A Case Study for Ray Tracing Cores: Performance Insights with Breadth-First Search and Triangle Counting in Graphs

ZHIXIONG XIAO, Shandong University, China
MENGBAI XIAO*, Shandong University, China
YUAN YUAN*, Shandong University, China
DONGXIAO YU, Shandong University, China
RUBAO LEE, Freelance, USA
XIAODONG ZHANG, The Ohio State University, USA

The emerging Ray-tracing cores on GPUs have been repurposed for non-ray-tracing tasks by researchers recently. In this paper, we explore the benefits and effectiveness of executing graph algorithms on RT cores. We re-design breadth-first search and triangle counting on the new hardware as graph algorithm representatives. Our implementations focus on how to convert the graph operations to bounding volume hierarchy construction and ray generation, which are computational paradigms specific to ray tracing. We evaluate our RT-based methods on a wide range of real-world datasets. The results do not show the advantage of the RT-based methods over CUDA-based methods. We extend the experiments to the set intersection workload on synthesized datasets, and the RT-based method shows superior performance when the skew ratio is high. By carefully comparing the RT-based and CUDA-based binary search, we discover that RT cores are more efficient at searching for elements, but this comes with a constant and non-trivial overhead of the execution pipeline. Furthermore, the overhead of BVH construction is substantially higher than sorting on CUDA cores for large datasets. Our case studies unveil several rules of adapting graph algorithms to ray-tracing cores that might benefit future evolution of the emerging hardware towards general-computing tasks.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; • **Theory of computation** → **Parallel algorithms**; **Graph algorithms analysis**.

Additional Key Words and Phrases: Ray Tracing, Breadth-First Search, Triangle Counting, GPU, OptiX

ACM Reference Format:
Zhixiong Xiao, Mengbai Xiao, Yuan Yuan, Dongxiao Yu, Rubao Lee, and Xiaodong Zhang. 2025. A Case Study for Ray Tracing Cores: Performance Insights with Breadth-First Search and Triangle Counting in Graphs. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 2, Article 16 (June 2025), 25 pages. https://doi.org/10.1145/3727108

1 Introduction

Graphs are a fundamental data structure widely adopted in real-world applications, including social network analysis [17], transportation systems [24], biological networks [47], and chemical structures [54]. With the ever-growing scale of graphs, processing them demands substantial

*Corresponding author.

Authors' Contact Information: Zhixiong Xiao, Shandong University, Qingdao, China, xiaozxiong@mail.sdu.edu.cn; Mengbai Xiao, Shandong University, Qingdao, China, xiaomb@sdu.edu.cn; Yuan Yuan, Shandong University, Qingdao, China, yyuan@sdu.edu.cn; Dongxiao Yu, Shandong University, Qingdao, China, dxyu@sdu.edu.cn; Rubao Lee, Freelance, Columbus, USA, lee.rubao@ieee.org; Xiaodong Zhang, The Ohio State University, Columbus, USA, zhang@cse.ohio-state.edu.

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.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2476-1249/2025/6-ART16
https://doi.org/10.1145/3727108

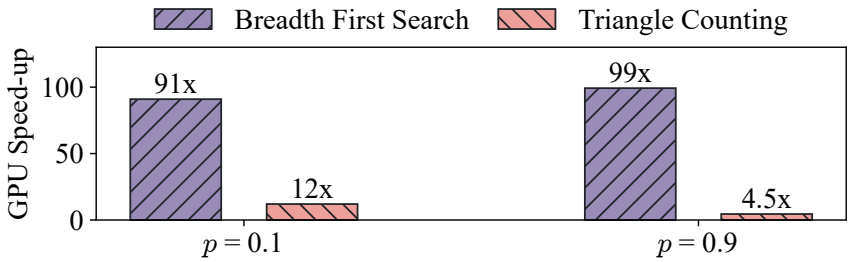

Fig. 1. The speed-up of Gunrock [58] to Galois [44] is evaluated for breadth-first search (BFS) and triangle counting (TC) on the same machine. Two power-law graphs having 1 million vertices are generated following the Barabási-Albert growth model [28] with the parameter p set to 0.1 and 0.9, where p indicates the probability of forming a new triangle by adding a random edge.

computational power. Graphics Processing Units (GPUs) become a promising platform due to the capability of massive parallelism. Various GPU-based graph processing frameworks [4, 19, 32, 52, 57, 58, 63] have been proposed to ease the algorithm implementation while reaching high-performance.

However, the performance gains of graph algorithms differ on the GPU device. The breadth-first search or PageRank algorithms featuring highly data-parallel could be effectively accelerated with proper workload partition. The execution of subgraph matching is less efficient because it is control flow-intensive and hardly fits the single instruction, multiple threads (SIMT) execution model of CUDA cores. Figure 1 shows the GPU speed-up for breadth-first search and triangle counting on synthesized graphs with 1 million vertices, respectively. While the GPU-based breadth-first search is nearly two orders of magnitude faster than its CPU-based counterpart, triangle counting is only accelerated by 4x-12x, depending on the skewness of the graph.

On the other hand, Ray Tracing (RT) cores are introduced to Nvidia GPUs since Turing architecture [9], which is a specialized hardware designed to accelerate ray tracing algorithms. An RT core sits on a Streaming Multiprocessor (SM) but is independent of CUDA cores. With the support of RT cores, the complex computation pipeline of ray tracing is efficiently realized on GPU. Specifically, RT cores efficiently detect the intersections between rays and graphic primitives based on hardware-accelerated Bounding Volume Hierarchy (BVH) [15]. Since BVH is an indexing structure, researchers are exploring the use of RT cores for more general tasks, like searching for specific numbers in a dataset [27, 36, 38]. These studies have demonstrated the potential of RT cores toward search workloads with intensive control flows.

In this work, we explore the acceleration of graph algorithms using RT cores through two case studies. Our objective is to gain performance insights into the dynamic interactions between graph algorithm workloads and the internal mechanisms of RT cores. We choose *breadth-first search (BFS)* and *triangle counting (TC)* as the representatives: BFS is dominated by sequential memory accesses but the performance knob of TC is set intersection, which is essentially searches [29–31]. The different characteristics of the two workloads are expected to help us gain more comprehensive insights into RT-based graph algorithms. To realize the graph algorithms on RT cores, we design RT-based BFS and RT-based TC. In both methods, we strive to construct an indexing structure with BVH to represent the graph, and use rays to accomplish the task. In RT-based BFS, the adjacency list of a vertex is mapped to one or a few lines of primitives placed at a position the same as the vertex ID, and we could issue a ray accordingly to simulate the visiting. We further design a scheme encoding multiple neighbors into a primitive for improving the memory efficiency. In RT-based TC, since the triangles are discovered by searching for the common elements between 2-hop neighbors and 1-hop neighbors of a vertex, we then organize the 2-hop neighbors into the BVH and issue rays

representing the 1-hop neighbors, or vice versa. In both versions of RT-based TC, ray-primitive intersections indicate triangles are found.

We evaluate our RT-based graph algorithms on extensive real-world datasets. RT-based BFS can hardly compete with the conventional CUDA-based baselines in our experiments. Building a primitive to represent a vertex increases its memory footprint by one order of magnitude. RT-based BFS issues 4.03x more memory instructions on average compared to its CUDA-based counterpart. The difference in memory accesses closely matches their performance gap, which is 3.94x on average. By encoding 1.91 vertices into a primitive on average, our scheme effectively reduces the memory instructions of the RT-based method. But it still issues 1.77x memory instructions than the CUDA-based baseline, and the execution time is 3.67x slower. RT-based TC shows superior performance compared to the CUDA-based baselines in our experiments. However, with careful analysis and experimental confirmation, the performance gain comes from our fine-grained workload partition instead of the hardware acceleration of RT: By replacing the BVH traversal with the binary search on CUDA cores, our implementations are further accelerated. In addition, the two versions of our RT-based TC methods (searching for 1-/2-hop neighbors among 2-/1-hop neighbors) also expose their shortcomings, i.e., high memory footprint and high miss ratios of rays, respectively. To avoid our evaluation of RT-based TC being limited only to a small scope of datasets, we carry out further experiments to evaluate RT-based set intersection on synthesized datasets with varying selectivity, skew ratio, set size, and density. The results show that the RT-based method gains its advantage over the fastest CUDA-based baseline when the skew ratio is high, which means the RT core is expected to accelerate searches in a large dataset.

In the end, we want to understand why the performance of RT cores is varying in cases. We experimentally compare RT-based binary search and CUDA-based binary search and enable Nsight Compute [12] for profiling. As both methods follow a logarithmic complexity $O(\log n)$, the RT-based search shows superior performance than the CUDA-based one in the large-n range, where n is the scale of a dataset to be searched in. However, the worse performance of the RT-based search in the wide small-n range exposes that the RT programming pipeline has a constant and non-trivial overhead, which limits the scope of deploying RT cores as a general-purpose accelerator. Moreover, the overhead of constructing a BVH is smaller than sorting on CUDA cores only in the small-n range, which offsets the gains of RT-based methods if the preparation time must be considered, further limiting the usage of RT cores. Overall, the in-depth evaluation of graph workloads on RT cores clarifies the scope of acceleration, and to extend the scope, a more memory-efficient index, a wider support to data types, and the capability of circumventing fixed programming pipeline is expected in the future-generation RT cores.

The contributions of the paper are as follows:

- We design and implement RT-based breadth-first search and RT-based triangle counting, demonstrating that graph algorithms could be mapped to ray tracing problems and be executed correctly.
- We evaluate our RT-based algorithms on extensive datasets, showing that the RT-based BFS is always slower but the RT-based set intersection is faster when the skew ratio of datasets is high.
- With the careful analysis of the binary search workloads, we identify that the scope of RT core as an accelerator for graph algorithms is limited due to its overhead of execution pipeline and BVH construction.

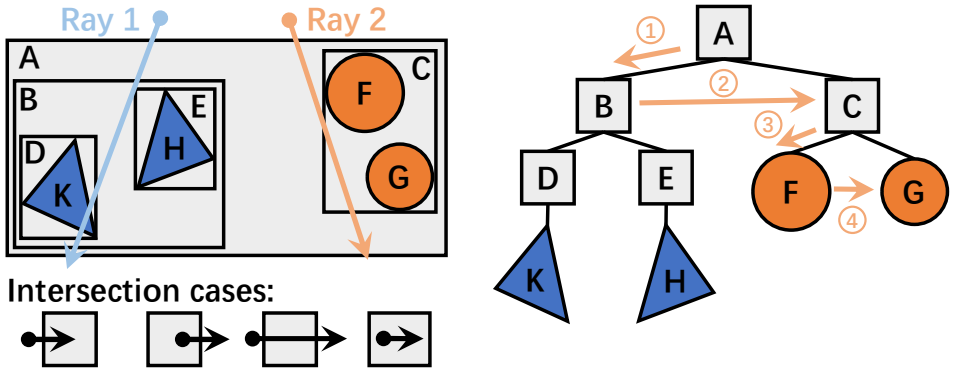

Fig. 2. A 2D example of BVH built for two triangles and two spheres. A, B, C, D, and E are AABBs, while K, H, F, and G are geometric primitives. Testing ray 1 follows the traversal path of A (hit) → B (hit) → D (hit) → K (hit) → E (miss) → C (miss). Testing ray 2 follows the traversal path of A (hit) → B (miss) → C (hit) → F (miss) → G (miss). For clarity, we show only the traversal of ray 2 in the figure.

2 Background

2.1 Hardware-Accelerated Ray Tracing

Ray tracing is a technique that models the propagation of light for rendering photorealistic frames. It traces rays from the camera until intersections with objects in a 3D scene. To efficiently test where intersections happen, *bounding volume hierarchy (BVH)* is developed to organize geometric primitives in a tree structure. It allows the intersection test to be mapped to a BVH traversal, which has a lower algorithmic complexity than brute-force comparisons. While intersection tests dominate the rendering time even with BVH [55], Nvidia introduces RT cores onto its GPUs since Turing architecture [7], which are dedicated hardware accelerating the traversal of BVH trees. On RT cores, coordinates of geometric primitives are represented using `FP32`.

BVH Traversal. In a BVH, geometric primitives are wrapped in bounding volumes like axis-aligned bounding boxes (AABBs). Multiple AABBs are enclosed in a larger AABB as their parent until an AABB contains all of them, forming a tree structure. Figure 2 shows an example of a BVH tree built for a few primitives in a 2D scene.

The intersection test between a ray and the primitives is realized by traversing BVH in a depth-first manner: We start from the root AABB, testing if it intersects with the ray. If an intersection happens, we recursively test AABBs the root contains. Otherwise, we skip the root. As the traversal reaches a leaf node, the intersection test between the ray and the primitive is carried out. The traversal stops only if all intersected AABBs have been visited, which means that multiple intersections could occur for a ray. It is worth noting that a ray with two ends in the AABB can also be identified as an intersection. Figure 2 also illustrates how to traverse a BVH tree.

Hardware Acceleration. The traversal of BVH is accelerated by RT cores on Nvidia GPUs. On the latest Ada Lovelace architecture, one streaming processor (SM) includes one RT core and 128 CUDA cores [10]. Tracing a ray is accomplished by interleaving executing programs on two types of cores. The CUDA cores generate a ray per thread, after which the BVH traversal is executed by the RT core. Once intersections are found or the traversal ends, the CUDA cores take the control back, executing user-defined callbacks.

Specifically, one could follow the programming model with Nvidia OptiX APIs [11], in which user-defined callbacks include *ray-generation*, *intersection*, *any-hit*, *closest-hit*, and *miss*. *ray-generation* generates rays with given origins and directions, and *intersection* is executed when traversing to

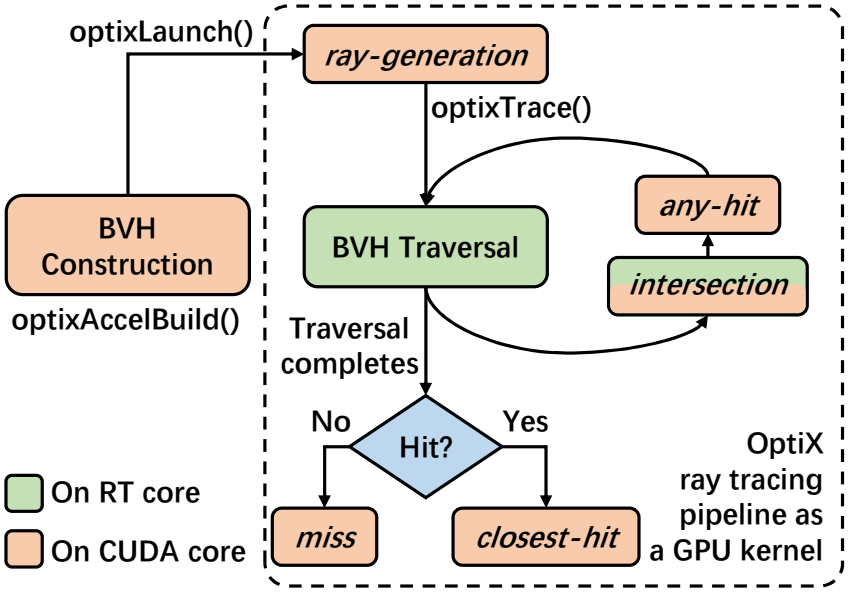

Fig. 3. The ray tracing pipeline of OptiX. `optixAccelBuild` builds the BVH of primitives, and `optixLaunch` starts the ray tracing pipeline. An RT program (a GPU kernel) interleaves callbacks on CUDA cores and BVH traversal on RT cores. *ray-generation* initiates BVH traversal with `optixTrace`.

a leaf node, identifying if there is a ray-primitive intersection. *any-hit* is called if an intersection occurs. While *intersection* is flexible enough for testing a ray against any geometric primitive, the ray-triangle intersection test is hardware-accelerated on the RT core, in which case *any-hit* is called directly [9, 41]. When a traversal ends, *closest-hit* or *miss* is invoked depending on whether an intersection is found. Figure 3 illustrates the ray tracing pipeline of OptiX.

2.2 General Computing on RT cores

With the broad integration of RT cores in recent commodity GPUs, researchers are increasingly interested in developing hardware-accelerated non-ray tracing tasks, which is similar to how the rasterization pipeline was repurposed in the pre-CUDA era. Accelerating other computation tasks also in Euclidean space shows promise, like locating a point in tetrahedral meshes [56], searching the k-nearest neighbors [16, 35, 37, 43, 61, 64], and unsupervised clustering in 3-dimensional space [42]. These tasks benefit from RT cores because the commonly used indexing structure, such as kd-tree [5], could be effectively replaced by the hardware-accelerated BVH.

On the other hand, efforts have been made to leverage RT cores for more general search workloads [2, 25, 27, 36, 38], which motivate us to accelerate graph algorithms using RT cores. GPU-based frameworks such as Gunrock [58] and SEP-graph [57] have advanced the performance of graph analytics to the cutting edge. However, the control-flow-intensive operations challenge the deployment of graph algorithms on GPUs. An example is set intersection, which is a building block of triangle counting [31], path joining [62], and more general subgraph matching [8]. The set intersection on GPU could be categorized into merge path-based [21, 22], binary search-based [18, 31], and hash indexing-based [46]. Among them, binary search-based is recognized to work the best [8, 58]. Following the previous studies generalizing RT cores for search workloads, we would like to explore if this emerging hardware could accelerate graph algorithms.

Algorithm 1: RT-based BFS

Input: A graph $\mathcal{G}$, a source vertex s
Output: the level of vertices stored in a key-value table of (v, l)

```
1  bvh ← BuildBVH (G)                                  ▷ Build BVH
2  levelTable ← ∅                                      ▷ Initialize the level of each vertex
3  level ← 0
4  que ← s                                             ▷ Initialize the queue
5  while que is not empty do
6      Filter (que)                                    ▷ Remove duplicate vertices
7      new_que ← ∅
8      RT-Expand (bvh, que, new_que, levelTable, level)
9      cudaDeviceSynchronize()                         ▷ CPU-side global synchronization
10     que ← new_que
11     level ← level + 1
12 end
```

3 RT-based BFS and RT-based TC

In this section, we present how to adapt graph algorithms to the RT programming pipeline for evaluation. We re-design two representative algorithms, *breadth-first search (BFS)* and *triangle counting (TC)*. Traversal is a fundamental algorithm in graphs, whose basic operation is sequentially reading neighbors. The most well-known traversal algorithms are depth-first search (DFS) and BFS. Since BFS favors the GPU architecture [34, 40, 51], we narrow the discussion to BFS. We should notice that there is no search operation in a traversal algorithm, so the hardware-accelerated BVH traversal might not benefit BFS. But how the performance varies when turning the graph traversal from CUDA cores to RT cores is still an interesting problem. For the completeness of our research, we realize and evaluate RT-based BFS.

On the other hand, TC is a classic graph-mining algorithm incorporating search operations. For a vertex and one of its neighbors, if they share a common neighbor, the vertex, the neighbor, and the common neighbor together form a triangle. Discovering a common neighbor is essentially to search a neighbor (of a vertex) in a neighbor list (of another vertex), and the hardware-accelerated BVH traversal is expected to benefit RT-based TC.

3.1 RT-based BFS

Starting from a source vertex, the basic BFS algorithm pushes neighbors in its adjacency list to a first-in-first-out queue and pops one from the queue for the next iteration. This process is repeated until all vertices of the graph have been visited or a stop condition is satisfied. When realizing BFS on GPU, vertices are visited in a hop-by-hop manner, where neighbors in the queue are fetched in parallel followed by a synchronization operation [39].

Following the similar idea, we develop our RT-based BFS, which is shown in Algorithm 1. The unvisited vertices are explored in iterations, and inside an iteration, neighbors of the vertices visited in the last iteration are traversed. A CPU-side global synchronization is executed to remove redundant neighbors between iterations. The difference from a CUDA-based BFS method is that we need to convert the graph into a BVH (Line 1) and then expand the queue of unvisited vertices using RT cores (Line 8). As a result, the two keys of RT-based BFS are how to construct a BVH representing the graph, and how to issue rays realizing the neighbor visiting.

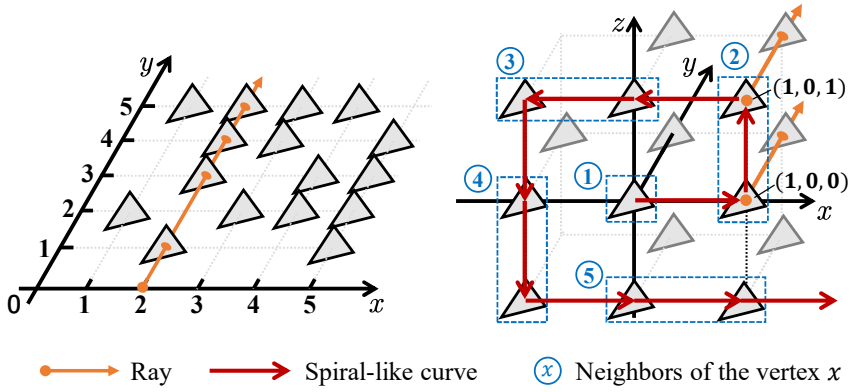

Fig. 4. An example of RT-based BFS. **Left**: Neighbors of a vertex is placed along the y-axis. The vertex 2 has its neighbors of 1, 3, 4, and 5, and we issue a ray at $(2, 0, 0)$ along the y-axis that intersects primitives with the corresponding y-coordinates. **Right**: primitive queues are placed along a spiral-like curve represented as the red arrows, triangle primitives are organized along the y-axis compactly, and multiple queues (in a dashed box) are created for a vertex. We issue two rays at $(1, 0, 0)$ and $(1, 0, 1)$ for visiting neighbors of the vertex 2.

3.1.1 **A naive solution.** A graph could be represented in an xOy plane. A triangle primitive is set at (i, j) if vertex i has a neighbor of j, where i and j are both vertex IDs. So, we have a queue of triangle primitives starting at $(i, 0, 0)$ along the y-axis representing the neighbors of vertex i. The left side of Figure 4 shows an example. The layout of triangle primitives is similar to the adjacency matrix. To retrieve neighbors of the vertex i, we issue a ray at the position of $(i, 0, 0)$ along the y-axis. The maximum length of the ray is N, which is the number of vertices in the graph and this guarantees all neighbors could be touched. For each *any-hit* callback, the y-coordinate of the intersected triangle primitive is one of the neighbor IDs.

However, our experimental evaluation exposes three shortcomings of this method: 1) The coordinates are represented by floating-point numbers in BVH and rays, so this could lead to incorrect results if the triangle primitives are placed in the large-N range as a vertex ID is an integer. 2) Even with hardware acceleration, the intersection tests are still computationally intensive because of long rays. We set the ray length to N to guarantee that all neighbors are visited, but this also greatly increases the time spent on BVH traversal. 3) The workloads of different rays (visiting different neighbor lists) are imbalanced. The more neighbors a vertex has, the more callbacks are executed on CUDA cores, which makes an iteration be throttled by a few rays intersecting the most primitives.

3.1.2 **Optimizations.** To alleviate the shortcomings, we rearrange the primitives and rays as follows: First, we fold the primitive queues from aligning with the x-axis to following a spiral-like curve starting at the origin in the xOz plane.[1] This reduces the maximum coordinate required for setting these queues from N to $\sqrt{N}$. Second, the primitives inside a queue are set compactly instead of placing them according to neighbor IDs. This reduces the length of a ray from N to the actual number of neighbors of a vertex, but this requires an additional table mapping the triangle IDs to the vertex IDs. Third, we impose the maximum length of a primitive queue L_{max} so that a queue of L primitives is separated to $\lceil L/L_{max} \rceil$ queues, which aims at balancing intersections of rays. While multiple queues are created for a vertex, we need another table recording their positions. When searching neighbors of a vertex, we decide how many and where the rays should be issued by

[1]Other space-filling curves, such as the Hilbert curve or the Z-order curve, should also be effective.

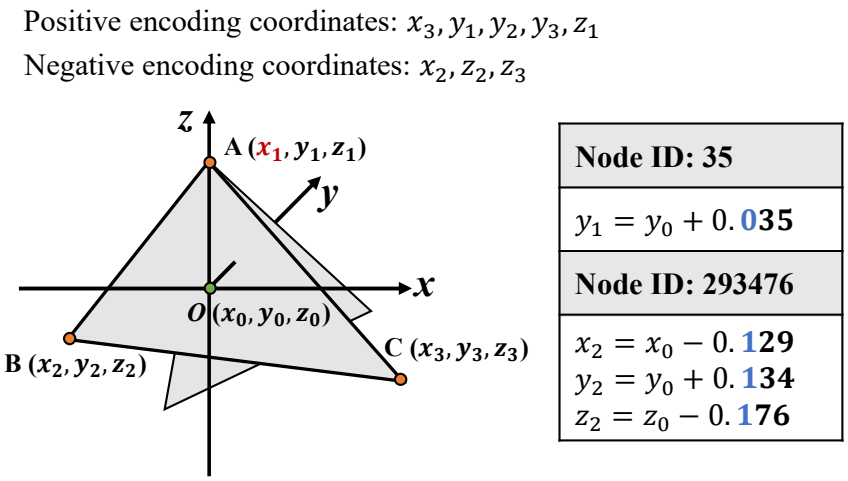

Fig. 5. An example of encoding graph node IDs into triangle coordinates. Two triangles are placed surrounding the ray origin at (x_0, y_0, z_0). A, B, and C are vertices of one triangle. x_1 is fixed to x_0 for ensuring the ray-primitive intersection, and the other coordinates are determined by *positive* or *negative encoding*. The ID 35 is encoded by 1 coordinate while the ID 293,476 is encoded by 3 coordinates.

checking the table, launching multiple rays of length L_{max}, collecting primitive IDs intersected, and translating them to vertex IDs eventually. We show an example of our new index structure on the right side of Figure 4. Since the primitive positions are not directly related to the graph IDs, the graph node reordering technique will not affect the RT-based BFS.

3.1.3 **Vertex encoding.** In the previous design, we use a triangle primitive to represent a neighbor, which is memory-inefficient. To improve this, we attempt to encode graph node IDs into triangle coordinates so that multiple neighbors could be represented by one primitive. When setting a triangle at a position, its vertex coordinates are determined by adding/subtracting the IDs to/from the position coordinates. For example, if we have a triangle at $(1, 0, 0)$ and one of its vertex has the x-coordinate as 1.004, we know that the graph node 4 is a neighbor. Moreover, we overlap multiple triangles at the positions where rays are issued instead of placing them along the y-axis at fixed intervals. This not only greatly reduces the length of rays but also allows us to directly use the ray origins to encode vertex coordinates of triangles.

Although this method is intended to encode 9 IDs into a triangle, the inaccuracy of floating-point representation prevents achieving this ideal outcome. We have to split an ID into multiple parts if it cannot be accurately encoded as a whole. So, additional digits need to be reserved to identify which coordinates belong to an ID. We experimentally find that for each coordinate, using 2 digits to encode the ID and 1 digit as the identifier could guarantee accurate decoding. Additionally, the IDs to be encoded might be small, and we need to fix one of the nine coordinates to ensure the triangle will be intersected by the ray. Figure 5 shows an example of encoding graph node IDs into triangles. Reordering the graph node IDs may change the number of coordinates required to encode IDs in a neighbor list, but it does not affect the total number of required coordinates.

3.2 RT-based Triangle Counting

A triangle is a clique of three vertices, and triangle counting is realized by set intersection: For two neighboring vertices, the number of their common neighbors is the number of triangles containing

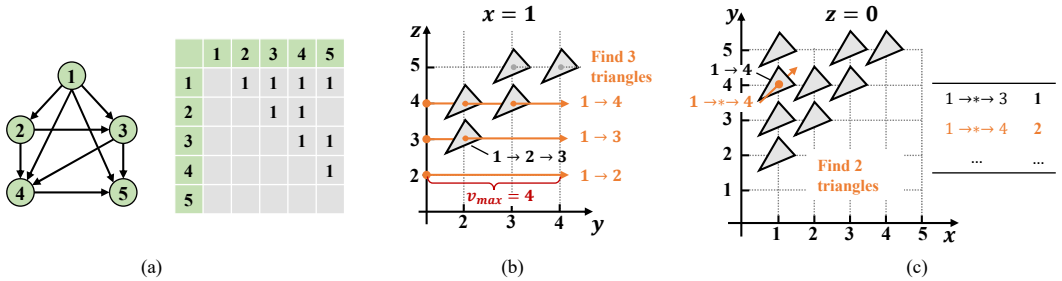

Fig. 6. An example of RT-based triangle counting. **(a)** A loop-free directed graph and its adjacency matrix. **(b) The 1-among-2 method**: This shows a yOz plane with $x = 1$, and the triangle primitives represent 2-hop relations as $1 \rightarrow * \rightarrow *$. For three neighbors of the vertex 1, we issue three rays accordingly, and the ray length is the maximum neighbor ID 4. **(c) The 2-among-1 method**: All 1-hop relations are primitives in the xOy plane with $z = 0$. For a 2-hop relation like $1 \rightarrow * \rightarrow 4$, we issue a ray at $(1, 4, 0)$. Since we have 2 such relations, the triangle count increases by 2.

these two vertices. To avoid redundant counting, a graph must be converted to a loop-free directed version by allowing a vertex to point to only another with a greater ID. Figure 6(a) is an example of such a directed graph with 5 vertices. One either iterates over the edges and counts the common neighbors of the two endpoints, or iterates over the vertices and searches for their 2-hop neighbors among their 1-hop neighbors (or vice versa). In GPU implementations, the counting tasks based on edges or vertices are executed in parallel for acceleration [29–31].

We follow the ideas of searching for 1-hop neighbors among 2-hop neighbors (the *1-among-2* method) and searching for 2-hop neighbors among 1-hop neighbors (the *2-among-1* method) to design two versions of RT-based TC. In the 1-among-2 method, primitives represent 2-hop neighbors and rays are 1-hop neighbors. Conversely, in the 2-among-1 method, the roles of primitives and rays are switched. Next, we illustrate how to build BVH and how to issue rays in the two methods.

3.2.1 **The 1-among-2 method.** Given a 2-hop relation $u \rightarrow v \rightarrow w$, where u, v, and w are vertex IDs, we place a triangle primitive at the position (u, v, w). In a yOz plane of $x = u$, the primitives placed along $z = w$ indicate all 2-hop relations $u \rightarrow * \rightarrow w$. For a 1-hop neighbor v of u, i.e., $u \rightarrow v$, by issuing a ray at $(u, 0, v)$ along the y-axis, each ray-primitive intersection indicates a triangle containing u and v, and the third vertex is the y-coordinate of the intersected primitive. Setting the ray length to v_{max}, which is the maximum ID of 1-hop neighbors of u, guarantees that all triangles can be found. We show an example of our 1-among-2 method applied on a loop-free directed graph in Figure 6(b). Since the BVH traversal is to match the coordinate of a ray (1-hop neighbor) to the coordinates of primitives (2-hop neighbors), our 1-among-2 method is basically the same as searching for 1-hop neighbors among 2-hop neighbors except that a hardware-accelerated index structure is used.

But in practice, v_{max} could be large, leading to long rays and thus inferior performance. Following the similar optimization methods discussed in Section 3.1.2, we could rearrange the primitives along the y-axis compactly and add a table that maps primitive IDs to vertex IDs. This shortens the ray length from v_{max} to the number of neighbors of u. Reordering the graph node IDs may change the distribution of primitives and rays along the z-axis. By arranging them more densely, it is expected to improve the affinity of rays executed in a warp, leading to higher performance [64]. To summarize, the number of rays issued is $|E|$ equaling the number of 1-hop neighbors in the graph. The number of triangles is equal to the number of 2-hop neighbors which is $\sum_{u \in V} \sum_{v \in N(u)} \text{degree}(v)$.

Table 1. Graph datasets of BFS

Dataset	Nodes	Edges	Avg. Degree	Max Degree
hollywood-2009	1.1M	113.9M	99.9	11.4K
kron_g500-logn21	2.1M	182.1M	86.8	213.9K
soc-LiveJournal1	4.8M	70M	14.2	20.3K
soc-orkut	3M	212.7M	71	27.5K
soc-twitter-2010	21.3M	530.1M	24.9	698.1K
road_usa	23.9M	57.7M	2.4	9

3.2.2 **The 2-among-1 method.** In the 2-among-1 method, the primitives are used to represent 1-hop neighboring relations. For a vertex u and its neighbor v, a triangle primitive is placed at $(u, v, 0)$ so that all 1-hop relations are established in the xOy plane with $z = 0$. Given a two-hop relation as $u \rightarrow v \rightarrow w$, we could issue a very short ray at the position $(u, w, 0)$. A ray-primitive intersection means that w is both the 1-hop neighbor and the 2-hop neighbor of u, thus a triangle is found. An example of the RT-based 2-among-1 method is shown in Figure 6(c). Reordering the graph node IDs changes the distribution of primitives on the xOy plane. An more even distribution is preferred since it is expected to enhance pruning efficiency during BVH traversal. In this method, the 2-hop relations are discovered using an individual kernel executed on the CUDA cores, and the RT cores are used to search for the 2-hop neighbors among 1-hop neighbors, which are represented by primitives and are organized as a BVH tree.

Since the roles of primitives and rays are switched compared to the 1-among-2 method, we need to build $|E|$ primitives and issue $\sum_{u \in V} \sum_{v \in N(u)} \text{degree}(v)$ rays. It is not hard to notice that the rays issued for the relations $u \rightarrow * \rightarrow w$ are the same, whose origins are all $(u, w, 0)$. So, we could count the number of $u \rightarrow * \rightarrow w$ before the RT pipeline. As long as an intersection occurs, we add the amounts of the relations to the final triangle count. Figure 6(c) shows an example of 1 intersection resulting in 2 triangle counting. It is worth noting that the 1-among-2 method could be adjusted to follow the similar idea. Specifically, we can set a primitive at $(u, w, 0)$ for the 2-hop relation $u \rightarrow * \rightarrow w$, record the number of 2-hop relations, and issue a ray at $(u, w, 0)$ for the 1-hop relation $u \rightarrow w$. However, this results in both methods sharing the same execution path, making them indistinguishable.

4 Experimental Evaluation

In this section, we conduct extensive experiments to evaluate the graph algorithms designed for RT cores, i.e., RT-based BFS and RT-based TC. All experiments are performed on two machines. One is an Ubuntu 20.04 system with an Intel Xeon Gold 6226R CPU and an NVIDIA GeForce RTX 3090 GPU. This GPU has 82 SMs, 10,496 CUDA Cores, 24 GB of GDDR6X memory, and 82 the second-generation RT cores [9]. The other is an Ubuntu 22.04 system with an Intel Xeon Platinum 8352V CPU and an NVIDIA GeForce RTX 4090 GPU. This GPU has 128 SMs, 16,384 CUDA cores, 24 GB of GDDR6X memory, and 128 the third-generation RT cores [10].

4.1 RT-based BFS

4.1.1 Datasets. We use six real datasets that have been widely used in prior GPU graph processing frameworks [4, 57, 58]. *hollywood, kron, LiveJournal1*, and *road_usa* are obtained from the SuiteSparse Matrix Collection [14]. *orkut* and *twitter* are from the Network Repository [49]. The details of the datasets are shown in Table 1.

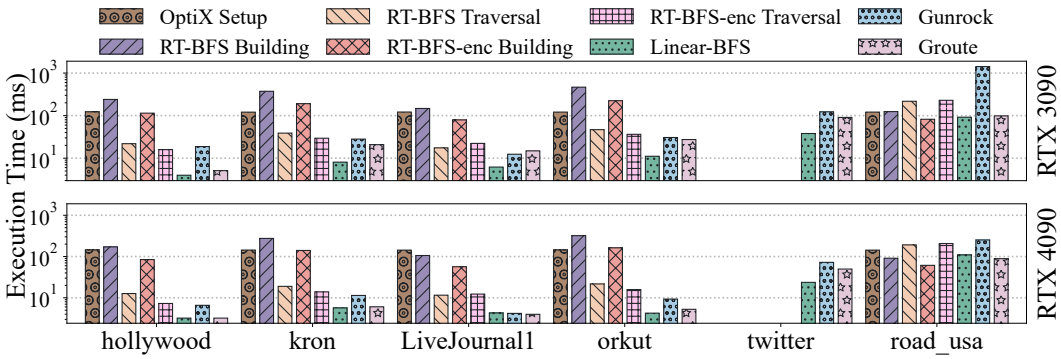

Fig. 7. Execution time of BFS methods on various datasets.

4.1.2 Baselines. We evaluate our RT-based BFS with and without the encoding technique described in Section 3.1.3, which are labeled *RT-BFS* and *RT-BFS-enc*, respectively. We also evaluate the BFS implementations in two GPU graph processing frameworks. *Gunrock* [58] is a bulk-synchronous graph library, which uses data-centric abstraction focused on operations on vertex or edge frontiers. *Groute* [4] is an asynchronous multi-GPU programming model. To better analyze the effect of RT cores in RT-BFS, we include a CUDA-based BFS, namely *Linear-BFS* [39]. Linear-BFS shares similar processing steps shown in Algorithms 1. The only difference is that in each iteration, Linear-BFS accomplishes the parallel expansion on CUDA cores while RT-BFS and RT-BFS-enc use RT cores.

4.1.3 Results. The performance of different BFS methods is presented in Figure 7, where the x-axis represents different datasets and the y-axis is the execution time in milliseconds. For RT-based methods, in addition to the traversal time, we also measure the time used to initialize OptiX and the time used to build BVH. Before the execution, we break long adjacency lists into fixed segments for load balance, and this benefits RT-BFS, RT-BFS-enc, and Linear-BFS. Gunrock and Groute arrange different numbers of threads to process vertices with different degrees for load balancing. We fail to execute RT-BFS and RT-BFS-enc on *twitter* because their BVHs are 31.94 GB and 17.38 GB, surpassing the available GPU memory of our testbed. The results show that RT-BFS performs worse than all other three schemes except Gunrock on the dataset of *road_usa*. The reason is that this dataset has a large diameter, incurring a high overhead of launching kernels in Gunrock. RT-BFS-enc further shortens the execution time compared to RT-BFS. It outperforms Gunrock in both *hollywood* and *road_usa*. On RTX 4090, similar performance trends are observed while all methods execute faster.

By analyzing the difference between RT-BFS and Linear-BFS, we can gain insights into why RT-BFS exhibits such inferior performance. As the adjacency matrix in our experiments is represented in the compressed sparse row (CSR) format, Linear-BFS needs to read the offset first and the vertex ID to get a neighbor. Then, the vertex ID is written to the unvisited queue of BFS. In RT-BFS, one ray-primitive intersection is required to get a neighbor. In such a case, a triangle primitive, which contains a primitive ID and three 3-dimensional coordinates, has to be read from the global memory (an `int32` and nine `float32`). The ray IDs and the upper structure of BVH are shared by multiple leaf nodes, further increasing the reading overhead per intersection test. Once we get the primitive ID by intersection, one more access is required to retrieve the vertex ID according to the optimization in Section 3.1.2. Due to its one order of magnitude more memory accesses, RT-BFS has an average traversal speed that is 3.94 times lower than Linear-BFS. To confirm this, we measure the number of instructions issued to the global memory by RT-BFS and Linear-BFS, and the results

Table 2. Executed SASS load/store instructions and memory footprints on RTX 3090

Dataset	Load/Store Instructions			Memory Footprint (GB)		
	RT-BFS	RT-BFS-enc	Linear-BFS	RT-BFS	RT-BFS-enc	Linear-BFS
hollywood	54.14M	20.55M	12.55M	11.84	5.35	0.67
kron	102.15M	39.84M	23.67M	11.69	9.04	1.38
LiveJournal1	46.03M	22.39M	11.71M	7.22	3.58	0.59
orkut	123.07M	48.05M	24.93M	13.13	9.98	1.1
twitter	N/A	N/A	77.88M	N/A	N/A	3.28
road_usa	98.21M	62.49M	36.87M	7.35	3.98	1.9

are reported in Table 2. We can find that RT-BFS issues 4.03 times more memory instructions on average, which matches the performance gap closely. Table 2 shows that the memory instructions of RT-BFS-enc almost halve compared to RT-BFS, which matches that on average 1.906 IDs are encoded into a triangle primitive. But this does not lead to half execution time in Figure 7. The decoding operations have offset the gains of reducing memory I/Os. In *LiveJournal1* and *road_usa*, we even get worse running time because the memory instructions are reduced by only 43.8% and 29.9%, respectively.

Additionally, we measure the time of initializing OptiX and building BVH. It is observed that the cost of OptiX initialization is fixed and independent of datasets. The construction time of BVH is related to the number of primitives, which is equal to the number of edges in a graph. The more edges the graph has, the more time it takes to build the BVH. The time used to build BVH is more than that used to traverse the graph except *road_usa*, and it becomes a new bottleneck of the RT-based BFS. For traversal time, it increases as the graph size grows and is also affected by the graph structure. For example, the *road_usa* dataset has a long diameter but low vertex degrees. So RT-BFS has to launch a few rays each round for a large number of rounds, which underutilizes the GPU resources. While RT-BFS-enc further reduces the number of rays by encoding multiple IDs into a triangle, its performance is worse in such low-degree graphs. This is another reason RT-BFS-enc is slower than RT-BFS on *LiveJournal1* and *road_usa*.

We measure memory footprints of different BFS methods using NVIDIA Nsight Systems, and report the results in Table 2, where RT-based methods consumes an order of magnitude higher memory than Linear-BFS. As for RT-based methods, RT-BFS-enc roughly halves the memory footprints due to that more than one IDs are encoded in a primitive.

4.1.4 Key takeaways. For BFS, whose dominant operations are visiting vertices sequentially, the RT-based method can hardly accelerate it because the memory accessing overhead of BVH traversal is one order of magnitude higher than the classic CUDA-based method even with our encoding technique. Furthermore, building a BVH that represents a graph takes time more than that used to traverse it. Both the 2nd and the 3rd generation RT cores show similar performance trends.

4.2 RT-based Triangle Counting

4.2.1 Datasets. We use seven datasets from SNAP [33] for experimental evaluation. Table 3 shows the statistics of the datasets. The names of these datasets are abbreviated as *dblp*, *youtube*, *patents*, *wiki*, *com-lj*, *ljournal*, and *orkut* in the following discussion. We focus on the smaller datasets since the high memory footprints of RT-based TC algorithms.

Table 3. Graph datasets of triangle counting

Dataset	Nodes	Edges	Triangles
com-dblp	317,080	1,049,866	2,224,385
com-youtube	1,134,890	2,987,624	3,056,386
cit-Patents	3,774,768	16,518,948	7,515,023
wiki-Talk	2,394,385	5,021,410	9,203,519
com-lj	3,997,962	34,681,189	177,820,130
soc-LiveJournal1	4,847,571	68,993,773	285,730,264
com-orkut	3,072,441	117,185,083	627,584,181

4.2.2 Baselines. We use *RT-1A2* and *RT-2A1* to represent our 1-among-2 and 2-among-1 RT-based triangle counting methods in the experiments, respectively. In addition, we select four CUDA-based triangle counting methods as baselines. *Fox* [18] uses a fine-grained load balancing for list intersection and enters the Graph Challenge 2018 finalists. *TriCore* [31] designs a binary search-based parallel algorithm that assigns a warp to each edge and caches the first k levels of the binary search tree in shared memory. *Hu* [29] proposes a fine-grained vertex-based algorithm that uses binary search with shared memory optimization. *H-Index* [46] is a hash-based GPU algorithm and wins the Graph Challenge 2019 champion. *GraphBLAST* [60] is the GPU version of GraphBLAS [1], which realizes the general sparse matrix-matrix multiplication-based (SpGEMM-based) triangle counting.

4.2.3 Results. The comparison results are shown in Figure 8(a), where we only measure the execution time of traversing BVH in RT-based methods. For other baselines, we only measure the time of executing the triangle counting kernel. The experimental results show that RT-1A2 outperforms all baselines on *com-dblp*, *com-youtube*, *cit-Patents*, and *wiki-Talk*. But the larger graphs of *com-lj*, *soc-LiveJournal1*, and *com-orkut* fail it. The reason is that RT-1A2 has to build all 2-hop relations as primitives into the BVH, which is prohibitively high to the GPU memory. The similar results are also found on the RTX 4090 GPU while all methods are faster. *GraphBLAST* cannot be executed on our RTX 4090 machine because it requires a lower version of CUDA than the lowest one supported by Ubuntu 22.04.

The superior performance of RT-1A2 could result from one of two reasons: 1) The ray-based parallel strategy RT-1A2 is more load-balancing than the baselines, or 2) the hardware-accelerated BVH is more efficient in finding target vertices than non-RT methods. To verify this, we realize a method named BS-1A2, which follows the same process of counting triangles of RT-1A2 but does not use RT cores. Instead, we search for the 1-hop neighbors among 2-hop neighbors with a binary search on CUDA cores. The results are shown in Figure 8(b). We can observe that BS-1A2 outperforms RT-1A2 on all datasets, and moreover, BS-1A2 has a smaller memory footprint so that it can be run on large datasets that RT-1A2 cannot. These results suggest that the superior performance of RT-1A2 results from a more load-balancing task arrangement instead of using RT cores to accelerate the search.

RT-2A1 is a more memory-efficient alternative to triangle counting algorithms on RT cores because 1-hop relations are less than 2-hop relations in most realistic graph datasets. From Figure 8(a), RT-2A1 is slower than RT-1A2 in most cases, but it could run on all datasets except *com-orkut*. *com-orkut* fails RT-2A1 because it launches too many rays. RT-2A1 outperforms other baselines only on *com-dblp*. As the graph scale grows, RT-2A1 becomes slower. On *soc-LiveJournal1*, it only

Table 4. Energy-delay product in $W \cdot ms^2$ of triangle counting methods on RTX 4090

Dataset	RT-1A2	RT-2A1	Fox	Hu	TriCore	H-Index
com-dblp	5.51	2.13	646.42	30.09	8579.36	6.47
com-youtube	38.61	158.57	466.58	185.34	12387.07	32.00
cit-Patents	629.69	1153.18	1442.22	3394.16	65469.69	560.00
wiki-Talk	424.96	1689.84	667.19	155.68	19920.08	317.10
com-lj	N/A	57464.42	14568.89	20865.80	1.73×10^5	7246.86
soc-LiveJournal1	N/A	1.07×10^5	35540.99	44366.35	3.40×10^5	14537.32
com-orkut	N/A	N/A	2.10×10^6	5.70×10^5	2.83×10^6	1.05×10^6

outperforms *TriCore*. By looking into the execution details of RT-2A1, we believe the poor performance of RT-2A1 results from its high miss ratio of rays. The miss ratio means the proportion of rays hitting no primitives, i.e., calling the *miss* callback. Only the rays hitting primitives effectively count triangles in RT-based methods while conventional TC algorithms have no such metric. Taking the experiments on RTX 3090 as an example, the average miss ratio of RT-2A1 is 94.08% while RT-1A2 has only 59.94%. On *com-dblp*, the only dataset RT-2A1 gains superior performance, RT-2A1 reaches the miss ratio of 81.59%. As a result, RT-2A1 can hardly accelerate triangle counting on large graphs, even if it has a low memory footprint.

RT-2A1 exhibits good performance on small-scale datasets, but the reason is not its low miss ratio because RT-1A2 has an even lower miss ratio of 37.58% (on RTX 3090). We are interested in that if this performance gain is from our designs on how to construct the BVH and how to issue rays. To understand this, we still design a triangle counting algorithm BS-2A1 following the same computation pipeline of RT-2A1. Instead of using BVH to determine whether a 2-hop neighbor is also a 1-hop neighbor, BS-2A1 uses binary search on CUDA cores to accomplish the task. The experimental results are presented in Figure 8(b). BS-2A1 significantly outperforms RT-2A1 on all datasets. This is consistent with the comparison between RT-1A2 and BS-1A2. Both results indicate that binary search on CUDA cores is a more efficient alternative to BVH traversal on RT cores for triangle counting.

We also test how long the BVH construction takes in two RT-based algorithms. We show the BVH construction time of RT-1A2 and RT-2A1 in Figure 8(b), alongside the time used to count triangles on RT cores. The results show that the construction time of both methods is less than the computation time, which is the opposite of BFS. Thus, the BVH construction is no longer a bottleneck in RT-based triangle counting methods.

Table 4 shows energy-delay product (EDP) in $W \cdot ms^2$ measured for various triangle counting methods on RTX 4090. We notice that RT-based algorithm is energy efficient on small datasets like *com-dblp* and *com-youtube*. While on large datasets like *wiki* and *com-lj*, RT-based methods consume more energy than CUDA-based ones except TriCore, which is always the most energy-inefficient among all methods. Table 5 reports memory footprints of different triangle counting methods, where RT-based methods still use one order of magnitude higher memory than the other methods except TriCore and H-Index. TriCore allocates a fixed piece of memory to each GPU thread for holding intermediate results in addition to the graph data. H-Index has to use a large size of memory as hashing buckets. The power and memory footprints of programs are measured using NVIDIA Nsight Systems [13].

4.2.4 Key takeaways. Both RT-based TC methods we have designed exhibit superior performance on small datasets on both RTX 3090 and RTX 4090. In large datasets, the 1-among-2 method fails the execution because of its high memory footprint, while the 2-among-1 method is slow because

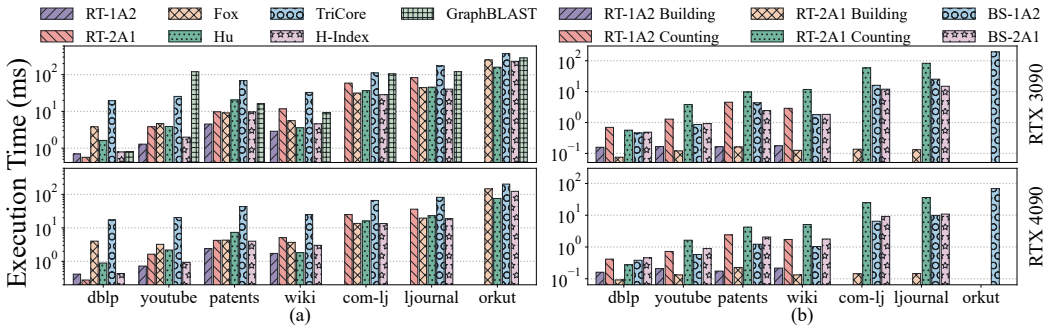

Fig. 8. The execution time of triangle counting methods on different GPU platforms: **(a)** The execution time of kernels used to count triangles in different methods. **(b)** The comparison between BVH traversal-based and binary search-based triangle counting methods.

Table 5. Memory footprints (GB) of triangle counting methods

Dataset	RT-1A2	RT-2A1	Fox	Hu	TriCore	H-Index	GraphBLAST
com-dblp	0.51	0.14	0.01	0.51	4.01	3.92	0.04
com-youtube	3.79	0.54	0.03	0.52	4.04	3.95	0.11
cit-Patents	8.25	2.19	0.12	0.59	4.23	4.12	0.56
wiki-Talk	15.76	1.42	0.05	0.52	4.07	3.98	0.12
com-lj	N/A	8.91	0.19	0.66	4.42	4.32	1.19
soc-LiveJournal1	N/A	11.95	0.35	0.70	4.52	4.42	1.62
com-orkut	N/A	N/A	0.99	0.97	5.33	5.24	3.95

of its high miss ratio of rays. Our in-depth analysis reveals that the performance improvement of RT-based methods stems not from hardware acceleration but from the fine-grained separation of workloads, and it can be replaced by binary search on CUDA cores for even higher performance. As the computation is more intensive on RT cores, the BVH construction is no longer the bottleneck of RT-based methods.

4.3 Set Intersection

In the previous experiments of evaluating TC methods, we notice that RT cores exhibit excellent performance for set intersections (discovering the common elements in two neighboring lists). Set intersections are widely used in graph algorithms like triangle counting, clique detection, and subgraph matching. To avoid our prior evaluation being limited to a small scope of datasets, we plan to more comprehensively explore whether this task could be accelerated by RT cores in synthesized workloads.

4.3.1 Method description. The execution of RT-based set intersection is illustrated in Figure 9, which we name as *RT-Set*. We perform set intersection between a single set $A = \{a_1, \cdots, a_n\}$ and m independent sets $\mathcal{B} = \{B_1, \cdots, B_m\}$. The final results are denoted by $C = \{C_1, \cdots, C_m\}$. Note that there are no duplicate elements in any set.

We convert each element of sets in $\mathcal{B}$ into a triangle primitive and each element in A into a ray. In an xOz plane, we assign a unique x-coordinate to a set in $\mathcal{B}$. Then, we place primitives perpendicular to the xOz plane, with its z-coordinate equal to the element value. After building this

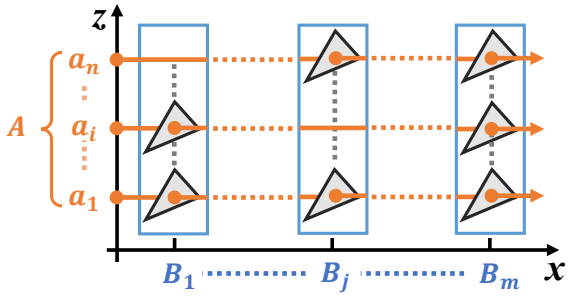

Fig. 9. The illustration of RT-based set intersection.

BVH, we launch rays representing elements in A for executing set intersection on different sets in $\mathcal{B}$ in parallel. The x-coordinate of ray origins is one smaller than B_1 and the z-coordinate is equal to the element value. The length of rays is m, and the direction is along the x-axis. A ray-primitive intersection at B_j means a common element is found between A and B_j. The size of C_j is equal to the number of intersections happening on the x-coordinate at B_j. The number of rays is $|A|$ and the number of primitives is $\sum_{1 \le j \le m} |B_j|$.

4.3.2 Baselines and experimental setup. We select five set intersection methods based on different strategies as baselines. *Intersect Path* [23]: It is a set intersection algorithm based on the GPU merge path algorithm [22]. It partitions the workload into subsets first and then carries out subset intersection in parallel. *Hash* [46]: It uses the shorter set to construct buckets and then probes the buckets with elements in other sets. Inside a bucket, a linear search is performed. *Bitmap* [3, 6]: It marks the occurrence of an element in each set with a bit. The intersection is obtained by checking which bits are still set after bitwise AND operations. Two variants are used in our experiment, i.e., *Naive Bitmap* and *Dynamic Bitmap*. Naive Bitmap means that the bitmaps are constructed before the intersection and are stored in the global memory. Dynamic Bitmap constructs bitmaps on the fly. *Binary Search*: We use the binary search in TriCore [31] for triangle counting, which is divided into two phases. It looks up the top levels of the binary search tree in shared memory and then looks up the remaining levels in global memory.

We evaluate RT-Set and other baselines in two scenarios: (1) intersection between two sets, and (2) intersection between a set and multiple sets. The intersection between multi-sets could be trivially transformed from scenario (2). In the experiments, we use synthetic datasets by manipulating the characteristics of sets. All datasets are by default generated with a uniform distribution. To saturate SMs on the GPU, we set 1K copies of A in the memory, and launch the same number of thread blocks to execute set intersection between A and $\mathcal{B}$ in non-RT baselines.

4.3.3 Intersection between two sets. We analyze the performance of RT-Set by varying selectivity, skew ratio, set size, and density. The results are shown in Figure 10. The selectivity is defined as $\frac{|C|}{\min(|A|,|B|)}$, which denotes the proportion of the result to the original set. The skew ratio is defined as $\frac{|B|}{|A|}$, representing the difference between two input sets. We first fix the size of A to 0.1M, the skew ratio to 1, and vary the selectivity from 0 to 1. We can get the following observations from Figure 10(a). The execution time of RT-Set and Binary Search both increase as the selectivity approaches 1, and RT-Set grows faster. In RT-Set, the larger the selectivity is, the more ray-primitive intersections occur, leading to more callbacks to the *any-hit* function executed on CUDA cores. According to the research of Han et al. [26], the proportion of low selectivity (≤ 0.3) set intersections in triangle counting, clique detection, and subgraph matching exceeds 90%. Binary Search achieves

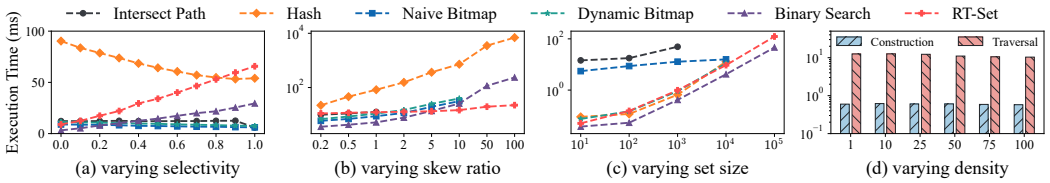

Fig. 10. Performance of various methods executing pairwise intersections.

better performance than RT-Set for all selectivity in our experiments. Thus, from the view of selectivity, there is no advantage towards accelerating set intersections in graph algorithms with RT cores.

We also experiment to understand how the skew ratio impacts the performance of various algorithms, and the results are shown in Figure 10(b). We fix the size of set A to 0.1M, the selectivity to 0.1, and change $|B|$ from 20K to 10M. As the skew ratio increases, the execution time of RT-Set remains stable while for other methods it keeps increasing. When the skew ratio is greater than 5, RT-Set starts to outperform other methods. This indicates that continuously increasing the size of the BVH has little impact on the performance of RT-Set when the number of rays and the number of ray-primitive intersections remain constant. For other methods, as the size of B grows, more data are read to the SM, thus their execution time is also increasing. The greater the skew ratio is, the greater the advantage RT-Set has.

In the next experiment, we vary the set size, and the result is illustrated in Figure 10(c). We fix the selectivity to 0.1, the skew ratio to 1.25, and vary the size of set A from 10 to 100K. We set the number of sets A to 10K instead of 1K configured in other experiments. Except for RT-Set and Binary Search, other baselines all fail to execute set intersection if the size of A is 100K, while Intersect Path also fails on A having 10K elements. From the figure, we observe that Binary Search outperforms the other methods across all set sizes. Additionally, RT-Set, Hash, and Dynamic Bitmap exhibit nearly identical performance. All four methods share the same growth rate. Given that the degrees of most vertices in the graph follow a power-law distribution, RT-Set is not suitable for acceleration.

Finally, since the position of a primitive is correlated to the corresponding element value in a set, we study whether the spatial closeness of primitives, or density, affects the performance of RT-Set. We quantify the density using the parameter λ in the probability density function (PDF) following an exponential distribution $\lambda e^{-\lambda x}$ to represent the density of a dataset we generate with the same PDF. For a dataset, the larger λ is, the more tightly the primitives are arranged. We fix the size of set A to 0.1M, the selectivity to 0.1, the skew ratio to 1, and vary the λ from 1 to 100. The result is presented in Figure 10(d). As the density increases, the construction time remains almost constant, while the traversal time slightly decreases. This indicates that higher density enhances RT-Set performance for pairwise set intersection. We suppose the difference results from how OptiX builds the BVH. However, in this experimental setup, RT-Set is still slower than Binary Search, which is 4.43 ms.

4.3.4 Intersection between a set and multiple sets. Next, we execute the intersection between a set and multiple sets with RT-Set. This is the set intersection form used in our RT-based triangle counting. We vary the number of sets B, selectivity, skew ratio, and density. All sets B in the experiments have the same size and the results are shown in Figure 11.

First, we vary the number of sets B from 1 to 0.1M and fix the size of set A to 10K, the skew ratio to 1, and the selectivity to 0.1. In Figure 11(a), the execution time of all methods increases at

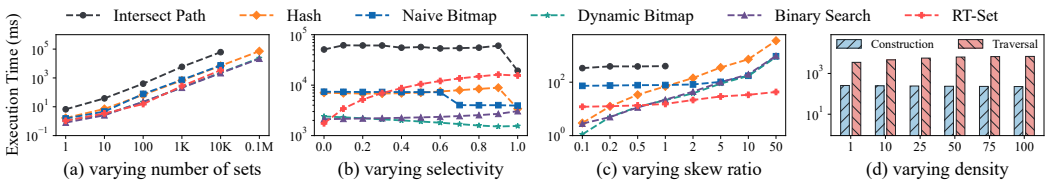

Fig. 11. Performance of various methods executing intersections between a set and multiple sets.

almost the same speed. The execution time of RT-Set, Binary Search, and Dynamic Bitmap achieve a similar execution time, which is lower than the other three methods. However, RT-Set fails to be executed in the case that the number of B is 0.1M because the memory size cannot store the BVH.

Figure 11(b) shows the impact of selectivity on the execution time of various methods. We fix the size of set A to 10K, the number of sets B to 10K, the skew ratio to 1, and change the selectivity from 0 to 1. Similar to Figure 10(a), RT-Set is mostly impaired by a growing selectivity, and the execution time of other methods increases more slowly. Because of more sets being processed, a larger performance gap between RT-Set and the other methods is discovered than that in Figure 10(a). Therefore, a low selectivity still favors RT-Set in the case of an intersection between a set and multiple sets.

In the experiment with varying skew ratios, we fix the size of set A to 10K, the number of sets B to 100, the selectivity to 0.1, and increase the skew ratio from 0.1 to 50. In Figure 11(c), we can observe a similar result as that in Figure 10(b). Compared to the pairwise set intersection, the advantage of RT-Set is larger in the intersection between one and multiple sets. When the skew ratio is greater than 1.0, RT-Set starts to outperform other methods, while the turning point appears at 5 in pairwise set intersection. This result indicates that RT-Set can work in a larger range of skew ratios when executing a one-to-many set intersection.

We present the time breakdown of RT-Set under different densities in Figure 11(d). The sizes of set A and set B are both 10K, the number of set B is 10K, and the selectivity is 0.1. With increasing density, the construction time of BVH decreases gradually, and the traversal time increases. This is opposite to Figure 10(d). One possible reason is that the increased density and the higher number of sets B increase the overlap between AABBs, leading to higher traversal time.

4.3.5 Key takeaways. For the pairwise set intersection, RT-Set shows better performance when the skew ratio is high. This advantage is further magnified in intersections between a set and multiple sets. When intersecting a set with multiple sets, RT-Set has competitive performance with other methods when the number of sets is increasing. Like that in RT-based TC, the BVH construction time is lower than the BVH traversal time.

5 Analysis of Search Performance on RT Cores

Due to the hardware-accelerated BVH traversal, RT cores show promising performance for search operations as observed in triangle counting and set intersection. A workload with sequential memory accesses only, like BFS, is hardly accelerated by this emerging hardware because reading a primitive requires more memory bandwidth than reading an element that is usually a word on the CUDA core. As for the search operation, it could be both implemented on the RT core and the CUDA core. On the RT core, we expect that a higher memory bandwidth is demanded as well but the search is accelerated by hardware (BVH traversal). On the CUDA core, we access less data but the threads are highly diverged when executing search, which is not friendly to the architecture.

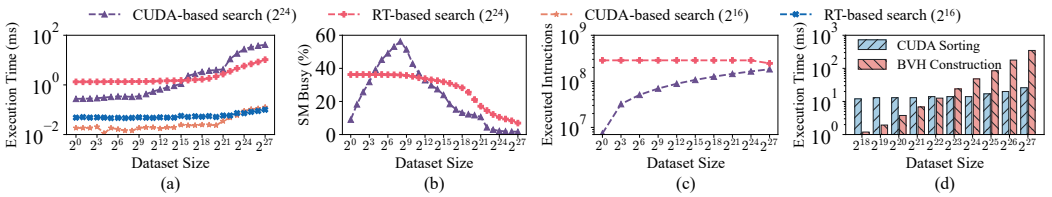

Fig. 12. The experimental results of RT-based search and CUDA-based search. We measure **(a) the execution time**, **(b) the SM busy** in terms of CUDA cores, and **(c) the executed instructions** on CUDA cores of two methods. The preparation overhead of two methods is measured as **(d) the execution time** of BVH construction and CUDA sorting.

To thoroughly investigate the search performance on RT cores and CUDA cores, we realize both RT-based and CUDA-based binary search for experimental comparison. They are marked *RT-based search* and *CUDA-based search* in this section, respectively. In the experiments, both the queries and the dataset are integers. We vary the size of the dataset to be searched, denoted by n, from 2^0 to 2^{27} and set the query set size to be 2^{16} and 2^{24}. While the data to be searched are sorted, the queries are not. For CUDA-based search, we assign a query to a thread that searches if the element exists in an ordered array in the global memory and does not use shared memory. For RT-based search, an element e is built as a triangle primitive, and the primitives are evenly distributed in a cube with its edge of $\sqrt[3]{e_{max}}$, where e_{max} is the maximum value of elements in the dataset. Thus, the primitive representing e is placed at $(e \bmod \sqrt[3]{e_{max}}, e \bmod (\sqrt[3]{e_{max}})^2, e/(\sqrt[3]{e_{max}})^2)$. When searching for a number, a short ray is issued at the corresponding position, and the ray-primitive intersection indicates the search succeeds.

5.1 Results and Analysis

We measure the execution time of two methods on different scales of datasets, and the results are shown in Figure 12(a). With the query set size of 2^{24}, we can observe that RT-based search is worse than CUDA-based search when the dataset size is less than or equal to 2^{15} but is better beyond that. For the query set size of 2^{16}, the turning point at 2^{24}. Since the vanilla binary search and BVH traversal both align with the complexity of $O(\log n)$, this indicates that the search operation is more efficient on RT cores. However, the inferior performance of RT-based search on small datasets, which maintains a nearly flat execution time in that range, implies that the RT pipeline has a non-trivial constant overhead independent of n. Although the CUDA-based method also incurs constant overhead, the turning point shifting to the right in a smaller query set means that it is smaller compared to the RT-based search. This helps answer why RT-based TC can not compete with its counterpart on the CUDA core: we have to build a BVH at the scales of all 1-hop or 2-hop neighbors, but on the other hand, the CUDA-based search is constrained in an adjacency list of a vertex. This means that n differs in the two methods, thus having the performance gap.

5.2 Kernel Profiling

We use NVIDIA Nsight Compute to analyze the two methods when the query set size equals 2^{24}, which is an interactive profiler for NVIDIA GPUs [12]. We measure SM busy and executed instructions, and the results are presented in Figure 12(b) and (c), respectively. Note that Nsight Compute does not report RT cores usage.[2]

[2]https://forums.developer.nvidia.com/t/is-there-a-way-to-measure-rt-core-util/168089

In Figure 12(b), the SM busy of CUDA-based search initially increases and then decreases. This occurs because more memory bandwidth is leveraged to feed data to the SMs as the dataset size grows at the beginning stage. As the dataset size keeps increasing, the memory accesses are more likely irregular, leading to severe under-utilization of memory bandwidth, and the SMs have to wait for data transfer again. For RT-based search, the SM busy exhibits more stability in its trend. Since the search process is transformed to BVH traversal executed on RT cores, RT-based search has lower SM busy in the dataset range that the CUDA-based search has balanced computation and memory accesses (between 2^5 and 2^{16}). The SM busy of RT-based search is as high as $\sim$ 40% when the dataset size is small, so we could conclude that the overhead of executing callback functions, i.e., *ray-generation*, *any-hit*, *closest-hit*, and *miss*, are high. In the range of dataset size greater than 2^{16}, the SM Busy of RT-based search also gradually declines. This suggests that the search workload has been dominated by the BVH traversal, so the CUDA cores have to wait until the RT core finds an intersection (or a miss).

In Figure 12(c), we can observe that the number of instructions executed by RT-based search stays at a high level almost unchanged even if the BVH contains only one primitive. This proves the non-trivial overhead of switching between two types of cores in OptiX pipeline, and only a large enough workload would compensate for it. But traversing a larger BVH does not incur more instructions on CUDA cores, so the overhead stays constant when more primitives are included. On the other hand, the instructions executed by CUDA-based search increase as the dataset size grows. The reason is that more instructions have to be spent on reading and comparing data on CUDA cores.

5.3 BVH Construction *vs.* CUDA Sorting

As a BVH needs to be constructed beforehand in the RT-based search, we have to sort the elements in the CUDA-based search. We adopt `thrust::sort()` in the experiment for sorting the datasets. The execution time of the preparation stage of the two methods is reported in Figure 12(d). Building a BVH for a dataset with elements less than 2^{23} is faster than sorting that amounts of elements. As the scale of dataset grows to 2^{23} or more, the BVH construction becomes slower than CUDA sorting. Therefore, when using RT-based search on large datasets, the potential performance gains from BVH traversal could be offset by the expensive construction time if the preparation time must be a consideration.

6 Discussion

The performance gap of two basic graph algorithms from the expectation indicates that accelerating a general computation task with RT cores is not straightforward on the current commodity GPUs. A few rules are summarized as follows:

Issue close and short rays, and use triangle primitives. Though RT cores execute control flow-intensive search efficiently, the memory and execution divergence are still concerns. When issuing distant rays inside a thread block, more AABBs and primitives must be read than issuing close ones. This is also discussed in the previous work [64], where the ray coherence technique is proposed to alleviate this. Moreover, long rays tend to create more imbalance workloads and straggler threads in warps. Lastly, only the ray-triangle intersection is hardware-accelerated [9, 41] while one has to write customized intersection-test callbacks for other geometric primitives, incurring non-trivial overhead.

Be aware of memory cost and floating-point representation. The fast traversal operation is based on the BVH in the global memory, which is designed to contain geometric objects in the 3-dimensional space using floating-point representations. This data structure might be a waste for many applications. Encoding multiple data to one geometric primitive could alleviate this, but

additional encoding/decoding overhead is introduced. The inherent floating-point representation of primitives makes the encoding operation harder if integers are used in the task. The floating-point representation also causes unbearable errors in intersection tests at large-coordinate areas.

Understand the narrow window of performance gain. The search operation, by being mapped to BVH traversal, is extremely fast on RT cores, but it does not necessarily result in acceleration of the workload. Due to the mixed execution model of RT programs, the overhead of callbacks on CUDA cores could dominate the running time if the data to be searched in is small. On the other hand, one has to take exponentially increasing time to build a BVH, which offsets the performance gain of employing RT cores to search in a massive dataset. As a result, the window of performance gain might be narrow for a specific workload.

The not-so-ideal results in our study are mostly due to that the pipeline is designed for ray tracing only. But we can still foresee a wider adoption of RT cores if 1) a more memory-efficient and general indexing structure is used, 2) more data types (at least integers) are supported, and 3) the complicated ray-tracing pipeline could be circumvented.

7 Related Work

Acceleration with RT cores. The RT core is originally designed for the ray-tracing problem. Recently, there have been studies attempting to harness it for accelerating non-graphics tasks [16, 50, 56, 61]. RTNN converts the nearest neighbor search in low-dimensional space into the intersection between rays and objects within a specified range [64]. TrueKNN performs optimizations on the basis of RTNN and can automatically adjust the search radius to find the k nearest neighbors [43]. RTIndeX indexes data in databases as triangles and transforms queries into rays, and then performs lookup through the hardware-accelerated BVH [27]. JUNO utilizes RT core to accelerate approximate nearest neighbor search in high-dimensional space by leveraging its hardware features for the distance comparison operation [35]. RTScan accelerates index scans in databases by transforming predicate evaluation into ray tracing in three-dimensional space [36]. Arkade achieves k-Nearest Neighbor (kNN) search on RT cores for non-Euclidean distances [37]. RayJoin uses RT Cores to accelerate real-time spatial join and overcomes the bottlenecks of traditional methods [20].

Graph Traversal on GPUs. A lot of works have been dedicated to accelerating graph traversal algorithms on GPUs. Merrill et al. [39] propose a fined-grained parallel BFS with an asymptotically optimal linear workload. Liu et al. [34] devise a GPU BFS system that integrates thread scheduling, workload balancing, and direction optimization to attain high performance. Furthermore, many graph processing frameworks incorporate parallel optimizations for graph traversal algorithms, such as Ligra[53], Galois [44], Medusa [63], CuSha [32], Gunrock [58], Groute [4], Frog [52], Tigr [45], SEP-Graph [57], and GraphBLAST [59].

Triangle Counting on GPUs. Triangle counting is a well-studied graph algorithm. The key of GPU-based triangle counting algorithms is to perform load balancing on its dominant workload set intersection. Polak et al. [48] assign an edge to each thread and then do set intersection using merge. Fox et al. [18] estimate the workload of each edge in advance and allocate threads adaptively. Bisson et al. [6] design four CUDA kernels with different granularity, which implements the set intersection based on bitmap. Hu et al. [29] propose a fine-grained vertex-based algorithm in which each thread spots a 2-hop neighbor using binary search. H-Index [46] is a hash-based method aiming to overcome the shortcomings of merge-based and binary search-based approaches. GraphBLAST [59] solves triangle counting by leveraging the power of GPU-specific optimizations for sparse matrix multiplication.

8 Conclusion

We explore that if RT cores could accelerate graph algorithms by re-designing BFS and TC. However, the experimental results exhibit inferior performance compared to the CUDA-based ones. By extending the evaluation to set intersection on synthesized workloads, RT cores gain advantages if the skew ratio between query and datasets is high. To understand the performance gap, we compare the RT-based and CUDA-based binary search. The RT cores more efficiently spot an element than CUDA cores, but they also incur a constant and non-trivial overhead from the OptiX execution pipeline. The high BVH construction overhead and high memory footprint further limit the deployment scope of RT cores. Though the acceleration window is narrow in our case study, graph algorithms are still expected to run more efficiently on RT cores if 1) the BVH indexing structure is memory-efficient, 2) integers are supported, and 3) the fixed execution pipeline could be circumvented.

Acknowledgments

We sincerely appreciate our shepherd, Lishan Yang, and the anonymous reviewers for their insightful feedback and constructive suggestions. This work is supported in part by Natural Science Foundation of Shandong Province, China, No. ZR2022ZD02, Joint Key Funds of National Natural Science Foundation of China under Grant U23A20302, China Postdoctoral Science Foundation under Grant numbers 2024M761806, and the U.S. National Science Foundation under grants MRI2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC2310510.

References

[1] 2017. *The GraphBLAS.* Retrieved Oct 16, 2024 from https://graphblas.org/

[2] Aaron Barnes, Fangjia Shen, and Timothy G. Rogers. 2024. Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration. In *2024 57th Annual IEEE/ACM International Symposium on Microarchitecture*.

[3] Christos Bellas and Anastasios Gounaris. 2022. Exploiting GPUs for Fast Intersection of Large Sets. *Information Systems* 108, C (2022).

[4] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 235–248.

[5] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] Mauro Bisson and Massimiliano Fatica. 2017. High Performance Exact Triangle Counting on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 28, 12 (2017), 3501–3510.

[7] John Burgess. 2020. RTX on—The NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44.

[8] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation*. 857–877.

[9] NVIDIA Corporation. 2020. *NVIDIA Ampere GA102 GPU Architecture whitepaper.* Retrieved Dec 25, 2023 from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[10] NVIDIA Corporation. 2023. *NVIDIA Ada GPU Architecture whitepaper.* Retrieved Oct 16, 2024 from https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf

[11] NVIDIA Corporation. 2023. *NVIDIA OptiX 7.5 Programming Guide.* Retrieved Dec 5, 2023 from https://raytracing-docs.nvidia.com/optix7/guide/index.html#preface#preface

[12] NVIDIA Corporation. 2024. *Nsight Compute Documentation.* Retrieved Mar 5, 2024 from https://docs.nvidia.com/nsight-compute/index.html

[13] NVIDIA Corporation. 2025. *Nsight Systems Documentation.* Retrieved Mar 27, 2025 from https://docs.nvidia.com/nsight-systems/index.html

[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011).

[15] Nick Stam Emmett Kilgariff, Henry Moreton and Brandon Bell. 2018. *NVIDIA Turing Architecture In-Depth.* Retrieved Jan 5, 2024 from https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth

[16] I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis. 2021. Fast Radius Search Exploiting Ray Tracing Frameworks. *Journal of Computer Graphics Techniques* 10, 1 (2021), 25–48.

[17] Wenfei Fan. 2012. Graph Pattern Matching Revised for Social Network Analysis. In *Proceedings of the 15th International Conference on Database Theory*. 8–21.

[18] James Fox, Oded Green, Kasimir Gabert, Xiaojing An, and David A. Bader. 2018. Fast and Adaptive List Intersections on the GPU. In *2018 IEEE High Performance Extreme Computing Conference*. 1–7.

[19] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*. 1–6.

[20] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 124–136.

[21] Oded Green, James Fox, Alex Watkins, Alok Tripathy, Kasimir Gabert, Euna Kim, Xiaojing An, Kumar Aatish, and David A. Bader. 2018. Logarithmic Radix Binning and Vectorized Triangle Counting. In *2018 IEEE High Performance Extreme Computing Conference*. 1–7.

[22] Oded Green, Robert McColl, and David A. Bader. 2012. GPU Merge Path: A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*. 331–340.

[23] Oded Green, Pavan Yalamanchili, and Lluís-Miquel Munguía. 2014. Fast Triangle Counting on the GPU. In *Proceedings of the 4th Workshop on Irregular Applications: Architectures and Algorithms*. 1–8.

[24] Sambor Guze. 2014. Graph Theory Approach to Transportation Systems Design and Optimization. *TransNav, the International Journal on Marine Navigation and Safety of Sea Transportation* 8, 4 (2014), 571–578.

[25] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M. Aamodt. 2024. Generalizing Ray Tracing Accelerators for Tree Traversals on GPUs. In *2024 57th Annual IEEE/ACM International Symposium on Microarchitecture*.

[26] Shuo Han, Lei Zou, and Jeffrey Xu Yu. 2018. Speeding Up Set Intersections in Graph Algorithms Using SIMD Instructions. In *Proceedings of the 2018 International Conference on Management of Data*. 1587–1602.

[27] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. In *Proceedings of the VLDB Endowment*. 4268–4281.

[28] Petter Holme and Beom Jun Kim. 2002. Growing Scale-free Networks with Tunable Clustering. *Physical Review E* 65 (2002), 026107.
[29] Lin Hu, Naiqing Guan, and Lei Zou. 2019. Triangle Counting on GPU Using Fine-Grained Task Distribution. In *2019 IEEE 35th International Conference on Data Engineering Workshops*. 225–232.
[30] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. 2017. TriX: Triangle Counting at Extreme Scale. In *2017 IEEE High Performance Extreme Computing Conference*. 1–7.
[31] Yang Hu, Hang Liu, and H. Howie Huang. 2018. TriCore: Parallel Triangle Counting on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 171–182.
[32] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-Centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. 239–252.
[33] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[34] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[35] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 549–565.
[36] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RTScan: Efficient Scan with Ray Tracing Cores. In *Proceedings of the VLDB Endowment*. 1460–1472.
[37] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 14–25.
[38] Enzo Meneses, Cristóbal A. Navarro, Héctor Ferrada, and Felipe A. Quezada. 2024. Accelerating Range Minimum Queries With Ray Tracing Cores. *Future Generation Computer Systems* 157, C (2024), 98–111.
[39] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 117–128.
[40] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Transactions on Parallel Computing* 1, 2 (2015), 1–30.
[41] Keith Morley. 2019. *How to Get Started with OptiX 7*. Retrieved Oct 16, 2024 from https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/
[42] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. In *2023 IEEE International Parallel and Distributed Processing Symposium*. 963–973.
[43] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th ACM International Conference on Supercomputing*. 289–300.
[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 456–471.
[45] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 622–636.
[46] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference*. 1–7.
[47] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using Graph Theory to Analyze Biological Networks. *BioData Mining* 4, 10 (2011).
[48] Adam Polak. 2016. Counting Triangles in Large Graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*. 740–746.
[49] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. 4292–4293.
[50] Justin Salmon and Simon McIntosh-Smith. 2019. Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. 19–29.
[51] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2021. Self-adaptive Graph Traversal on GPUs. In *Proceedings of the 2021 International Conference on Management of Data*. 1558–1570.
[52] Xuanhua Shi, Xuan Luo, Junling Liang, Peng Zhao, Sheng Di, Bingsheng He, and Hai Jin. 2018. Frog: Asynchronous Graph Processing on GPU with Hybrid Coloring Model. *IEEE Transactions on Knowledge and Data Engineering* 30, 1

(2018), 29–42.
[53] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.
[54] Edward H Sussenguth. 1965. A Graph-Theoretic Algorithm for Matching Chemical Structures. *Journal of Chemical Documentation* 5, 1 (1965), 36–43.
[55] Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel. 2018. A Detailed Study of Ray Tracing Performance: Render Time and Energy Cost. *The Visual Computer* 34, 6-8 (2018), 875–885.
[56] Ingo Wald, Will Usher, Nate Morrical, Laura Lediaev, and Valerio Pascucci. 2022. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. In *Proceedings of the Conference on High-Performance Graphics*. 7–13.
[57] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 38–52.
[58] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Transactions on Parallel Computing* 4, 1 (2017), 3:1–3:49.
[59] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Software* 48, 1 (2022).
[60] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Proceedings of the 24th International Conference on Parallel and Distributed Computing*. 672–687.
[61] Stefan Zellmann, Martin Weier, and Ingo Wald. 2020. Accelerating Force-Directed Graph Drawing with RT Cores. In *2020 IEEE Visualization Conference (VIS)*. 96–100.
[62] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. In *Proceedings of the VLDB Endowment*. 340–351.
[63] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.
[64] Yuhao Zhu. 2022. RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 76–89.

Received January 2025; revised April 2025; accepted April 2025