally stored in BFS order in memory, the algorithm proves cache-friendly.

7.4 Mantainance of the Minimum Value

Our pruning method hinges on estimating the upper bound of the minimum distance, where the minimum value is derived from the upper bounds calculated by all BVTT nodes.

We adopt a block-wise reduction strategy to maintain the minimum value: Threads store the calculated upper bounds in shared memory, perform a reduction operation within each block, and then one thread in the block uses atomic operations to update the global minimum distance upper bound. It's important to note that because the CUDA-provided atomic operation **atomicMin** doesn't support floating-point types, the atomic operations mentioned above must be implemented manually via **atomicCAS** for integers.

8 IMPLEMENTATION AND RESULTS

We implemented our algorithm on three commodity GPUs, an NVIDIA RTX 2060 (featuring 1960 CUDA cores at 1.7GHz and 6GB memory), an NVIDIA GeForce RTX 3090 (with 10496 CUDA cores at 1.7GHz and 24GB memory), and an NVIDIA GeForce RTX 4090 (with 16384 CUDA cores at 2.52GHz and 24GB memory). These GPUs, characterized by varying core counts, were employed to assess the parallel performance of our approach. The implementation is developed using the CUDA toolkit version 11.7, and Visual Studio 2022 serves as the underlying development environment. All computations on the GPU are conducted using single-precision floating-point arithmetic. The testing environment involves a standard PC running Windows 11 Ultimate 64-bit, equipped with an Intel i5-12400f CPU operating at 2.5GHz and 16GB RAM.

As part of preprocessing, we construct the AABB-based BVH (i.e., f12-BVH) on the CPU. Additionally, for scenes involving rotation motions or deformations, we perform BVH refitting to ensure the algorithm's adaptability to dynamic scenarios. The resulting minimum distance in the presence of penetration is zero. For all our benchmarks, we adhered to a uniform distribution of time intervals.

Our performance evaluation uses 10 diverse benchmarks, each targeting a specific aspect of our algorithm's capabilities:

- **Rings:** Two rings are rotating around their respective axes. Each ring, defined by an axis passing through its local center, comprises 7.5M triangles (Fig. 1).
- **Apples:** Two Voronoi apples are moving apart from each other. Each apple model consists of 720K triangles. (Fig. 10).
- **Tools:** This benchmark features a tool moving in parallel to another tool. Each tool comprises 1M triangles (Fig. 11).
- **Comet-Tool:** This benchmark involves a comet model, comprising 1M triangles, moving through a hole in a tools model, also with 1M triangles (Fig. 12). Importantly, the sizes of the two models significantly differ.
- **Balls:** A smaller Voronoi ball is enclosed by a larger one, and both are rotating synchronously about the same axis (Fig. 13). Each ball model is made up of 130K triangles.
- **Penetration:** A ratchet (with 1M triangles) moves through a hole in a base tool (also with 1M triangles) (Fig. 14). Notably, the two tools are at the same scale.

Benchmarks	Our	PQP	FCL	SSE	NaïveGPU	Speedups
Tools	0.31	463.93	733.48	541.98	2.58	8.45
Rings	0.53	14.04	34.99	12.85	39.72	75.58
Apples	0.21	7.36	15.84	7.42	2.46	11.99
Comet-Tool	0.80	73.49	167.96	81.27	353.53	442.60
Balls	0.49	19.62	37.46	18.47	4.54	9.29
Penetration	0.38	7.36	14.81	7.58	142.19	377.72
Truck	0.19	44.27	55.19	43.65	159.03	836.98
Terrains	0.35	106.60	239.28	122.85	4.72	13.48
MaxDist	0.25	/	/	/	/	/
Deformable	0.51	/	/	/	/	/

Fig. 5. **Performance Comparison:** we conduct a comprehensive comparison with several optimized CPU algorithms (PQP [Larsen et al. 2014], FCL [Pan et al. 2012], and SSE [Shellshear and Ytterlid 2014]) and a 'naïve' GPU implementation (based on [Wang et al. 2018]) (quantified in milliseconds). Across all benchmarks, our GPU algorithms consistently exhibit remarkable speedups, notably outperforming the tested algorithms on an NVIDIA GeForce RTX 4090.

Traingles	Benchmarks	RTX 4090	RTX 3090	RTX 2060
2M	Tools	0.31	0.47	0.68
15M	Rings	0.53	0.98	1.97
1.4M	Apples	0.21	0.31	0.38
2M	Comet-Tool	0.80	1.44	2.39
260K	Balls	0.49	0.90	1.44
2M	Penetration	0.38	0.63	1.07

Fig. 6. **Performance on GPUs:** We rigorously evaluate the performance (quantified in milliseconds) of our GPU-based algorithm on three commodity GPUs, establishing its efficacy across diverse GPU architectures.

- **Terrains:** Two terrain meshes (each with 9.7K triangles) are in close proximity to one other (Fig. 15).
- **MaxDist:** This benchmark mirrors the configuration of the Tools benchmark. However, in this case, the focus is on computing the maximum distance instead of the minimum distance (Fig. 16).
- **Truck:** A truck (1.1M triangles) is traveling between two rows of shelves (2.41M triangles) (Fig. 17). This benchmark consists of objects with non-uniform tessellation and poorly shaped triangles (large and slanted triangles on the shelves).
- **Intersection:** A challenging scene contains two triangle clusters. Each cluster comprises 1 million triangles and intersects at a pre-specified point (Fig. 18).

All the benchmarks are computing the minimum distance between models except the MaxDist benchmark. The tool and the comet models are sourced from [Krispel et al. 2018], while the Voronoi ball, apple, and ring are downloaded from https://www.cgtrader.com/.

8.1 Performance

A comparative analysis (Fig. 5) was conducted against several optimized CPU algorithms, namely PQP [Larsen et al. 2014], FCL [Pan et al. 2012], and SSE [Shellshear and Ytterlid 2014], and a 'naïve' GPU implementation derived from [Wang et al. 2018].

All the CPU algorithms utilize rectangular swept sphere (RSS) trees but employ different elementary test approaches such as triangle-triangle distance computation, GJK algorithm [Gilbert et al. 1988], and SSE-accelerated algorithm. Since we haven't identified any

publicly available GPU implementations for comparison, we developed a 'naïve' GPU solution derived from [Wang et al. 2018]. In this strategy, each thread on the GPU coordinates with a triangle in one model and employs a Depth-First Search (DFS)-like method for traversing the other model's BVH tree. In a CPU single-thread algorithm, it is straightforward to utilize the transient results generated in the DFS process for culling. However, the DFS-like algorithm on GPU necessitates each thread to use atomic operations for modifying/accessing a global variable that stores the current minimum/maximum distance.

In practice, we found that this DFS-like algorithm not only traverses more BVTT nodes (at least nine times, usually dozens of times, in all our benchmarks) but also easily causes severe thread load imbalance issues. We noticed that in some benchmarks, few threads searched too many leaf nodes, leading to a hundredfold decrease in the efficiency of the DFS algorithm. This is often caused by the characteristics of the scene itself. Therefore, although this GPU algorithm achieved improved performance in some scenarios, it could potentially be even slower than existing CPU methods, such as PQP, as depicted in Fig. 5. Across all benchmarks, our GPU algorithms achieved noteworthy speedups over the 'naïve' GPU solution ranging from $8.45 - 836.98X$.

Our algorithm achieved significant speedups in the Comet-Tools and Penetration benchmarks (442.6X and 377.72X, respectively), thanks to the GPU's parallelization, which efficiently handles numerous triangle pairs with the same minimum distance.

The Truck and Terrains benchmarks demonstrate that our algorithm achieves excellent performance for scenarios involving objects with non-uniform tessellation, or scenes containing meshes in a very close distance (such as in the calculation of Hausdorff distance), reporting about 836.98X and 13.48X speedups, respectively. These are common cases in engineering practices, which have verified the generality of our algorithm.

The MaxDist benchmark, designed for maximum distance computing, is not supported by the CPU algorithms mentioned above. Our GPU algorithm showcased an average query time of approximately $0.25ms$ per frame in this scenario. Additionally, the Deformable benchmark underscores our algorithm's adaptability to scenes featuring deformable objects, achieved through dynamic refitting of our AABB-based BVH on-the-fly. In contrast, CPU algorithms face challenges due to prolonged waiting times for updating their underlying bounding volumes (RSSes), rendering them impractical for such dynamic scenes.

Our algorithm undergoes testing under a worst-case scenario configuration, exemplified by Benchmark Intersection [Krispel et al. 2018]. In this particular scene, each model comprises 1 million triangles and intersects at a pre-specified point. This configuration poses a formidable challenge for the 'naïve' GPU solution, requiring approximately 3778 seconds for minimum distance computation between the two models. This scenario is highly specialized, where the BVH itself plays a minimal role. In contrast, our GPU algorithm accomplishes the same task in a mere 0.59 milliseconds, showcasing a remarkable speedup of above $6,403$ times. This emphasizes the robustness and efficiency of our algorithm, especially in scenarios that prove highly challenging for conventional CPU-based and GPU-based approaches.

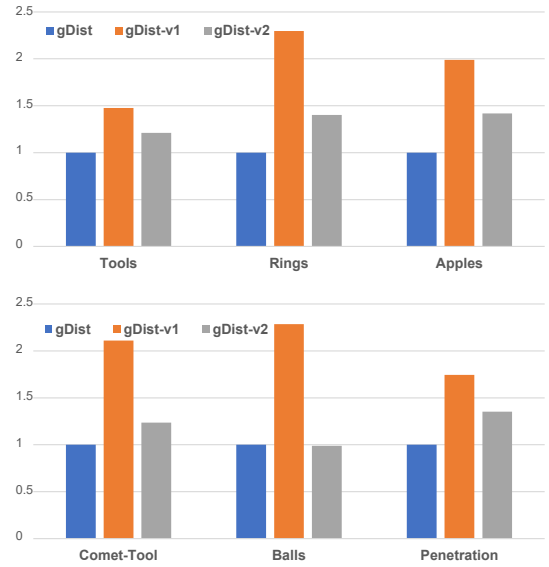

Fig. 7. **Ablation Study 1:** The figure presents an ablation study to highlight the contribution of each component in our method.

Fig. 6 provides an overview of the triangle count across various benchmarks, emphasizing the efficacy of our algorithm in diverse scenarios. The average query time of our GPU-based algorithm on three distinct GPUs (NVIDIA RTX 4090, 3090, and 2060) underscores its compatibility with different GPU architectures, showcasing performance scalability in tandem with the number of CUDA cores.

8.2 Ablation Studies

Figure 7 presents an ablation study to highlight the contribution of each component in our method. For each benchmark, we sampled three implementations for performance comparison: gDist is our proposed algorithm, gDist-v1 is a variant of gDist where the enhanced distance bound computation for AABBs is turned off, and only conventional bound calculation is utilized, and gDist-v2 is another variant of gDist where adaptive BVTT expansion is disabled. In the figure, we normalize the running time of gDist as 1, and display the running times of gDist-v1 and gDist-v2 as ratios over gDist. It can be observed that in different benchmarks, disabling the enhanced bound calculations causes a running time increase of $1.5 - 2.2X$. Similarly, disabling the adaptive BVTT expansion based on f-12 BVH also results in performance deterioration in nearly all scenarios. These ablation studies have clearly illustrated the contributions of different components, and the final performance gain we have achieved is the combined result of these components.

Figure 19 presents another ablation study: Through the use of conventional distance bounds and our enhanced bounds, we can observe the changes in the number of BVTT nodes. We set the number of nodes when using the conventional bounds as 1, and the number when using the enhanced bounds as its proportion. The culling rates in all scenarios are less than 45%. In Benchmark Apples, the number can be less than 4%. This experiment clearly demonstrates how the enhanced bounds effectively reduce the number of BVTT

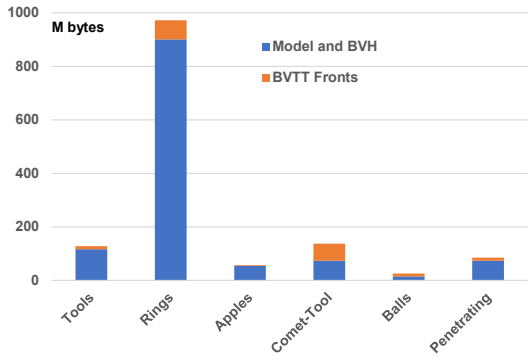

Fig. 8. **Memory Overhead:** The figure compares the GPU memory overhead of our algorithm on different benchmarks. The memory overhead can be broken down into two parts: a static part (for Model and BVH) and a dynamic part (for BVTT fronts).

nodes, thereby decreasing memory usage and computational costs. For complex scenes, e.g. Benchmark Intersection, not employing the enhanced bounds may result in an exceedingly high number of BVTT nodes, leading to the algorithm's crash.

8.3 Running Time Ratios

Figure 9 meticulously dissects the running time ratios across various computation stages for Benchmark Tools (left) and Benchmark Comet-Tool (right) on the NVIDIA GeForce RTX 4090. The results reveal that CPU/GPU Sync stands out as the most computationally intensive segment, constituting a substantial portion (40% – 66%) of the total running time. This aligns with the iterative nature of our algorithm, which necessitates frequent data transfers from the GPU to the CPU. These transfers are crucial for calculating BVTT front expansion depths and determining iteration termination criteria. Typically, CUDA kernel calls operate asynchronously. However, before launching the CUDA kernel for BVTT expansion, the CPU must specify the number of threads, It is a value closely tied to the number of BVTT nodes, and needs to be transmitted from the GPU to the CPU. This operation disrupted the asynchronous execution between CPU and GPU, incurring additional synchronization costs.

8.4 Memory Overhead

Figure 8 presents a comparative overview of the GPU memory overhead for different benchmarks. The memory overhead is dissected into two components: a static part (for Model and BVH) and a dynamic part (for BVTT fronts). The dynamic part showcases the peak memory utilization for each benchmark. The first part involves the BVH tree nodes and information about the model, which is stored on the leaf nodes of the BVH. This part is essential for almost all BVH-based distance query algorithms. The second part pertains to the additional memory consumption introduced by our algorithm, which we want to highlight is very low.

Even in the most intricate scenario (Benchmark Rings with 15M triangles), the BVTT fronts consume less than 72M GPU memory. We utilized around 900M (500M+400M) to store the BVH tree (leaf+non-leaf nodes), and 71.6M for BVTT fronts. This starkly contrasts with earlier GPU algorithms [Tang et al. 2018; Wang et al. 2018], which often incurred approximately 500M memory usage for BVTT fronts. Comparable memory reduction is observed across various benchmarks, attesting to the efficiency of our algorithm in managing GPU memory resources.

9 CONCLUSION AND LIMITATIONS

We introduce gDist, a GPU-based method for distance computation, leveraging an iterative algorithm pipeline (Algorithm 1), tight distance bounds for AABBs (Section 4), parallel BVTT expansion (Algorithm 2), adaptive expansion depth computing (Section 6), and other GPU optimization techniques (Section 7). The efficacy of gDist is demonstrated through extensive benchmarking on complex datasets comprising $19.4K - 15M$ triangles. Our results showcase remarkable speedups and substantially reduced memory overhead compared to previous algorithms. This work introduces a novel algorithm explicitly tailored for GPU parallel mesh-to-mesh distance queries. To the best of our knowledge, no publically available GPU algorithms provided satisfactory performance for this specific task before. Therefore, our approach makes a valuable contribution by addressing the challenge.

Our approach has several limitations. While gDist represents a significant advancement, certain limitations warrant consideration. First, the acceleration achieved by our algorithm is contingent on scene complexity. In simpler scenes with fewer triangles, the potential effectiveness of our parallel acceleration might be constrained. Additionally, the culling effectiveness of BVTT depends on the scenario. For instance, in models like two large concentric spheres where numerous triangles share identical maximum/minimum distances, BVTT culling might not eliminate many detection branches, thereby limiting our acceleration performance. This limitation is inherent, and even CPU algorithms struggle in similar scenarios. Another constraint arises in very complex scenes where GPU memory usage surpasses the capacity of commodity GPUs. In such cases, performing distance calculations in batches becomes necessary, leading to a noticeable decline in performance.

In paper [Quinlan 1994], relative error is used to improve computational efficiency, leading to conservative estimates of minimum and maximum distances. Although this method may enhance culling and computation speed in some scenarios, our primary focus in this paper is the precise distance computation between objects. To ensure accuracy, we chose not to use relative error estimation. We will consider integrating both methods in future work to improve efficiency while maintaining accuracy.

Several promising avenues for future research emerge. Overcoming the identified limitations is a priority, and exploring the computational potential of multiple GPUs through judicious data/task partitioning for parallel distance computation in large-scale scenes presents an intriguing prospect [Li et al. 2020]. Other intriguing avenues for future research involve extending our current 1-to-1 distance query framework to accommodate 1-to-N distance queries, exploring distance computations between SDF-based models [Macklin et al. 2020], as well as using relative errors [Quinlan 1994] for faster and conservative distance computing.

REFERENCES

Attila T. Áfra, Ingo Wald, Carsten Benthin, and Sven Woop. 2016. Embree Ray Tracing Kernels: Overview and New Features. In *ACM SIGGRAPH 2016 Talks* (Anaheim, California) *(SIGGRAPH '16)*. Association for Computing Machinery, New York, NY, USA, Article 52, 2 pages.

Stephen Cameron. 1997. Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra. In *Proceedings of International Conference on Robotics and Automation*, Vol. 4. 3112–3117 vol.4. https://doi.org/10.1109/ROBOT.1997.606761

Jung-Woo Chang, Yi-King Choi, Myung-Soo Kim, and Wenping Wang. 2011. Computation of the Minimum Distance between Two Bézier Curves/Surfaces. *Computers & Graphics* 35, 3 (2011), 677–684. Shape Modeling International (SMI) Conference 2011.

Floyd M. Chitalu, Christophe Dubach, and Taku Komura. 2020. Binary Ostensibly-Implicit Trees for Fast Collision Detection. *Computer Graphics Forum* 39, 2 (2020), 509–521.

Gino Van den Bergen. 1999. A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. *Journal of Graphics Tools* 4, 2 (1999), 7–25. https://doi.org/10.1080/10867651.1999.10487502

Wenshan Fan, Bin Wang, Jean-Claude Paul, and Jiaguang Sun. 2011. A Hierarchical Grid Based Framework for Fast Collision Detection. *Computer Graphics Forum* 30, 5 (2011), 1451–1459.

Elmer G. Gilbert, Daniel W. Johnson, and Sathiya S. Keerthi. 1988. A Fast Procedure for Computing the Distance between Complex Objects in Three-dimensional Space. *IEEE Journal on Robotics and Automation* 4, 2 (1988), 193–203. https://doi.org/10.1109/56.2083

Jae-Pil Heo, Joon-Kyung Seong, DukSu Kim, Miguel A. Otaduy, Jeong-Mo Hong, Min Tang, and Sung-Eui Yoon. 2010. FASTCD: Fracturing-Aware Stable Collision Detection. In *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, MZoran Popovic and Miguel Otaduy (Eds.). The Eurographics Association.

David E. Johnson. 2005. Minimum Distance Queries for Haptic Rendering. *PhD thesis, Computer Science Department, University of Utah.*

David E. Johnson and Elaine Cohen. 1998. A Framework for Efficient Minimum Distance Computations. *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)* 4 (1998), 3678–3684 vol.4.

Tero Karras. 2012. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 33–37.

Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-eui Yoon. 2009. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. *Comput. Graph. Forum* 28 (10 2009), 1791–1800.

Yong-Joon Kim, Young-Taek Oh, Seung-Hyun Yoon, Myung-Soo Kim, and Gershon Elber. 2011. Coons BVH for Freeform Geometric Models. *ACM Trans. Graph.* 30, 6 (dec 2011), 1–8.

James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. 1998. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics* 4, 1 (Jan. 1998), 21–36.

Adarsh Krishnamurthy, Rahul Khardekar, Sara McMains, Kirk Haller, and Gershon Elber. 2008. Performing Efficient NURBS Modeling Operations on the GPU. In *Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling* (Stony Brook, New York) *(SPM '08)*. Association for Computing Machinery, New York, NY, USA, 257–268.

Adarsh Krishnamurthy, Sara McMains, and Kirk Haller. 2011. GPU-Accelerated Minimum Distance and Clearance Queries. *IEEE Transactions on Visualization and Computer Graphics* 17, 6 (2011), 729–742. https://doi.org/10.1109/TVCG.2010.114

Ulrich Krispel, Dieter W. Fellner, and Torsten Ullrich. 2018. A Benchmark for Distance Measurements. In *2018 International Conference on Cyberworlds (CW)*. 120–125. https://doi.org/10.1109/CW.2018.00031

Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. 1999. Fast Proximity Queries with Swept Sphere Volumes. *Technical Report TR99-018, Dept. of Computer Science, UNC.*

Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. 2014. *PQP*. http://http://gamma.cs.unc.edu/SSV/

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David P. Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384. https://doi.org/10.1111/j.1467-8659.2009.01377.x

Christian Lauterbach, Qi Mo, and Dinesh Manocha. 2010. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Comput. Graph. Forum* 29, 2 (2010), 419–428.

Cheng Li, Min Tang, Ruofeng Tong, Ming Cai, Jieyi Zhao, and Dinesh Manocha. 2020. P-Cloth: Interactive Cloth Simulation on Multi-GPU Systems using Dynamic Matrix Assembly and Pipelined Implicit Integrators. *ACM Transaction on Graphics (Proceedings of SIGGRAPH Asia)* 39, 6 (December 2020), 180:1–15.

Tsai-Yen Li and Jin-Shin Chen. 1998. Incremental 3D Collision Detection with Hierarchical Data Structures. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Taipei, Taiwan) *(VRST '98)*. 139–144.

Ming C. Lin and John F. Canny. 1991. A Fast Algorithm for Incremental Distance Calculation. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*. 1008–1014 vol.2. https://doi.org/10.1109/ROBOT.1991.131723

Ming C. Lin, Dinesh Manocha, and Young J. Kim. 2017. Collision and Proximity Queries, Handbook of Discrete and Computational Geometry. Chapman and Hall/CRC.

Miles Macklin, Kenny Erleben, Matthias Müller, Nuttapong Chentanez, Stefan Jeschke, and Zach Corse. 2020. Local Optimization for Robust Signed Distance Field Collision. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3 (2020), 1 – 17. https://api.semanticscholar.org/CorpusID:225967766

Daniel Meister, Jakub Boksansky, Michael Guthe, and Jiri Bittner. 2020. On Ray Reordering Techniques for Faster GPU Ray Tracing. In *Symposium on Interactive 3D Graphics and Games* (San Francisco, CA, USA) *(I3D '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages.

Simon Pabst, Artur Koch, and Wolfgang Straßer. 2010. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Comp. Graph. Forum* 29, 5 (2010), 1605–1612.

Jia Pan, Sachin Chitta, and Dinesh Manocha. 2012. FCL: A General Purpose Library for Collision and Proximity Queries. In *2012 IEEE International Conference on Robotics and Automation*. 3859–3866. https://doi.org/10.1109/ICRA.2012.6225337

Sean Quinlan. 1994. Efficient Distance Computation between Non-convex Objects. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. 3324–3329 vol.4. https://doi.org/10.1109/ROBOT.1994.351059

Rohan Sawhney. 2023. *FCPW: Fastest Closest Points in the West.* https://github.com/rohan-sawhney/fcpw

Evan Shellshear and Robin Ytterlid. 2014. Fast Distance Queries for Triangles, Lines, and Points using SSE Instructions. *Journal of Computer Graphics Techniques* 3, 4 (2014), 86–110.

Sang Son, Seung-Hyun Yoon, M. Kim, and Gershon Elber. 2020. Efficient Minimum Distance Computation for Solids of Revolution. *Computer Graphics Forum* 39, 2 (05 2020), 535–544. https://doi.org/10.1111/cgf.13950

Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. 2018. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 18 (jul 2018), 18 pages. https://doi.org/10.1145/3203188

Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. 2011. Collision-Streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games* (San Fransisco, CA). 63–70.

Min Tang, Dinesh Manocha, and Ruofeng Tong. 2010. MCCD: Multi-Core collision detection between deformable models using front-based decomposition. *Graphical Models* 72, 2 (2010), 7–23.

Gino van den Bergen. 1997. Efficient Collision Detection of Complex Deformable Models using AABB Trees. *Journal of Graphics Tools* 2, 4 (1997), 1–13.

Xinlei Wang, Yuxing Qiu, Stuart R. Slattery, Yu Fang, Minchen Li, Song-Chun Zhu, Yixin Zhu, Min Tang, Dinesh Manocha, and Chenfanfu Jiang. 2020. A Massively Parallel and Scalable Multi-GPU Material Point Method. *ACM Trans. Graph.* 39, 4, Article 30 (aug 2020), 15 pages.

Xinlei Wang, Min Tang, Dinesh Manocha, and Ruofeng Tong. 2018. Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring. *Computer Graphics Forum (Proceedings of Eurographics 2018)* 37, 2 (2018), 1–12.

René Weller, Nicole Debowski, and Gabriel Zachmann. 2017. kDet: Parallel Constant Time Collision Detection for Polygonal Objects. *Computer Graphics Forum* 36, 2 (2017), 131–141.

Ruicheng Xiong, Yang Lu, Cong Chen, Jiaming Zhu, Yajun Zeng, and Ligang Liu. 2023. ETER: Elastic Tessellation for Real-Time Pixel-Accurate Rendering of Large-Scale NURBS Models. *ACM Trans. Graph.* 42, 4, Article 133 (Jul. 2023), 13 pages.

Xinyu Zhang and Y. J. Kim. 2014. Scalable Collision Detection Using p-Partition Fronts on Many-Core Processors. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (March 2014), 447–456.

Chen Zong, Jiacheng Xu, Jiantao Song, Shuangmin Chen, Shiqing Xin, Wenping Wang, and Changhe Tu. 2023. P2M: A Fast Solver for Querying Distance from Point to Mesh Surface. *ACM Trans. Graph.* 42, 4, Article 147 (Jul. 2023), 13 pages.

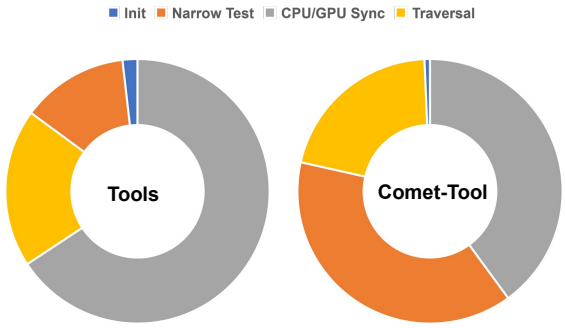

Fig. 9. **Runtime Ratios:** The figure shows the running time ratios of different computing stages: data initialization, BVTT traversal, narrow phrase testing, and CPU/GPU sync. These data are collected by running our system for Benchmark Tools (left) and Benchmark Comet-Tool (right).

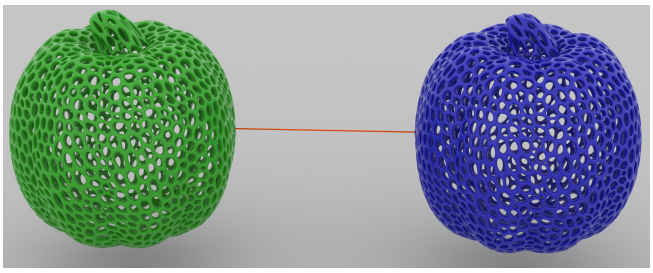

Fig. 10. **Benchmark Apples:** Two Voronoi apples are moving apart from each other. Each apple comprises $720K$ triangles. Compared to the 'naïve' GPU implementation, we observe about 12X speedups along the moving trajectory on an NVIDIA GeForce 4090.

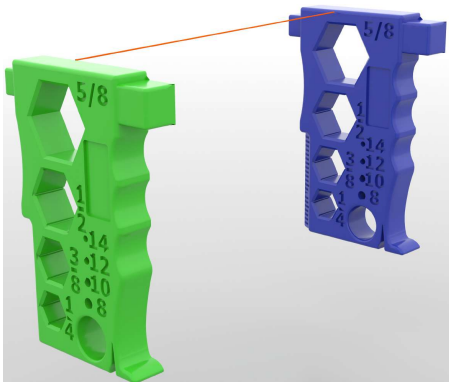

Fig. 11. **Benchmark Tools:** This benchmark features a tool moving in parallel to another tool. Each tool consists of $1M$ triangles. Compared to the 'naïve' GPU implementation, we observe about 8.5X speedups on an NVIDIA GeForce 4090.

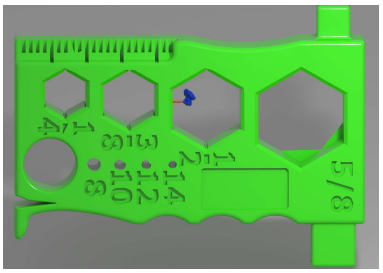

Fig. 12. **Benchmark Comet-Tool:** This benchmark involves a comet model, comprising 1M triangles, moving through a hole in a tools model, also with 1M triangles. The two objects are at quite different scales. Compared to the 'naïve' GPU implementation, we observe above 440X speedups on an NVIDIA GeForce 4090.

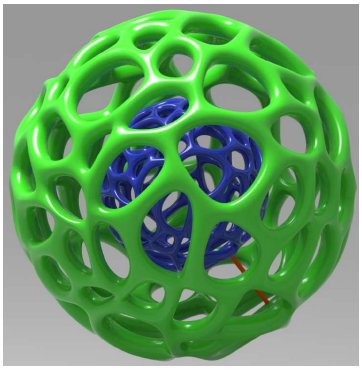

Fig. 13. **Benchmark Balls:** A smaller Voronoi ball is enclosed by a larger Voronoi ball, and both are rotating synchronously about the same axis. Each apple comprises 130K triangles. Compared to the 'naïve' GPU implementation, we observe about 9.3X speedups on an NVIDIA GeForce 4090.

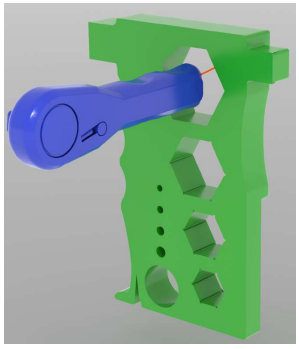

Fig. 14. **Benchmark Penetration:** A ratchet (with 1M triangles) moves through a hole in a base tool (also with 1M triangles). Compared to the 'naïve' GPU implementation, we observe above 377X speedups on an NVIDIA GeForce 4090.

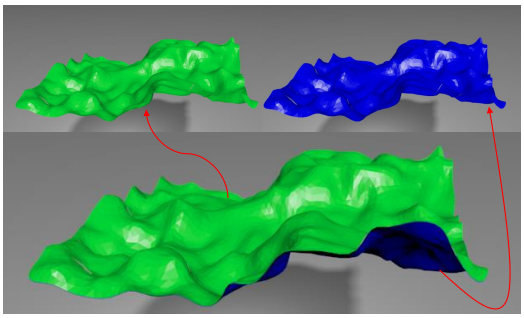

Fig. 15. **Benchmark Terrains:** Two terrain meshes (each with 9.7K triangles) are in close proximity to one other. Compared to the 'naïve' GPU implementation, we observed about 13.5X speedups on an NVIDIA GeForce 4090.

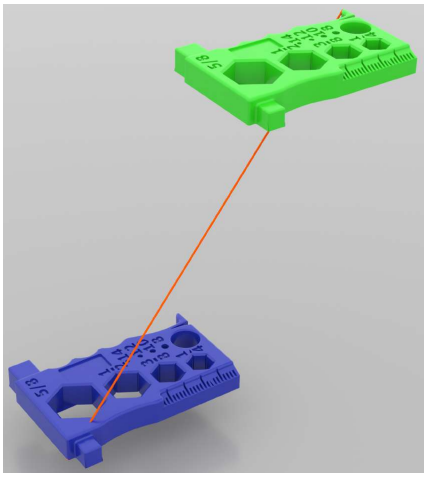

Fig. 16. **Benchmark MaxDist:** This benchmark mirrors the configuration of the Tools benchmark. However, in this case, the focus is on computing the maximum distance instead of the minimum distance. The average query time is about 0.25 milliseconds per frame on an NVIDIA GeForce 4090.

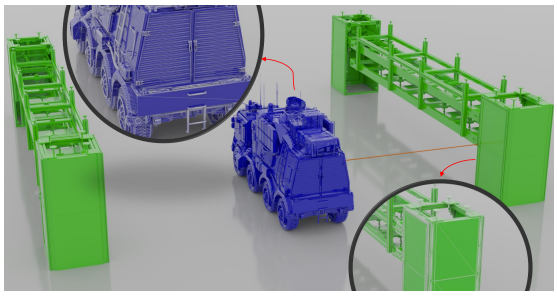

Fig. 17. **Benchmark Truck:** A truck (1.1M triangles) is traveling between two rows of shelves (2.41M triangles), and the objects in this scenario display non-uniform tessellation. Compared to the 'naïve' GPU implementation, we observed about 835X speedups on an NVIDIA GeForce 4090.

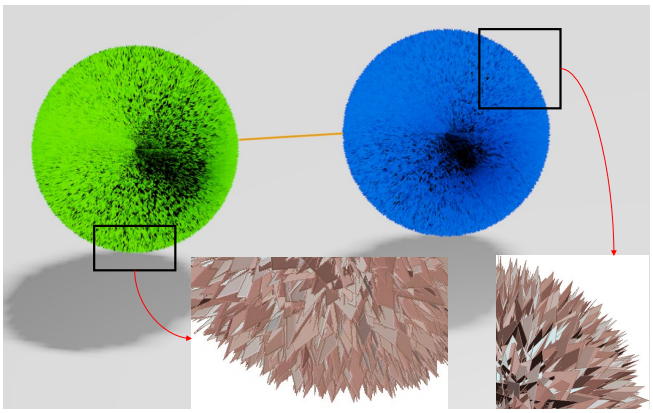

Fig. 18. **Benchmark Intersection:** Each model comprises 1 million triangles and intersects at a pre-specified point. The 'naïve' GPU implementation requires approximately 3778 seconds for minimum distance computation between the two models, while our GPU algorithm accomplishes the same task in a mere 0.59 milliseconds on an NVIDIA GeForce 4090, which is about 6, 403X faster.

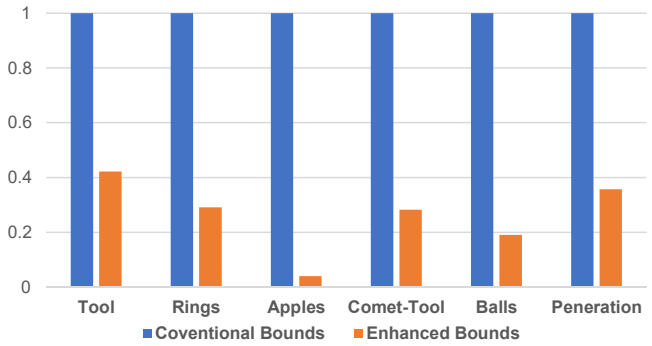

Fig. 19. **Ablation Study 2:** This figure presents another ablation study: Through the use of conventional distance bounds and our enhanced distance bounds, we can observe the changes in the number of BVTT nodes. We set the number of nodes when using the conventional bounds as 1, and the number when using the enhanced bounds as its proportion. The culling rates in all scenarios are less than 45%. In Benchmark Apples, the number can be less than 4%. This experiment clearly demonstrates how the enhanced bounds effectively reduce the number of BVTT nodes, thereby decreasing memory usage and computational costs.

gDist: Efficient Distance Computation between 3D Meshes on GPU Supplemetary Material

PENG FAN, Zhejiang University, China
WEI WANG, Zhejiang University, China
RUOFENG TONG, Zhejiang University, China
HAILONG LI, Shenzhen Poisson Software Co., Ltd., China
MIN TANG*, Zhejiang University, Zhejiang Sci-Tech University, China

ACM Reference Format:
Peng Fan, Wei Wang, Ruofeng Tong, Hailong Li, and Min Tang. 2024. gDist: Efficient Distance Computation between 3D Meshes on GPU Supplemetary Material. 1, 1 (August 2024), 4 pages. https://doi.org/10.1145/3680528.3687619

1 ALGORITHM FOR F12-BVH CONSTRUCTION

Algorithm 1 outlines how the GPU parallelly constructs our f12-BVH. The input parameter t denotes the thread ID, where the t-th thread processes the t-th leaf node of the BVH. n represents the number of triangles, $depth$ is the BVH depth, which can also be calculated from n, and $Triangles$ is the input array of triangles.

In lines 1-12 of the algorithm, we determine the indices of triangles that should be included in the t-th leaf node. The while loop starting from line 4 can be viewed as a search process from the root node to leaf node t, with each search incrementing the depth of the current node (line 11). The condition *OnTheRight* checks if leaf node t is in the right subtree of the current node, a condition easily implemented with bitwise operations.

Lines 13 – 16 of the algorithm build the bounding box for the t-th leaf node, with each leaf node containing only $1-2$ triangles.

Finally, the BVH is constructed in a bottom-up manner (line 17), following the approach outlined in [Chitalu et al. 2020].

2 ALGORITHMS FOR MAXIMUM DISTANCE COMPUTING

Algorithm 2 and 3 outline how we calculate the maximum distance between two BVHs. The overall process of the algorithm is consistent with the minimum distance computing, with only slight differences. During the traversal process, we maintain a lower bound on the maximum distance $\text{lower}_{\text{global}}$ (Algorithm 3, line 10) and eliminate all BVTT nodes whose upper bounds on the maximum distance are smaller than $\text{lower}_{\text{global}}$ (Algorithm 3, line 8).

In addition, since the maximum distance between meshes is equivalent to the distance between their vertices, the primitives stored in our BVH will change from triangle facets to vertices.

Algorithm 1 F12-BVH Construction

Input: $t, n, deep, Triangles$
Output: BVH
1: $l \leftarrow 0$
2: $r \leftarrow n$
3: $d \leftarrow 0$
4: **while** NotReachLeaf$(d, deep)$ **do**
5: **if** OnTheRight$(t, d, deep)$ **then**
6: $l \leftarrow l + \lceil r/2 \rceil$
7: $r \leftarrow \lfloor r/2 \rfloor$
8: **else**
9: $r \leftarrow \lceil r/2 \rceil$
10: **end if**
11: $d \leftarrow d + 1$
12: **end while**
13: Initialize $AABB_{leaf}$
14: **for** $i = l$ to $r - 1$ **do**
15: $AABB_{leaf} \leftarrow Triangles_i$
16: **end for**
17: ConstructBVHBottomUp$(AABB_{leaf}, BVH)$
18: **return** BVH

Algorithm 2 Parallel Maximum Distance Computing

Input: Two BVHs: $root_A, root_B$
Output: The maximum disance: D_{max}
1: Initialize $buffer_1, buffer_2$ // BVTT fronts
2: $lower, upper \leftarrow$ calculateDistanceBounds$(root_A, root_B)$
3: $buffer_1 \leftarrow root_A, root_B, lower, upper$ // Update the front
4: $lower_{global} \leftarrow lower$
5: **while** Not Reach Leafs **do**
6: $step \leftarrow$ calculateAdaptiveDepth$(buffer_1^{length})$
7: $buffer_2 \leftarrow$ expandBvtt$(buffer_1, lower_{global}, step)$
8: swapBuffer$(buffer_1, buffer_2)$
9: **end while**
10: $D_{max} \leftarrow lower_{global}$
11: **return** D_{max}

*Corresponding author, https://min-tang.github.io/home/gDist/, This work was supported by the Leading Goose R&D Program of Zhejiang under Grant No. 2024C01103.

Authors' addresses: Peng Fan, Zhejiang University, China, yingtix2021@163.com; Wei Wang, Zhejiang University, China, wangweiydyk@zju.edu.cn; Ruofeng Tong, Zhejiang University, China, trf@zju.edu.cn; Hailong Li, Shenzhen Poisson Software Co., Ltd., China, lihailong@poissonsoft.com; Min Tang, Zhejiang University, Zhejiang Sci-Tech University, China, tang_m@zju.edu.cn.

ACM XXXX-XXXX/2024/8-ART
https://doi.org/10.1145/3680528.3687619

Algorithm 3 Expand BVTT Buffer(Maximum Distance)

Input: $buffer_{input}, lower_{global}, step$
Output: $buffer_{output}$

```
Initialize buffer_output
for each BVTT ∈ buffer_input do
    children_A ← getChildren(BVTT_A, step)
    children_B ← getChildren(BVTT_B, step)
    for each child_A ∈ children_A do
        for each child_B ∈ children_B do
            lower, upper ← calcDistanceBounds(child_A, child_B)
            if upper > lower_global then
                buferd_output ← child_A, child_B, upper
                lower_global ← max(lower_global, lower)
            end if
        end for
    end for
end for
return buffer_output
```

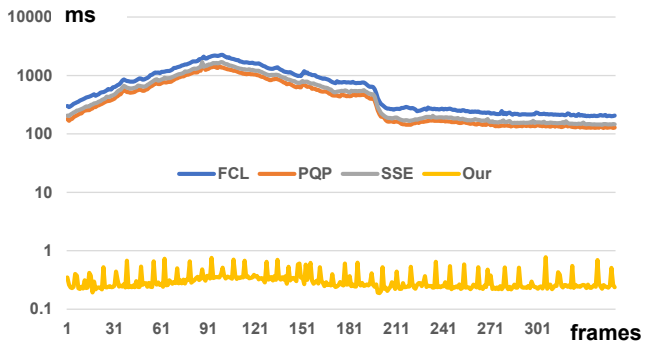

Fig. 1. **Performance Comparison for Benchmark Tools:** The figure compares the performance of distance calculations for each frame of rigid body motion. Compared to PQP, we observe above 1500X speedups on average.

3 PERFORMANCE COMPARISON

Fig. 1 is a performance comparison for Benchmark Tools between PQP [Larsen et al. 2014], FCL [Pan et al. 2012], SSE [Shellshear and Ytterlid 2014] and our algorithm, gDist. To facilitate a more intuitive comparison, the vertical axis in the figure is annotated in exponential notation.

Fig. 2 is a performance comparison for Benchmark Rings at each motion frame. To facilitate a more intuitive comparison, the vertical axis in the figure is annotated in exponential notation.

Fig. 3 is a performance comparison for Benchmark Comet-Tool at each motion frame. To facilitate a more intuitive comparison, the vertical axis in the figure is annotated in exponential notation.

Fig. 4 is a performance comparison for Benchmark Penetration at each motion frame. To facilitate a more intuitive comparison, the vertical axis in the figure is annotated in exponential notation.

Fig. 5 highlights the performance of Benchmark Deformable at each deforming frame. The average query time is about 0.51 milliseconds per frame on an NVIDIA GeForce 4090.

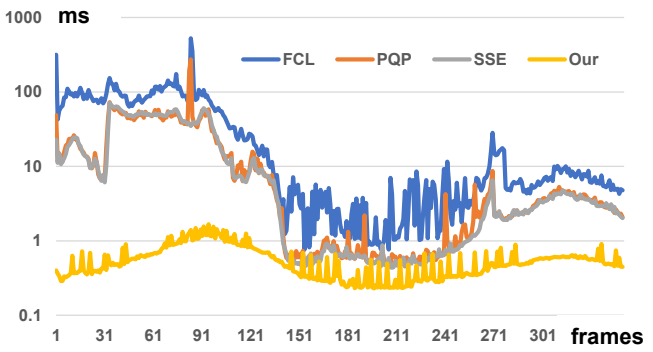

Fig. 2. **Performance Comparison for Benchmark Rings:** The figure compares the performance of distance calculations for each frame of rigid body motion. Compared to PQP, we observe above 26X speedups on average.

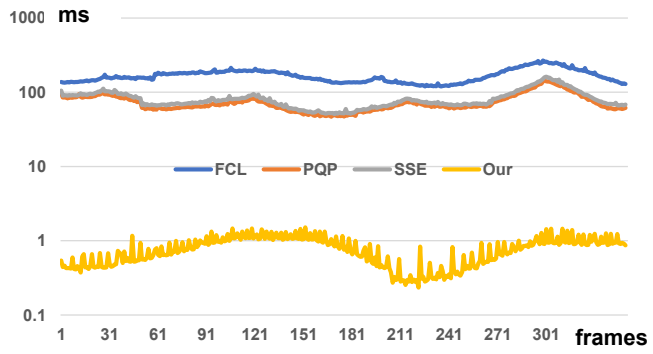

Fig. 3. **Performance Comparison for Benchmark Comet-Tool:** The figure compares the performance of distance calculations for each frame of rigid body motion. Compared to PQP, we observe about 92X speedups on average.

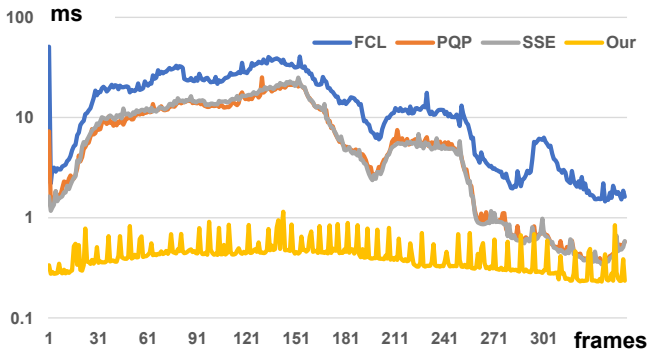

Fig. 4. **Performance Comparison for Benchmark Penetration:** The figure compares the performance of distance calculations for each frame of rigid body motion. Compared to PQP, we observe about 19.56X speedups on average.

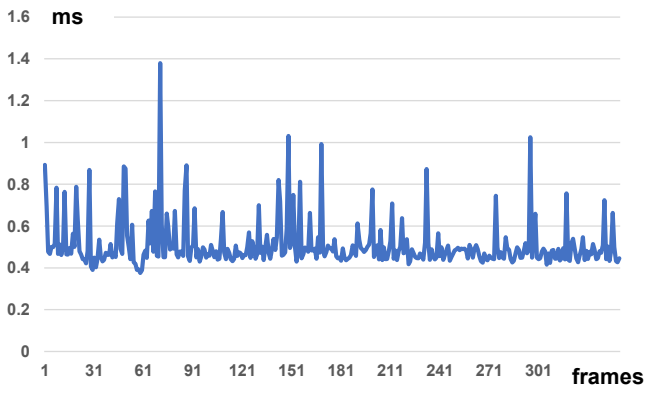

Fig. 5. **Performance of Benchmark Deformable:** The figure highlights the performance of Benchmark Deformable at each deforming frame.

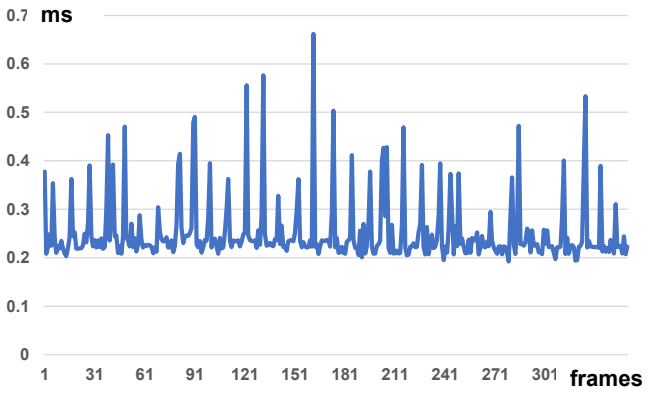

Fig. 6. **Performance of Benchmark MaxDist:** The figure highlights the performance of maximum distance computation for Benchmark MaxDist at each motion frame.

Fig. 6 highlights the performance of maximum distance computation for Benchmark MaxDist at each motion frame. The average query time is about 0.25 milliseconds per frame on an NVIDIA GeForce 4090.

4 TEMPORAL COHERENCY

The inherent temporal coherency of deformable bodies is strategically exploited using a lightweight approach. Specifically, we record model vertices (V_a, V_b) in proximity to the nearest point pair (P_a, P_b) from the previous frame. In the subsequent frame, these recorded vertices (V_a, V_b) aid in initializing estimates for maximum and minimum distances, using $Dist(V_a, V_b)$. This strategy incurs minimal computational cost and yields notable benefits. Notably, this temporal coherency utilization extends beyond deformable bodies, proving advantageous for rigid bodies undergoing continuous motion. Empirical results demonstrate a noteworthy $10\% - 12\%$ speedup compared to an implementation devoid of this temporal coherency strategy.

Fig. 8 illustrates a comparison between distance computations with and without temporal coherency on Benchmark Tools. The

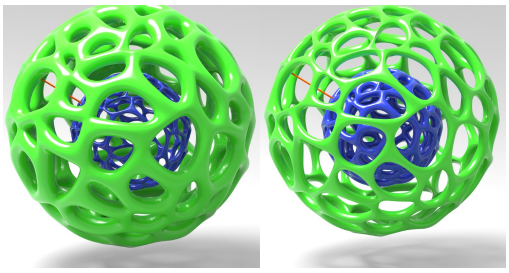

Fig. 7. **Benchmark Deformable:** Two nested Voronoi balls undergo wavelike deformations along their surface normal vectors. Each ball model is composed of 130K triangles. The average query time is about 0.51 milliseconds per frame on an NVIDIA GeForce 4090.

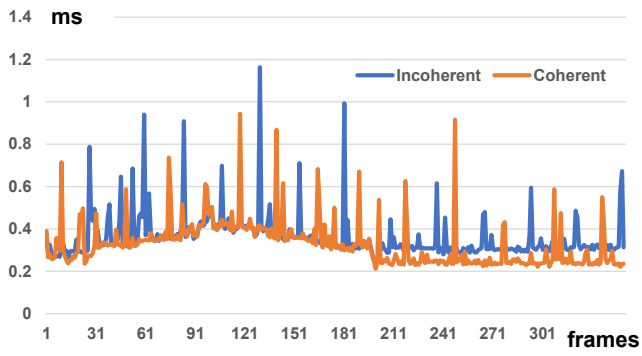

Fig. 8. **Speedups with temporal coherency on Benchmark Tools:** Figure 8 illustrates a comparison between distance computations with (in orange) and without (in blue) temporal coherency on Benchmark Tools. The utilization of temporal coherency in this benchmark results in a notable performance improvement of 10.6%.

utilization of temporal coherency in this benchmark results in a notable performance improvement of 10.6%. In Fig. 7, two nested Voronoi balls undergo wavelike deformations along their surface normal vectors. Each ball model is composed of 130K triangles (Fig. 7). The average query time is about 0.51 milliseconds per frame on an NVIDIA GeForce 4090.

Temporal coherence provides a good estimate of the upper bound of the minimum distance in our algorithm. However, as we can obtain a reasonably good estimate even without temporal coherence after a few expansions, its impact on the subsequent execution of the algorithm is limited.

In the event of a significant distance between the previous and subsequent frames, the algorithm's reliance on model vertices V_a and V_b from the previous frame may have a smaller acceleration effect. However, this optimization is designed to provide a reasonable initial estimate of the upper bound of the minimum distance. Even with a large deformation between frames, it won't lead to incorrect results.

Fig. 9 illustrates a comparison between distance computations with and without temporal coherency on Benchmark Rings. The utilization of temporal coherency in this benchmark results in a notable performance improvement of 12.6%.

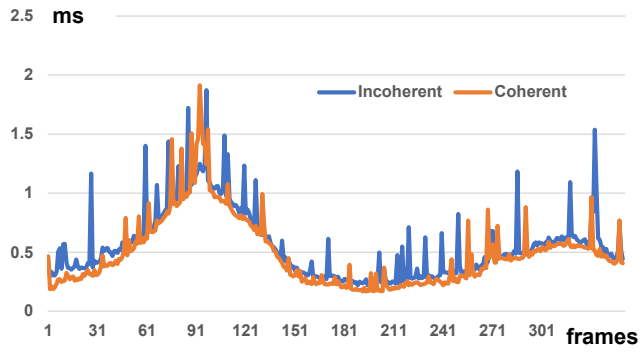

Fig. 9. **Speedups with temporal coherency on Benchmark Rings:** Figure 9 illustrates a comparison between distance computations with (in orange) and without(in blue) temporal coherency on Benchmark Rings. The utilization of temporal coherency in this benchmark results in a notable performance improvement of 12.6%.

5 COMPARISON WITH CPU/GPU IMPLEMENTATIONS

Our algorithm is generic, as demonstrated by testing on three different GPUs. We did not conduct any specific tuning for the targeted GPU; however, the tuning could further improve performance.

The insights gained from our approach may not be directly applicable to accelerate CPU-based algorithms. Our algorithm is tailored for GPU architecture. While the tighter distance bounds benefit CPU performance, the fine-grained BVTT expansion, optimized for GPU parallelism, might lead to suboptimal CPU performance.

6 BVH SHARING

Regarding BVH sharing, we deliberately chose not to share the BVHs of the two identical meshes. While BVH sharing is a common practice, we opted for algorithmic generality, treating each mesh independently. Considering body-space BVH for scenes with rigid bodies could offer advantages by avoiding the cost of mesh memory transfer and BVH construction, but it may compromise the algorithm's generality, especially for scenes with deformable objects.

7 PROOF OF ENHANCED BOUND

Given two meshes $Mesh_A$ and $Mesh_B$ with their bounding AABB V_a and V_b respectively. V_a and V_b are defined as:

$$V_a = \{P | A_i^{min} \leq P_i \leq A_i^{max}, i = 1, 2, 3\} \tag{1}$$

$$V_b = \{P | B_i^{min} \leq P_i \leq B_i^{max}, i = 1, 2, 3\} \tag{2}$$

We term an AABB as tight only when at least one point on each of its six bounding rectangles coincides with a vertex of its encompassing model.

Theorem 1: *The upper bound, d^*, for the minimum distance between AABBs, is,*

$$d^* = min\{max_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}, \tag{3}$$

where S represents the six bounding rectangles of each AABB, and $i = 1, 2, ..., 6$, $j = 1, 2, ...6$. Further, d^* also is an upper bound for the minimum distance between $Mesh_A$ and $Mesh_B$.

Proof: Given that two AABBs are convex polyhedra, the points of minimum distance between them necessarily fall on their boundary faces, ie., F_a and F_b. So the minimum distance between the two AABBs, d, must fulfill:

$$d \leq d^* = min\{max_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}. \tag{4}$$

Moreover, since the AABBs here are tight, there must exist vertices V_a in F_a and V_b in F_b such that the distance $d(V_a, V_b) < d^*$. Thus, d^* also serves as an upper bound for the minimum distance between $Mesh_A$ and $Mesh_B$.

Theorem 2: *Given two tight AABB V_a and V_b, there is always an inequality that holds:*

$$d_{min}^U = d^* \leq \hat{d}^* = \hat{d}_{min}^U, \tag{5}$$

where $\hat{d}_{min}^U$ is determined by the fastest points contained by the two AABBs.

Proof: Given that two AABBs are convex polyhedra, the points of maximum distance between them necessarily fall on their boundary faces, ie., F_a and F_b. So the maximum distance between the two AABBs, $\hat{d}^*$, must fulfill:

$$\hat{d}^* = max\{max_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}. \tag{6}$$

Since

$$d^* = min\{max_{P \in S_i^{V_a}, Q \in S_j^{V_b}} ||PQ||\}, \tag{7}$$

so we conclude: $d^* \leq \hat{d}^*$.

REFERENCES

Floyd M. Chitalu, Christophe Dubach, and Taku Komura. 2020. Binary Ostensibly-Implicit Trees for Fast Collision Detection. *Computer Graphics Forum* 39, 2 (2020), 509–521.

Eric Larsen, Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. 2014. *PQP*. http://http://gamma.cs.unc.edu/SSV/

Jia Pan, Sachin Chitta, and Dinesh Manocha. 2012. FCL: A General Purpose Library for Collision and Proximity Queries. In *2012 IEEE International Conference on Robotics and Automation*. 3859–3866. https://doi.org/10.1109/ICRA.2012.6225337

Evan Shellshear and Robin Ytterlid. 2014. Fast Distance Queries for Triangles, Lines, and Points using SSE Instructions. *Journal of Computer Graphics Techniques* 3, 4 (2014), 86–110.