GraphRTX: Lighting the Way to Scalable Graph Analytics

ALEXANDER BAUMSTARK, TU Ilmenau, Germany
KAI-UWE SATTLER, TU Ilmenau, Germany

Modern GPU architectures are powerful platforms for accelerating database processing, including graph analytics. However, due to irregular access patterns in graphs caused by properties such as power-law distribution of vertex degrees, pure SMT execution models can result in resource under-utilization and poor memory coalescing. Modern GPUs address this issue by using heterogeneous accelerators for different types of workload. Among these are ray tracing (RT) cores, which perform hardware-accelerated spatial index traversal, overcoming similar challenges of irregular memory access. By offloading traversal and intersection to fixed-function hardware, RT cores align well with graph traversal patterns. However, there is still no unified graph representation for RT that avoids costly index rebuilding across algorithms. To address this gap, we introduce GraphRTX, a framework that leverages the RT cores of modern NVIDIA GPUs to accelerate graph algorithms. GraphRTX employs a unified graph representation that maps graphs to bounding volume hierarchies (BVHs) and a ray-based query abstraction and enables multiple algorithms to operate on the same model. We also propose a hybrid execution model that combines RT acceleration with GPU thread-level parallelism. Experiments on real-world datasets demonstrate that GraphRTX matches or outperforms state-of-the-art baselines, offering lower BVH (re)build overhead and a smaller memory footprint.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Graphics processors**; • **Information systems** → **Graph-based database models**.

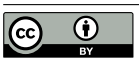
Additional Key Words and Phrases: Graph Analytics, Unified Graph Model, Graph Updates

ACM Reference Format:
Alexander Baumstark and Kai-Uwe Sattler. 2026. GraphRTX: Lighting the Way to Scalable Graph Analytics. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 193 (June 2026), 30 pages. https://doi.org/10.1145/3802070

1 Introduction

Graphs are a fundamental data model for representing relationships in domains such as social networks, knowledge bases, biological systems, and communication infrastructures. As graph datasets grow in scale and complexity, the demand for efficient, scalable graph analytics has increased across data management, scientific computing, and large-scale machine learning pipelines. Typical graph workloads, such as breadth-first search (BFS), single-source shortest path (SSSP), PageRank (PR), and community detection, form the computational backbone of tasks like influence analysis, recommendation, and fraud detection. Given their high throughput and massive parallelism, GPUs have become the preferred platform for accelerating analytical workloads. Graph databases already follow these trends by providing GPU-based acceleration for algorithms, either through their own implementations or via available graph frameworks [42, 66]. These frameworks, such as Gunrock, cuGraph, and GraphBLAST, have demonstrated substantial speedups compared to CPU-based systems [43, 90, 96]. Despite these advances, graph workloads remain challenging to accelerate on GPUs. The irregular and data-dependent access patterns characteristic of power-law graphs lead to poor memory utilization [82], load imbalance [70, 92], and warp underutilization [70].

Authors' Contact Information: Alexander Baumstark, TU Ilmenau, Ilmenau, Germany, alexander.baumstark@tu-ilmenau.de; Kai-Uwe Sattler, TU Ilmenau, Ilmenau, Germany, kus@tu-ilmenau.de.



ACM 2836-6573/2026/6-ART193
https://doi.org/10.1145/3802070

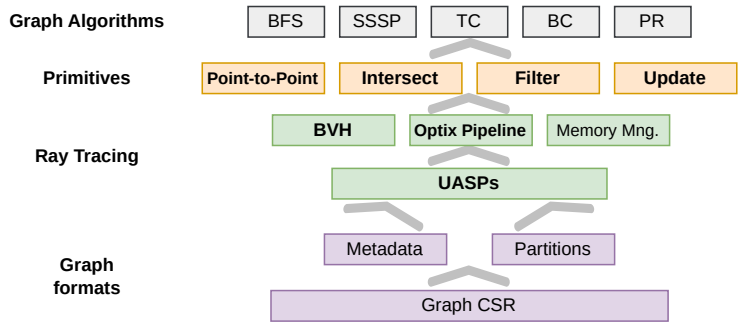


Fig. 1. Overview of the GraphRTX architecture. Key contributions are highlighted in bold.

Many approaches have tried to tackle these challenges through scheduling heuristics and preprocessing. Especially for sparse graphs, there is no general solution. At the same time, GPU architectures themselves are becoming increasingly *heterogeneous* to serve the current requirements, especially driven by machine learning. Modern GPUs combine general-purpose SIMT cores (also referred to as CUDA cores) with specialized accelerators such as *tensor cores* for dense matrix computations and *ray tracing (RT)* cores for spatial traversal [15, 74]. While tensor cores have seen adoption in data management and graph analytics [35, 102], the potential of RT cores remains largely unexplored for data processing, despite a few recent efforts in this direction [32, 84, 94].

Originally designed for real-time photorealistic rendering, RT cores provide highly optimized spatial-index traversal via a bounding volume hierarchy (BVH), which efficiently supports millions of ray–object intersections per frame. While traditional RT once required hours to render a single frame due to slow algorithms [27], dedicated RT cores have reduced this to milliseconds, achieving massive acceleration over conventional shader-based approaches. To achieve this, RT cores perform ray–triangle intersection tests directly in dedicated hardware, leveraging a BVH to prune the search space and drastically reduce the number of required intersection checks. Beyond graphics, these traversal units have demonstrated competitive performance for relational indexing [84], scalar aggregation [84], and spatial query workloads [25]. As newer GPU generations, especially consumer GPUs, continue to increase the number of RT cores (e.g., up to 188 in NVIDIA Blackwell), utilizing these specialized units for data management and graph analytics presents a compelling direction for improving performance and maximizing hardware efficiency. Crucially, RT hardware is designed to accelerate irregular traversal workloads, such as ray–object intersections in complex scenes, which share structural similarities with graph traversals. This observation motivates our exploration of RT-based acceleration for analytical graph workloads. However, applying RT to non-spatial workloads poses several fundamental challenges, as demonstrated in other work [32, 94], which can be summarized as: (1) *Mapping*: Queries and data must be expressed as RT-compatible geometry. (2) *BVH Construction*: Main overhead with limited update support. (3) *Workload Heterogeneity*: Graph workloads include traversals, aggregations, and clustering. (4) *Heterogeneous Execution*: Efficient use requires coordinating SIMT and RT. Moreover, existing approaches are highly specialized, typically designed to accelerate a single workload [94]. As a result, each new problem requires constructing a separate BVH, introducing substantial preprocessing overhead. This specialization severely limits the broader applicability of RT acceleration and poses a major obstacle to its integration into data management systems. As RT cores are becoming a readily available hardware resource in modern GPUs, there is a growing need for a general, reusable framework that can effectively leverage them for data management and graph analytics without incurring repeated preprocessing overhead.

We address this gap with GraphRTX[1], a GPU framework that unifies traditional SIMT and RT-core execution to accelerate graph analytics (cf. Figure 1). At its core, GraphRTX employs a *Unified Adjacency Segment Primitives (UASP)* model, which maps a complete graph to spatial primitives, thereby enabling direct traversal on RT cores while preserving graph-algorithm semantics. Instead of mapping each edge to an individual primitive, such as a triangle, as done in prior work, we group segments of a vertex's neighbor set into a single triangle primitive, enabling graph traversals to be expressed through efficient set intersection operations, where RT can be employed for efficient candidate identification of neighbor sets. Building on UASP, we provide a set of composable primitives for expressing and executing graph queries, from which we implement a range of analytical algorithms. Together with a GPU memory management approach, tailored around the RT-based pipeline, we provide a scalable framework for graph analytics, which is also updateable with low overhead. These all enable providing a framework that can achieve an $2-3\times$ speedup for sparse graphs, while maintaining update efficiency and low preprocessing overhead.

Contributions. Our main contributions are as follows:
(1) Unified execution model: We introduce the UASP model, which provides a unified data and query abstraction for RT-accelerated graph analytics. UASP enables efficient execution across diverse graph workloads, including BFS, SSSP, PageRank, Betweenness Centrality, Triangle Counting, Weakly Connected Components, and Label Propagation.
(2) Efficient BVH and memory management: We design a lightweight BVH construction and GPU memory management scheme for large-scale graphs, featuring a hybrid update strategy that performs refits for small changes and rebuilds for major updates.
(3) SIMT–RT cooperative execution: We develop a cooperative execution framework that jointly utilizes SIMT compute units and RT cores to maximize overall GPU utilization and throughput.
(4) Comprehensive evaluation: We evaluate GraphRTX on multiple real-world graph datasets and compare it against state-of-the-art GPU graph frameworks, analyzing scalability, performance, and hardware efficiency. In addition, we validate our results using the LDBC Graphalytics benchmark.

The remainder of this paper is organized as follows. We first review hardware-accelerated RT and graph analytics. Next, we describe the UASP model, its construction, and detail how graph algorithms are implemented. Finally, we evaluate GraphRTX on diverse graph workloads.

2 Background

2.1 Ray Tracing

Ray tracing is a several-decade-old technique for image synthesis that produces photorealistic images of 3D scenes [27]. The goal is to determine the correct color for each pixel in the picture, aiming for as realistic a result as possible, which is influenced by factors such as camera angle, light sources, object material, and numerous other parameters. The actual technique mimics the process of light in nature: the propagation of light rays through space, with reflection and interaction with objects. Technically, it starts by tracing (multiple) rays from the camera position into the scene and returns all intersections between rays and objects in the scene, which demonstrates the computational complexity. A ray is defined as a parametric line

$$r(t) = \mathbf{o} + t\mathbf{d}, \quad t \in [t_{\min}, t_{\max}],$$

with origin $\mathbf{o} \in \mathbb{R}^3$, direction $\mathbf{d} \in \mathbb{R}^3$, and ray length t. Given a set of geometric primitives $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, ray tracing evaluates intersections between $r(t)$ and the primitives in $\mathcal{P}$. Here, each primitive $P_i \subseteq \mathbb{R}^3$ is a simple geometric object, such as a triangle, sphere, or bounding box, of the form $P_i = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}]$, which defines a bounded region of space.

[1]https://github.com/dbis-ilm/graphrtx

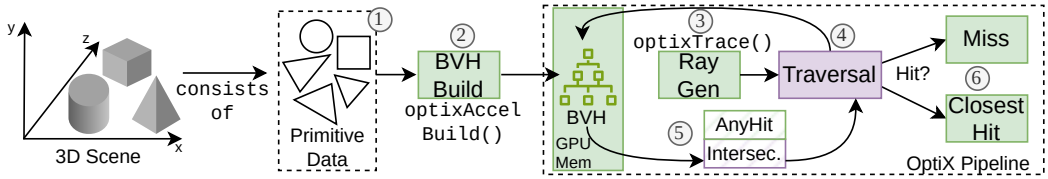


Fig. 2. Optix Ray Tracing Pipeline

Formally, the intersection query is $I(r) = \{ (P_i, t) \mid P_i \in \mathcal{P},\ r(t) \in P_i,\ t \in [t_{\min}, t_{\max}] \}$. The ray tracer can then resolve intersections in several modes:

$$\textbf{AnyHit}: f(I(r)) = \{ g(P_i, t) \mid (P_i, t) \in I(r) \},\ \textbf{ClosestHit}: (P^*, t^*) = \arg \min_{(P_i,t) \in I(r)} t,\ \textbf{Miss}: I(r) = \emptyset$$

where g is a user-defined predicate or transformation applied to each intersection as it is discovered during traversal. `AnyHit` collects all intersections, `ClosestHit` returns the nearest intersection, and `Miss` indicates that no ray intersected with a primitive. Since the test for the intersection between rays and every object in the scene is very costly, objects are organized in bounding volume hierarchies (BVH), which reduces the number of intersections within the scene, and is fundamentally an index. This further enables efficient hardware acceleration by using dedicated RT cores solely for intersection tests.

BVH & Acceleration. The BVH organizes primitives into a tree structure, analogous to an R-tree for spatial indexing, but specialized for efficient traversal rather than dynamic balancing [6]. The leaves of the BVH hold the geometric primitives of the objects, such as points, lines, triangles, or circles. The internal nodes enclose their children with minimal bounding boxes, also known as axis-aligned bounding boxes (AABBs). These AABBs are then enclosed by higher-level parent AABBs until the entire scene is organized into the BVH structure. Instead of testing for intersections with every object in the scene, the BVH is traversed from the root: each node's bounding box is tested, and traversal continues down to the next level until a leaf containing the corresponding primitive is reached. This reduces the need to test the intersection between the ray and each object, increasing performance. On recent NVIDIA GPUs, BVH traversal is accelerated by dedicated RT cores. Each SMT core is paired with an RT co-processor that is dedicated to handling BVH traversal. For instance, the Ada Lovelace architecture integrates up to 128 SMT/RT core pairs, while the Blackwell 2.0 architecture features up to 188 SMT/RT pairs. The RT workflow for ray handling and intersection test is processed within a pipeline.

Pipeline. The pipeline interfaces with NVIDIA OptiX [75], which integrates seamlessly with CUDA programs and provides APIs for ray casting and hardware-accelerated BVH traversal. In this setup, RT cores handle BVH traversal and triangle intersection tests, while CUDA cores execute all remaining stages. As illustrated in Figure 2, the workflow proceeds as follows: (1) geometric primitives (e.g., triangles or custom types) are defined by the host or CUDA kernel; (2) the BVH is built via `optixAccelBuild`, specifying primitive types, acceleration structure, and target buffers; (3) rays with origin, direction, and length are generated on CUDA cores, optionally carrying payloads; (4) traversal and (5) intersections are executed: triangles on RT cores, custom primitives on CUDA cores; (6) intersection triggers AnyHit, ClosestHit, or Miss programs. Optionally, a Shader Binding Table (SBT) enables direct access to additional attributes within the RT pipeline.

Alternative Hardware Architectures. Similar architectures are offered by other GPU vendors such as AMD [64], or Intel [38]. Unlike NVIDIA, their approaches use specialized instructions on SMT cores instead of dedicated RT cores. While our approaches are compatible with this architecture, performance may suffer as RT competes for compute resources.

2.2 Graph Analytics

Model. We define a graph as $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ the set of edges. An optional weight function $w : E \to \mathbb{R}^+$ assigns positive edge weights. Graphs are typically stored as *adjacency lists* or in the *compressed sparse row* (CSR) format, where vertices are indexed, and adjacency lists are stored contiguously. CSRs, widely used in analytical frameworks [83, 90, 96], provide a compact layout with coalesced memory access for fast neighborhood queries $N(u) = \{v \mid (u, v) \in E\}$. It comprises a row offset array `row`[0..|V|], a column index array `col`[0..|E|-1], and optionally a values array `val`[0..|E|-1] for edge weights. This structure minimizes pointer overhead and supports efficient bulk traversal, especially on GPUs where coalesced accesses are crucial. However, CSR favors static graphs, as updates require costly rebuilds; adjacency lists remain preferable for dynamic workloads and can be efficiently converted to CSR when needed [37].

Primitives. We consider three core primitives of graph analytics that form the foundation of many graph algorithms [90]: (1) *Expand:* Traversal primitives that iteratively grow a frontier of active vertices to explore neighborhoods and discover reachable nodes. (2) *Intersection:* Substructure primitives that intersect adjacency lists to identify shared neighborhoods or motifs. (3) *Aggregation:* Message-passing primitives that accumulate and propagate values across neighbors until convergence. Together with basic filtering, these primitives are sufficient to express more complex algorithms discussed below.

► *BFS.* Breadth-first search (BFS) is a fundamental traversal algorithm that discovers the shortest path in terms of hops from a given source vertex to all other vertices in an unweighted graph. In each iteration, BFS expands the current frontier F_t of active vertices by discovering unvisited neighbors. Formally, the next frontier is $F_{t+1} = \bigcup_{u \in F_t} \big(N(u) \setminus V_{\text{visited}}\big)$, where $N(u)$ denotes the neighbor set of u, and V_{visited} the set of already discovered vertices. This can be interpreted as a set intersection between $N(u)$ and the set of unvisited vertices [9, 92]. Thus, BFS expansion can be viewed as repeatedly performing set intersections at the frontier level.

► *SSSP.* The single-source shortest path (SSSP) problem generalizes BFS to weighted graphs, computing the minimum distance from a source vertex s to all others. Each vertex u maintains a tentative distance $d(u)$, initialized as 0 for s and ∞ otherwise. In each iteration, the active frontier F_t contains vertices whose distance values were updated in the previous step. Relaxation examines all edges $(u, v) \in E$ with $u \in F_t$ and updates where $w(u, v)$ is the edge weight. To identify which neighbors v should be relaxed, SSSP intersects the current frontier F_t with each neighbor set $N(u)$, restricted by distance-based predicates. Hence, like BFS, SSSP performs successive neighbor-set intersections to expand the frontier, differing only in the relaxation criterion [14, 87].

► *TC.* The triangle counting (TC) is a subgraph problem where a triangle is a clique comprising 3 vertices (u, v, w) such that $(u, v), (v, w), (w, u) \in E$. A naive approach enumerates all triples of vertices, which is computationally slow. Instead, efficient algorithms reduce triangle counting to neighbor set intersections [88]: (1) For each edge $(u, v) \in E$, compute the intersection $N(u) \cap N(v)$. (2) Each common neighbor $w \in N(u) \cap N(v)$ forms a triangle (u, v, w). (3) Each triangle is discovered once per edge; the total count is divided by 3. Based on TC, the local clustering coefficient (**LCC**) of a node is computed as the ratio of the number of triangles involving the node to the number of possible neighbor pairs, to describe how likely the node's neighbors are to be connected.

► *WCC.* Weakly connected components (WCC) can be formulated using iterative set intersection. Each vertex v maintains a candidate set S_v of vertices that belong to the same component. Initially, S_v contains v and its immediate neighbors. In each iteration, v updates its candidate set by intersecting it with the candidate sets of its neighbors. Iteration continues until convergence, at which point all vertices belonging to the same weakly connected component share an identical set (or a consistent representative). In contrast, vertices in different components converge to disjoint sets.

► *CDLP.* For Community Detection using Label Propagation (CDLP), each vertex v maintains a label ℓ_v and the corresponding neighbor support set, representing the neighbors of v that currently carry label ℓ. In each iteration, v computes the intersection between its neighbor set and the label partitions induced by its neighbors and selects the label whose support set has maximum cardinality. Iteration proceeds until labels stabilize or a fixed point is reached, yielding densely connected vertex groups that correspond to graph communities.
► *PR.* The PageRank (PR) algorithm is typically described as an iterative propagation algorithm in which each vertex u distributes its score to its neighbors. Each iteration calculates the contribution to a vertex. While this is usually seen as a weighted sum over incoming edges, it can also be expressed in terms of set intersections. For each vertex v, the update requires identifying which vertices u contribute to it, i.e., the set of neighbors $N(v)$ with outgoing edges into v. This is equivalent to intersecting $N(u)$ with the global set $\{v\}$ across all $u \in V$, or, operationally, testing membership of v in each neighbor set.
► *BC.* The problem of Betweenness centrality (BC) measures how frequently a vertex v lies on the shortest paths between all source and target pairs. BC computation consists of two traversal phases: a forward phase to identify shortest-path vertices and a backward phase to propagate dependency values. During the forward traversal, each frontier expansion can be expressed as intersecting the current frontier with the sets of unvisited neighbors, analogous to BFS. The backward phase similarly intersects dependency-propagation frontiers with predecessor sets to aggregate contributions [62, 63]. Thus, both phases of BC can be unified under the same neighbor-set intersection primitive that underlies traversal and path aggregation.

This illustrates that many graph algorithms share a common foundation—*set intersection over neighbor sets* and our work motivates: if diverse algorithms reduce to intersection queries, a unified RT-based computational model becomes possible. The next section introduces related work to graph analytics and RT.

3 Related Work

3.1 Graph Analytics

Algorithms. BFS is a fundamental in graph analytics [92], with significant GPU speedups achieved through parallelism, and algorithmic optimizations [10, 21, 29, 55]. TC and LCC, other core algorithms [5], benefit from GPU parallelization [36, 77] and set intersection optimizations [88], further improved by workload balancing and vertex ordering [34]. PR has been accelerated using both SMT and tensor cores [19, 26, 79], while SSSP and BC algorithms also achieve substantial GPU speedups [14, 62, 63, 87]. Graph clustering acceleration on GPUs has also been studied for connected components (SCC, WCC) and for label propagation [45, 54]. We focus on these algorithms due to their support in graph frameworks and standardized benchmarks, such as those by LDBC Graphalytics.
Graph Processing Frameworks. Parallel graph traversal has been extensively studied across CPU, GPU, hybrid, and out-of-core systems—including vertex-centric CPU frameworks [28, 58, 76, 81, 85], GPU frameworks [7, 13, 43, 44, 80, 89, 97, 104], hybrid and tensor-core approaches [16, 103], and out-of-core systems [48, 78, 105]. While many adopt CSR segmentation and partitioning for balance and scalability [7, 44, 80, 90], none exploit RT cores; our approach bridges this gap by mapping graph traversal onto RT hardware using partitioned graphs augmented with hierarchical BVHs.
Formats and Dynamic Graphs. Dynamic graph analytics focuses on evolving graph structures. Systems such as CommonGraph [1], GraphOne [47], and Hornet [11] target dynamic graphs optimized for GPUs. Most graph DBMSs support analytics [17, 69] using CSR-based layouts for compact storage and efficient processing, though CSR updates remain expensive due to costly rebuilds. Recent work improves update efficiency via delta structures [41], LSM-style compaction

[37, 66, 86, 99], or packed memory arrays with insertion gaps [40, 50, 51, 93]. Adjacency lists, while easily updatable, incur higher traversal costs, motivating hybrid designs [20, 42]. In contrast, our approach supports graph updates within a BVH structure, enabling fast traversal by leveraging GPU RT-core spatial acceleration for a unified data and query model.

Memory and Workload Management. Prior work on throughput-oriented architectures [23], runtime systems for efficient data movement and kernel coordination [2, 3, 61], and dynamic warp-level load balancing [53, 72] demonstrates how fine-grained scheduling balances memory and compute utilization.

We adopt these principles to efficiently manage irregular graph workloads on RT cores.

3.2 Hardware-accelerated Ray Tracing

General. Hardware-accelerated RT was originally introduced to accelerate real-time rendering [27]. However, as a dedicated hardware resource, RT cores can be repurposed for other domains since they effectively perform hardware-accelerated index traversal [56, 68, 91, 101, 106]. Recent studies have applied RT cores to nearest-neighbor search via set intersection [56, 68, 106], outlier detection (RTOD) [91], and even matrix multiplication [101], demonstrating use-cases beyond rendering.

DBMS-related. GPUs are now central to modern DBMSs, enabling JIT query compilation, analytical acceleration, and scalable joins [4, 8, 22, 46, 59], with recent work extending RT cores to databases via RTIndex [32], cgRX [33], RTScan [57], RTCUDB [84], RayJoin [24], GPH hashing [12], and LibRTS [25]. Unlike these systems, we generalize RT acceleration to graph analytics through a unified data and query model that maps graphs onto an RT problem.

Spatial Graph Processing. Spatial problems parallel those handled by RT hardware, as both navigate space-defining structures: early GPU approaches such as GPU R-trees [98], spatial indices [18], GPU *k*D-trees [100], and BVHs [49] enabled scalable parallel queries but relied on software-managed traversal. G-PICS [52] and NVIDIA's cuSpatial [71] accelerated joins and kNN.

Trees & Graphs. Recent work has explored RT acceleration for hierarchical and graph-like structures, including RT-Tree for hardware-accelerated tree traversal [30], Arkade for kNN search in non-Euclidean spaces [60], RT-BarnesHut for *n*-body simulations via BVH-mapped particle clusters [67], and graph analytics on RT cores by Xiao et al. [94]. In contrast, we introduce a unified RT-based graph model in which a single geometric abstraction supports multiple algorithms without re-designing data layouts or traversal logic.

Frameworks and Acceleration. RT-accelerated computation has primarily been explored on NVIDIA GPUs via OptiX [75], which provides programmable access to RT cores compatible with CUDA. Similar frameworks exist for other vendors, including AMD's HIPRT [65] and Intel's Embree extension [38], suggesting comparable performance trends across architectures. Beyond RT cores, specialized accelerators such as tensor cores and FPGAs have also been used for graph workloads: TGraph and TCUDB map traversal and aggregation to tensor cores [35, 103], while Graphicionado [31] and HyGCN [95] explore graph accelerators.

► To the best of our knowledge, no prior work provides a unified data and query model for graph analytics for RT hardware.

4 Unified Graph Model for Ray Tracing

We propose a unified abstraction that maps graphs to spatial RT problems by representing vertices and adjacency segments as geometric primitives and expressing algorithmic steps as ray traversals. The model runs on RT cores for hardware acceleration and SIMT cores for portability, enabling efficient execution of diverse graph algorithms within a single framework and bridging graph analytics with RT to exploit spatial parallelism and traversal efficiency.

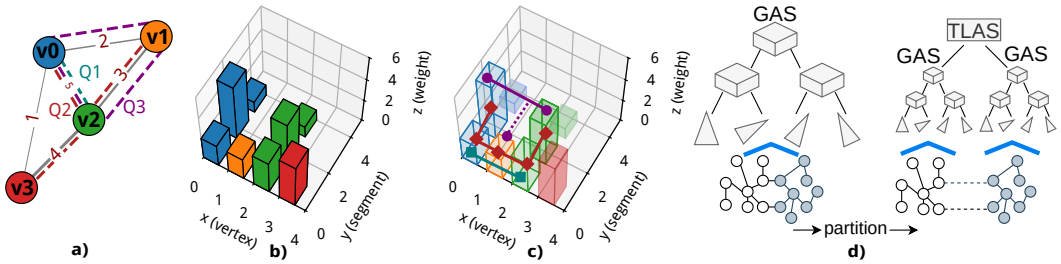


Fig. 3. UASP representation and query encoding. (a) Input graph with weighted edges and example queries: direct (v_0–v_2, Q1), neighbor expansion (v_2, Q2), and triangle detection (Q3). (b) UASP lattice: adjacency lists partitioned into fixed-size 3D cubes indexed by vertex, segment, and weight. (c) Query encoding: Q1 (cyan) → segment traversal; Q2 (red) → segment iteration; Q3 (purple) → neighbor intersection. (d) BVH construction: GAS builds per-partition structures; TLAS assembles the full graph.

4.1 Model Decisions

We design a unified graph data model that executes efficiently on RT cores while remaining compatible with general-purpose SIMT execution, considering prior work [32, 84, 94]. The key design decisions are summarized below.

MD1 *Vertex-centered layout.* A CSR-like, vertex-centric organization where each vertex owns its adjacency segments for compact storage and locality.

MD2 *Adjacency segmentation.* Large adjacency lists are partitioned into bounded segments to mitigate power-law imbalance.

MD3 *Geometric embedding.* Adjacency segments are encoded as geometric primitives in 3D space, using triangles or AABBs for flexible attribute encoding.

MD4 *Graph–metadata separation.* Per-vertex metadata is stored separately and linked via SBT records to avoid duplication and enable efficient access.

MD5 *Queries as rays.* Graph operators are expressed as rays traversing the primitive lattice within the OptiX pipeline.

MD6 *Hierarchical acceleration.* Partitioned BVHs replace a single global BVH, reducing build and traversal costs while enabling scalable memory management.

Together, these decisions yield a unified, geometry-based abstraction that maps graph queries to RT problems, combining GPU data-parallelism with acceleration on dedicated RT-cores.

4.2 From Graph to Geometric Primitives

Based on **MD1**–**MD6**, we introduce *Unified Adjacency Segment Primitives* (UASP) as the core data model, mapping graph adjacency to geometric primitives in 3D space. The CSR representation of G consists of an array `row_ptr` of length $n + 1$, such that `row_ptr`$[v]$ indexes the start of the adjacency list of vertex v, and an array `nbrs` of length $|E|$ storing all adjacency lists concatenated. A direct mapping of vertices or edges to geometric primitives is unsatisfactory. If each vertex were represented by a single primitive, high-degree vertices would dominate traversal; conversely, mapping every edge to a primitive would produce $|E|$ primitives, quickly exhausting memory. To balance these extremes, we partition each vertex's adjacency list into contiguous slices of bounded size, defined by a segment size k. For vertex v and segment index i we set $s = \texttt{row\_ptr}[v] + ik$ and $t = \min(\texttt{row\_ptr}[v] + (i+1)k,\ \texttt{row\_ptr}[v+1])$. We observed that both BVH build time and memory usage are minimized as the number of segments scales as the square root of $|E|$.

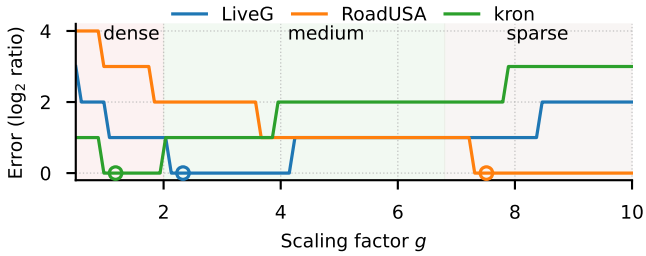


Fig. 4. Sensitivity analysis for scaling factor g, minimizing BVH build time and size (log ratio).

This follows from a simple trade-off between per-segment workload and per-segment overhead. Smaller k values increase the number of UASPs, improving load balance but enlarging the acceleration structure. Larger k reduces structure size at the cost of higher per-intersection workload. Thus, k controls the trade-off between memory footprint and computation. Without segmentation, high-degree vertices would yield excessively large primitives, degrading traversal performance. This maps vertex degree skew to spatial skew, so high-degree vertices correspond to denser spatial regions. Spatial irregularities can be efficiently managed by BVH-based RT, which is designed to handle non-uniform spatial distributions through hierarchical partitioning.

To further account for differences in graph density and structure, we introduce a density-dependent scaling factor g. $k = \text{round\_pow2}((\sqrt{|E|})/g)$ where round_pow2 denotes rounding to the nearest power of two (e.g., 256, 512). The scaling factor g depends on the graph density and is chosen from a range 1-10, through a dataset analysis, as shown in Figure 4, to reduce BVH build and memory footprint. The sparsity is always evaluated on startup. We chose the values that minimized the BVH build time and memory footprint for different graph sparsity, resulting in ranges $g = (1.0, 2.0)$ for dense, $g = (2.1, 6.7)$ for medium, and $g = (6.8, 10.0)$ for sparse graphs. The exact value within the range will be selected based on $|E|$. To accommodate changing sparsity due to updates, we select this design to achieve low sensitivity, ensuring that suboptimal parameter choices do not significantly degrade performance. Varying g only shifts k to the next power of two (e.g., $256 \rightarrow 512$), impacting end-to-end execution time by at most 5%.

For example, a dense graph with $|E| = 1M$ would have a segment size of 512, a sparse 256. Each UASP segment will be assigned metadata, as $M(v, i) = (v,\ s,\ t - s,\ i)$, encoding the parent vertex, start offset, length, and segment index. The metadata is stored in a *Shader Binding Table* (SBT) in GPU memory, where each entry is accessible via its BVH primitive index. The SBT serves as a layer between geometry and computation, enabling the association of graph metadata (e.g., properties) that are not encoded in the UASP and enabling direct $O(1)$ access to the corresponding data during RT. The ray payload is dynamically updated with information retrieved from the intersected edge (via UASP). However, the ray payload should be limited as the size directly affects RT performance.

To leverage spatial RT traversals, we embed UASPs into a three-dimensional lattice by defining a mapping $\phi(v, i) = (x, y, z) = (i,\ v,\ a(v, i))$, where the x-axis encodes the segment index, the y-axis encodes the parent vertex identifier, and the z-axis encodes an attribute function $a(v, i)$. The function $a(v, i)$ may be constant, yielding $(i, v, 0)$, or it may incorporate algorithm-specific state such as edge weights, traversal depth, or visitation flags. Whenever another encoding for the attribute function is required, a complete rebuild is required, or refitting updates, as described in Section 4.6. In this embedding, the adjacency list of v appears as a strip of UASPs along the x-axis at coordinate $y = v$. Each UASP is realized as a geometric primitive occupying a bounded

region around its embedding. In the AABB realization, a UASP with coordinates (i, v, a) is mapped to $B(v, i) = [i, i + 1] \times [v, v + 1] \times [0, z]$, where the z-extent is optional: by default $[0, 1]$, but it may instead encode application-specific properties such as edge weights, labels, or visitation state. To encode multiple attributes within the z-dimension, we adopt a composite mapping that fuses attributes into a single scalar coordinate. In the triangle realization, a UASP is mapped to a triangle whose vertices encode (v, i, a). In both cases, the geometric representation enables OptiX to build BVHs, while the semantic identity of the UASP remains in its metadata $M(v, i)$. The advantage of triangles lies in the higher traversal performance due to direct hardware acceleration for intersection tests. Figure 3a illustrate the UASP model for a weighted graph with $k = 2$, allowing up to two vertices per segment. Each adjacency list is partitioned into fixed-size segments represented as lattice-aligned geometric primitives. Cubes are shown for clarity, though the same approach applies to triangles. The x-axis encodes the vertex index, y the segment index, and z an attribute such as edge weight, transforming adjacency lists into primitive strips. However, OptiX is limited to indexing 2^{32} primitives due to its use of 32-bit identifiers. We therefore adopt a hierarchical BVH.

4.3 BVH Construction

The BVH hierarchically partitions space to accelerate traversal on RT cores. Each UASP is enclosed in an AABB, and these bounds are recursively aggregated into parent nodes. BVH construction is handled on the GPU via `optixAccelBuild()`, while semantic metadata $M(v, i)$ is stored separately and linked to primitive identifiers, allowing graph data retrieval upon ray–primitive intersection. OptiX generates a geometric acceleration structure (GAS) for each group of primitives, representing the BVH. However, we organize multiple GAS structures into a top-level acceleration structure (TLAS), which governs traversal across the graph. Flattening all UASP primitives into a single GAS is possible but inefficient. First, the number of primitives is limited to 2^{32}. Second, multiple GAS instances align with data-parallel execution, as each can be built, refit, or traversed independently. Third, TLAS traversal enables pruning of irrelevant GAS instances, reducing traversal cost. Finally, updates affect only local GAS structures instead of requiring a full BVH rebuild. Thus, a TLAS-based design offers more scalability. For TLAS construction, as shown in Figure 3d, the graph is partitioned into batches, with one GAS built per batch. Each GAS contains a contiguous slice of UASPs, enabling balanced, parallel construction that improves memory locality and amortizes rebuild costs. Asynchronous CUDA streams interleave batch builds for optimal throughput. Each TLAS instance encodes a unique 64-bit identifier as $\texttt{id}_{64} = (\texttt{sbtOffset} \ll 32) \mid \texttt{instanceId}$, where `sbtOffset` denotes the base SBT record index. Combining both fields yields globally unique IDs accessible within hit or intersection programs, allowing efficient memory and metadata management.

4.4 Queries as Rays

We define a graph query as an RT intersection program that, given one or more source vertices, returns vertices from their adjacency information. This generalizes adjacency lookup to neighborhood enumeration, connectivity, and overlap – the core primitives of graph analytics. Queries are represented as rays traversing a lattice of UASPs. Each primitive encodes a bounded portion of adjacency data; ray intersections return its metadata $M(v, i)$ (vertex, offset, length, segment index) and contents $U(v, i)$ (neighbor set). Graph queries are thus expressed geometrically through *ray generation* and *ray–primitive intersection*. A ray originates at $o = (x, y, z)$ from the embedding $\phi(v, i)$ and travels in direction d, where x scans segments, y indexes vertices, and z encodes attributes such as weights or traversal depth. Depending on direction, rays perform segment scanning, neighborhood exploration, or attribute filtering. We define three primitives as the basis for higher-level graph algorithms.

Point-to-point. Given two vertices $u, v \in V$, the query tests whether (u, v) is an edge of G. This is realized by generating a ray with origin $o = (i, u, 0)$ at the UASPs of u and direction $d = (0, 1, 0)$ toward $y = v$. If the ray intersects a primitive $U(u, i)$ whose metadata $M(u, i)$ covers the adjacency range containing v, the edge is confirmed. Point-to-point queries underlie shortest-path relaxations and conditional updates in traversal-based algorithms.

Expansion. For a single source vertex $u \in V$, the expansion query enumerates all neighbors of u. This corresponds to generating rays with origins $o = (i, u, 0)$ for each segment i of u, and directions $d = (1, 0, 0)$ along the x-axis. Each primitive contributes a bounded slice of neighbors, ensuring balanced traversal even when $\deg(u)$ is large. This models a repeated frontier expansion step, where large adjacency lists are naturally partitioned into smaller, balanced traversal units.

Set intersection. For two vertices $u, v \in V$, this query finds common neighbors $w \in V$. In the UASP lattice, rays are generated from $o = (i, u, 0)$ and $o' = (j, v, 0)$ to probe the UASPs of u and v; an intersection occurs when both rays access primitives sharing a neighbor w. This operation underlies subgraph-centric algorithms such as triangle counting.

Figure 3c shows the query encoding within the model for a graph subset. Q1 (*expansion*) enumerates all neighbors of v_2 by generating rays from $o = (i, 2, 0)$ directed along $d = (1, 0, 0)$ across the x-axis. Each intersection returns a bounded neighbor slice from $U(v_2, i)$, producing connections to v_0, v_1, and v_3. This model balances frontier expansion, as each UASP contributes only a fixed-size segment, even for high-degree vertices. Q2 (*point-to-point*) tests whether v_2 is a neighbor of v_0 by casting rays from v_0 with $o = (i, 0, 0)$ and $d = (0, 1, 0)$, intersecting the primitive containing v_2 and confirming the edge (v_0, v_2). Q3 (*set intersection*) computes $Adj(v_0) \cap Adj(v_2)$ by emitting rays from both vertices—$o = (i, 0, 0)$ and $o' = (j, 2, 0)$—each along $d = (1, 0, 0)$ to traverse their adjacency segments. Whenever rays from v_0 and v_2 intersect primitives sharing a neighbor $w \in V$, w is identified as a common neighbor (here, v_1). The BVH accelerates these operations by restricting comparisons to spatially adjacent UASPs instead of full adjacency scans.

4.5 RT Pipeline

The UASP data model integrates with the OptiX RT pipeline, where graph queries are mapped to programmable stages executed on RT cores (cf. Figure 2). Our design leverages the OptiX programming model deliberately: each query operator is mapped to the pipeline stage that best matches its computational structure. The ray generation program provides natural query-level parallelism, issuing one or more rays per source vertex or primitive. This ensures balanced GPU occupancy and amortizes launch overheads across many concurrent queries. Intersection programs are used only to filter and qualify hits against UASP primitives. We deliberately avoid heavy computation here so that traversal stays resident on the RT cores and only minimal metadata extraction is performed (e.g., mapping a primitive index to its UASP (v, i)). This division keeps traversal bandwidth-bound and exploits the fixed-function RT hardware for BVH walking and box/segment intersection. For expansion, the AnyHit program is used because multiple intersections along a ray correspond to multiple adjacency segments that must all be processed. This allows consuming neighbors directly without waiting for traversal to terminate. In contrast, point-to-point queries (e.g., existence checks) use terminate-on-first-hit flags with a lightweight hit program, minimizing shader work once the answer is known. Set-intersection queries benefit from programmable hit shaders that compare neighbors from multiple rays as they converge on common UASPs, enabling efficient detection of shared adjacency. Miss programs remain trivial by design: they encode the absence of results. This exploits the CUDA-RT hybrid processing: RT cores accelerate hierarchical traversal and primitive intersection, while general-purpose CUDA cores implement the programmable logic for the reduced candidate set. The integration of a multi-level BVH hierarchy with the ray tracing pipeline facilitates efficient updates to the graph structure.

4.6 Updates

The BVH can be refitted efficiently when the positions of *existing* primitives change, as this operation only updates the bounding boxes and leaf node extents without altering the tree topology. However, refitting is only possible when the set of primitives remains fixed. Any topological changes to the graph cannot be incorporated into the BVH through refitting and therefore require a complete rebuild. To support incremental updates without full rebuilds, we adopt a *pre-allocation strategy* using *dummy primitives*.

During the initial BVH construction, we allocate a number of placeholders outside the scene (the dummies), unreachable from any ray, e.g., in negative bounds. These dummy primitives do not participate in any ray intersections. In case of an update, i.e., by adding a new node with a new neighbor set, we reuse a subset of these preallocated dummy primitives by overwriting their vertex data with the new data. For new edges, we select the appropriate UASP, update the neighbor set, create a new UASP, switch the primitives, and relocate the others. For deletions, we move the primitive outside the scene, unreachable for rays. This allows the BVH to be refitted, which is much faster than a complete rebuild.

For a small number of updates, the UASP model with scaling factor g is largely unaffected due to its low sensitivity. Even when the graph topology changes significantly (e.g., from sparse to dense), the overhead caused by suboptimal BVH spatiality remains lower than performing a full rebuild after each update. Once all dummies are consumed, g is recalculated to ensure optimal parametrization. This strategy enables lightweight incremental updates in otherwise static acceleration structures, at the cost of maintaining a pool of inactive primitives that must be preallocated at build time. When all dummy primitives are used, the whole BVH has to be rebuilt, and new dummy primitives are allocated into the dummy pool.

5 GraphRTX: RT-Accelerated Graph Analytics

We implement a set of graph algorithms grouped by their dominant RT operation: *traversal*, *subgraph identification*, and *aggregation*. The following sections detail representative algorithms and their RT-based implementations.

Algorithm 1: BFS using RT acceleration. CUDA is shown in green, RT intersection in red.

Input : UASP graph, source vertex s, OptiX pipeline $pipe$, TLAS handle t
Output: Distances $dist[0..N-1]$

1 $dist[:] \leftarrow \infty$, $visited[:] \leftarrow 0$; $dist[s] \leftarrow 0$, mark s visited;
2 $frontier \leftarrow \{s\}$, $level \leftarrow 0$, $buf \leftarrow 0$
3 **while** $frontier \neq \emptyset$ **do**
4 $jobs[buf] \leftarrow$ build from $frontier$; $nextCount[buf] \leftarrow 0$
5 $params \leftarrow \{t, jobs[buf], nextFrontier[buf], nextCount[buf], visited, dist\}$
6 `optixLaunch`$(pipe, params, |frontier|)$:
7 RayGen: for each job $(u, depth)$ trace ray from $\phi(u)$ along $d = (1, 0, 0)$
8 Intersection: identify UASP segment belonging to u
9 AnyHit: enumerate neighbors (u, v)
10 if $visited[v] = 0$ then mark, set $dist[v] = depth + 1$, enqueue v
11 $frontier \leftarrow nextFrontier[buf][0..nextCount[buf] - 1]$
12 $buf \leftarrow 1 - buf$, $level \leftarrow level + 1$

Graph Traversals. For traversals, we implement two algorithms.

► *BFS.* Algorithm 1 presents the BFS algorithm in pseudo-code, with CUDA components highlighted in green and RT components in red. The algorithm follows a frontier-expansion model. On the host, the current frontier is maintained, and a *job buffer* is constructed for each level in parallel, where each job represents an active vertex u with its BFS depth. Jobs are transferred to the device, where the OptiX pipeline expands them in a ray-parallel manner. Using a push-based strategy, each active vertex launches a ray representing a traversal request from a canonical origin derived from its index, propagating along the x-axis. Although the direction has no graph semantics, it enables traversal through the UASP acceleration structure. At traversal, the intersection program identifies intersecting adjacency segments belonging to the source u, ensuring mapping between rays and primitives. The anyhit program then enumerates all neighbors of u, atomically updating the visited bitmap and assigning $dist[v] \leftarrow dist[u] + 1$ for each newly discovered vertex. Discovered vertices are appended to the next frontier, and rays are retraced, alternating execution between CUDA and RT stages.

► *SSSP.* The overall structure of SSSP closely mirrors the BFS formulation in Algorithm 1, differing mainly in job payload and relaxation semantics. Each job carries a vertex identifier and its tentative distance $dist[u]$. Rays expand the current frontier in parallel as in BFS. During intersection, after ownership verification, the anyhit program enumerates weighted edges (u, v, w) and relaxes them by atomically updating $dist[v] \leftarrow dist[u] + w$ if a shorter path is found. Vertices whose distances change are added to the next frontier, and propagation continues until convergence.

Sub-Graph Identification identifies graph patterns or clusters. We implemented three algorithms with RT.

► *TC.* Unlike other work [94], we do not perform set intersection entirely within RT, but combine RT and CUDA execution: RT identifies candidate adjacency segments, while CUDA performs the actual set intersection. We define this algorithm as an edge-centric intersection problem: finding common vertices in neighbor sets. On the host side, we define RT jobs for each edge (u, v) with $u < v$. Therefore, a ray encodes the edge (u, v) rather than a single source vertex. Again, intersection in combination with the anyhit program ensures that the ray only reports the adjacency segment belonging to u, but representation now considers candidate sets for set intersection rather than frontier expansion. Once the correct adjacency segment is identified, the anyhit program performs a set intersection between $N(u) \cap N(v)$. Every match corresponds to a closed triangle (u, v, w). The count is accumulated locally and later aggregated across jobs on the host. We use this TC algorithm to calculate the *LCC* for each node.

► *WCC.* Similar to TC, WCC is implemented as a set-intersection pipeline, where intersections provide evidence of 2-hop connectivity. A component representative array is maintained on the host and updated iteratively until convergence. For each undirected edge (u, v) with $u < v$, an RT job encodes the edge as in TC, and traversal identifies the adjacency segments for u and v. CUDA computes the intersection $N(u) \cap N(v)$, and a non-empty intersection implies the existence of a witness w such that (u, w, v) forms a length-2 path, indicating that u and v belong to the same connected component.

► *CDLP.* CDLP follows the TC pipeline, but uses set intersection to infer shared-neighbor support and propagate the most frequent label. An array of vertex labels is allocated on the host, and RT jobs are generated for each edge (u, v) with $u < v$, where each ray encodes the vertex pair. RT traversal identifies the corresponding adjacency segments, yielding the sorted neighbor lists $N(u)$ and $N(v)$, which are intersected on the GPU. Labels of common neighbors are updated atomically, and updates are accumulated locally and periodically aggregated on the host until the number of label changes falls below a convergence threshold.

Aggregational. For these, RT is only used for coordination while CUDA processes the actual computation of the results.
►*PR.* The host maintains two score arrays (PR and PR_{new}), initializes them uniformly, and iteratively updates them until convergence. In each iteration, RT jobs are generated for all vertices, where a ray from vertex u carries the contribution $PR[u]/\text{outdeg}(u)$. Traversal identifies the adjacency segment for u, and the AnyHit program atomically accumulates $\alpha \cdot PR[u]/\text{outdeg}(u)$ into $PR_{\text{new}}[v]$ for each neighbor $v \in N(u)$. After completion, the arrays are swapped, and convergence is checked.
►*BC.* BC combines an SSSP-style forward traversal with a PR-like backward accumulation. In the forward pass, rays emitted from frontier vertices carry distance and path-count information, enabling edge relaxation and shortest-path counting. In the backward pass, vertices are processed in reverse level order, and rays propagate dependency values along shortest-path edges. After dependency propagation completes, the host accumulates centrality scores for each vertex. This execution demonstrates how identical RT primitives support both shortest-path discovery and dependency aggregation.

5.1 Memory Management & Multi-GPU

We incorporate an adaptive memory management strategy that partitions the graph's spatial representation into independently loadable units. Each partition corresponds to a contiguous range of spatial primitives and is mapped to an individual BVH, allowing fine-grained control over GPU memory usage (cf. Figure 3d). If all partitions fit into GPU memory, a single TLAS is constructed referencing all GAS instances; otherwise, a streaming mode is activated in which partitions are loaded and evicted on demand.

The number and size of partitions are dynamically determined according to the available GPU memory. We approximate this as $p = \left\lceil \frac{C(G+B)}{M_{\text{avail}}} \right\rceil$, where M_{avail} denotes the usable GPU memory (typically about 70% of total VRAM after accounting for system overhead, G is the graph size in GB, B the estimated BVH size in GB, and C a small correction factor ($C \approx 1.5$ for dense and $C \approx 1.7$ for sparse graphs). The correction factor C accounts for temporary buffers, alignment, and runtime workspace. In our experiments, it typically stabilizes around $C \approx 1.5$ for dense and $C \approx 1.7$ for sparse graphs, as their fragmented adjacency structure leads to more BVH nodes, greater boundary duplication, and higher temporary buffer overhead in algorithms. The correction factor C is not performance-critical, and primarily serves as a safety margin. Varying C by 10% has no impact on partitioning or runtime, unless it leads to memory overcommitment. Algorithms request GAS instances from the memory manager for the required UASP vertices.

If a partition is resident, it is accessed directly; otherwise, the least recently used BVH is evicted with an LRU policy to maintain high memory utilization. Transfer sizes are typically around 1-2 MB. Exploiting frontier-based temporal locality, the manager prefetches partitions for upcoming frontiers in the background, enabling RT-based graph analytics to efficiently process graphs larger than GPU memory via TLAS-managed partitions. In addition, GraphRTX supports parallel execution of traversal-based graph algorithms (BFS, SSSP) across multiple GPUs. Similar to [73], to limit synchronization and communication overhead, graph metadata is replicated on all GPUs, while each GPU owns a distinct BVH partition; only algorithm-specific state (e.g., the visited set in BFS) is shared. Work is distributed dynamically among GPUs.

For example, in BFS, a frontier of 16 vertices is split across four GPUs, each processing four vertices. After each iteration, intermediate results (such as updated frontiers) are sent to the host, which synchronizes and redistributes them. Coordination is handled via a queue-based mechanism in which GPUs dequeue assigned work, execute one iteration, and enqueue results for host-mediated redistribution. Memory management follows the per-device strategy and is handled on each GPU.

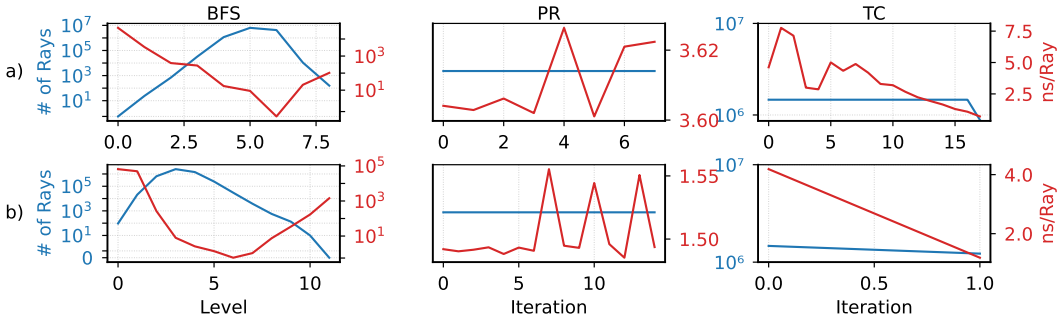


Fig. 5. RT ray performance on a) Orkut and b) LiveG.

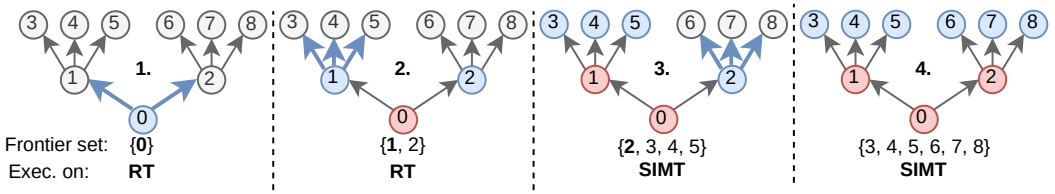


Fig. 6. Stepwise cooperative BFS (threshold $F = 4$).

5.2 Cooperative Execution

While RT accelerates spatial and sparse traversals effectively, its internal scheduling limits aggregation-heavy computation and throughput. In order to achieve the maximum possible ray processing performance, a high resource utilization of RT-cores is necessary, which requires a sufficiently large number of RT calls. With a lower number of traced rays, some RT-cores would idle, increasing the total time required per ray. For very large data, processing is limited by ray-level parallelism. Consequently, RT-based graph algorithms exhibit limited performance benefits for very small or very large workloads when processing traversals.

Figure 5 shows the RT-based processing performance for BFS, PR, and TC. While BFS starts initially with a low frontier set, the processing time per ray is high, since only a few rays per belonging segment could be traced, leading to low ray-level parallelism. More rays can be traced when the frontier size increases, leading to decreased per-ray processing time in total. After reaching an optimal ray number, limited by OptiX, processing time increases again. To mitigate this, we introduce a cooperative CUDA–RT execution strategy guided by the UASP model that dynamically switches between RT and SIMT based on algorithm-specific thresholds, as illustrated in Figure 6. The algorithms always start with RT as soon as a measured optimal threshold for frontier size or work for the next iteration is reached, and switch to SIMT-based execution via usual CUDA. We use a lightweight cooperative switching model to decide per iteration whether an operator should run on RT cores (OptiX) or as a pure CUDA kernel. The decision is based on a measured crossover between a fixed RT overhead and a per-active-work cost.

For traversal algorithms, each active vertex contributes a constant amount of traversal work, so the iteration cost can be modeled as $T_{\mathrm{RT}}(F) = t_o + F\,t_n$ and $T_{\mathrm{CUDA}}(F) = F\,t_c$ where F denotes the frontier size, t_o is the fixed RT overhead (e.g., OptiX dispatch and synchronization), t_n is the measured amortized per-frontier cost of RT execution, and t_c is the measured per-frontier cost

of the CUDA implementation. A short startup calibration measures t_o, t_n, and t_c using micro-benchmarks on representative frontiers. The crossover $T^\star$ between T_{RT} and T_{CUDA} defines the switching threshold.

For compute-intensive algorithms, the runtime is dominated by the amount of active work. We therefore define the active workload as $W = \sum_{v \in \mathcal{A}} \deg(v)$, where $\mathcal{A}$ denotes the set of active vertices. The corresponding cost model becomes $T_{\text{RT}}(W) = t_o + W\, t_n$, and $T_{\text{CUDA}}(W) = W\, t_c$, yielding the threshold $W^\star = \left\lceil \frac{t_o}{t_c - t_n} \right\rceil$. In practice, PR and BCC typically exceed $W^\star$ quickly, which explains why only a small fraction of execution benefits from RT acceleration.

6 Evaluation & Discussion

We evaluate GraphRTX (**GRX**) and its cooperative (= hybrid) execution **GRX-H** with competitive baselines. Our objectives are: (*O1*) analyzing sensitivity to parameters such as segment count (k) and partitions (p); (*O2*) benchmarking GRX against state-of-the-art frameworks with profiling insights; (*O3*) evaluating scalability across graphs, and hardware; and (*O4*) assessing update efficiency.

Setup. Experiments are conducted on three GPU-equipped hosts. The first uses an AMD Ryzen 9 9950X3D CPU with an RTX 4070 (128 RT cores, 12 GB VRAM) for memory-pressure tests. The second and third employ 2× RTX 5090 (170 RT cores, 24 GB VRAM) and RTX A6000 Pro (188 RT cores, 96 GB VRAM) GPUs to evaluate scalability. All experiments use NVIDIA OptiX 9 and CUDA 13.0.0. We evaluate GRX on real-world graphs of varying sparsity and size (cf. Table 1), also used in prior work [7, 90, 94]. RT-based baselines include **RTIndex** [32] and reference models **RT-BFS** and **RT-TC** (both as the optimized implementations from [94]). Non-RT baselines are Gunrock [90], GraphBlast [96], cuGraph [43], and Subway [80]. Profiling and kernel analysis are performed using Nsight Compute. All baselines are tuned for peak performance and verified for correctness. We additionally evaluate our approach using the LDBC Graphalytics benchmarks [39]. However, the set of available systems supporting the workloads, particularly for GPU-based execution, is limited. As a result, our LDBC evaluation is restricted to GraphBLAST and an adapted Gunrock runner.

6.1 Model

Segements & Partitions. First, we present results for different UASP generation parameters, including the number of segments k, partitions p, and BVH (TLAS) size and build time (*O1*). Figure 7 summarizes BVH size and build time across datasets, showing predictable scaling consistent with our memory model. As shown in Figure 7a, total BVH size decreases gradually with increasing segments, reflecting the linear relationship between UASPs and their primitives.

Table 1. Graphs used for evaluation and their properties. Sparse graphs are green, dense graphs red.

Dataset	Type	$\lvert V \rvert$ (M)	$\lvert E \rvert$ (M)	Avg. deg.	Diam.	Size (GB)
RoadUSA	Spatial/Road	23.9	57	2.4	>5000	0.53
Livegraph	Social/Web	4.8	86	18	~10	0.68
Kron21	Synthetic	1.5	182	236	~7	1.40
Weibo	Social	58.6	523	17.8	~8	4.21
Twitter10	Social	21.3	530	49.7	~7	4.9
Orkut	Social	11.8	11.8	1	~9	6.5
SK-2005	Web	50.6	1,949.4	38.5	14	23.2
Friendster	Social/Web	65.6	1,806	55	~6	36
datagen	Synthetic	0.6–434	34–2,500	2.3–88	5–15	0.1–40.5
graph500	Synthetic	2–65	64–1,000	26–32	6–12	0.2–60

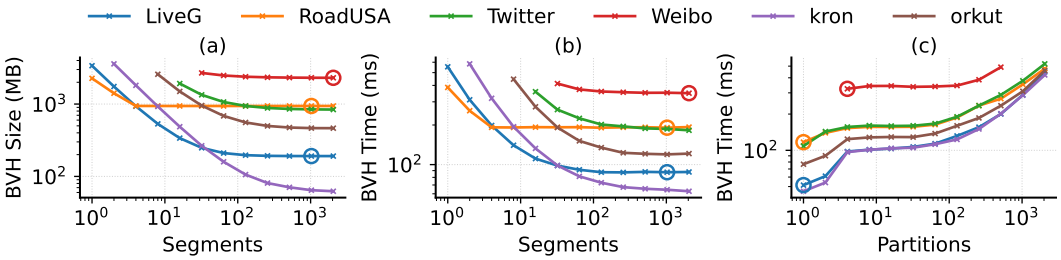


Fig. 7. UASP parameters: a) size vs. segments, b) build time vs. segments, c) build time vs. partitions. Circles mark optimal configurations.

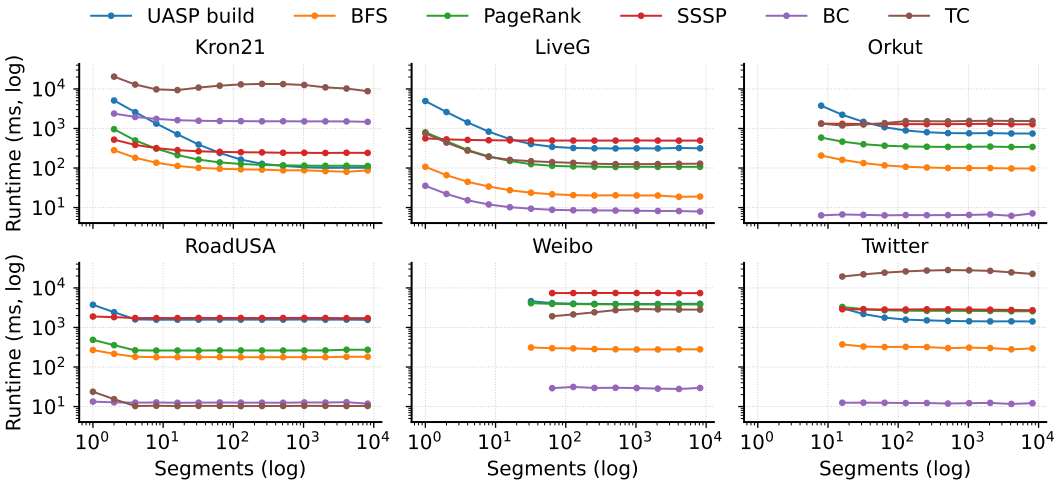


Fig. 8. Runtime of graph algorithms vs. segment count.

A low segment count results in a higher number of primitives. Across all graphs, the BVH footprint remains below 2 GB and decreases with increasing segment count until a plateau is reached, which indicates the optimal number of segments.

Figure 7b shows that BVH construction scales near-linearly with the number of segments, similar to the sizes. A higher segment size than the calculation of our model does not create any further benefits. Figure 7c illustrates the BVH dependence on the partition count p, with missing points for a partition number indicating OOM. For large p, build time is dominated by global preprocessing; decreasing p reduces the BVH build time overhead. Our model calculates the required partitions correctly, illustrated by the circles. For graphs fitting entirely on VRAM, a low partition count is sufficient; for example, one partition is enough for `LiveG`. However, larger graphs such as `Weibo` do not fit entirely into VRAM, including their metadata and BVH structures. Pre-calculation shows that only 1 GB of memory remains available for the BVH. Consequently, the model estimates 8–10 graph partitions, which aligns with our correction factors. Increasing partition count increases BVH build time. The scaling behavior validates our partitioning strategy.

Segments & Algorithms. The impact of segmentation on algorithms is shown in Figure 8. For most algorithms and datasets, runtime decreases with increasing segment count, as a higher segmentation shrinks the BVH and reduces traversal overhead.

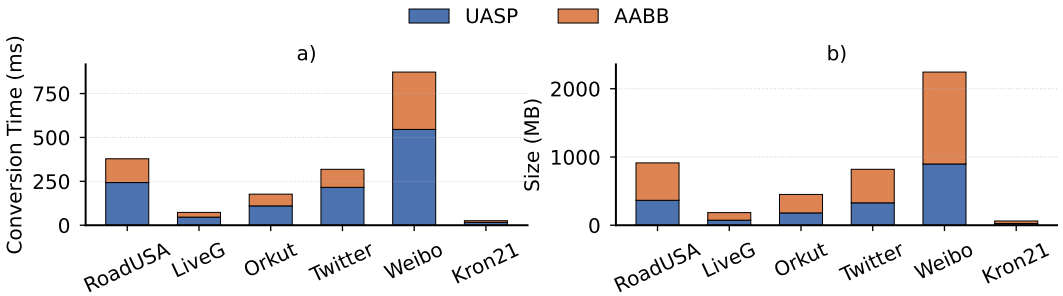


Fig. 9. a) UASP+AABB total conversion time and b) size.

Low segment counts result in more primitives and more intersection tests per ray, increasing computational cost. For smaller graphs, runtimes decrease noticeably with higher segmentation, while for large graphs (e.g., Weibo, Twitter), the effect is marginal, as memory access dominates over BVH traversal.

An exception is TC, where runtime increases with higher segment counts due to larger neighbor sets per segment, increasing pairwise comparison costs during triangle enumeration. In summary, segmentation enhances traversal-based algorithms (e.g., BFS and SSSP) while also providing effective support for TC, PR, WCC, and CDLP.

Graph-Model Conversion. Since our framework relies on spatial acceleration structures, graphs must first be converted into spatial representations before execution of the algorithm. This conversion introduces a one-time preprocessing overhead, which we quantify in Figure 9. Specifically, we measure the time and memory footprint for converting each graph into (1) the UASP using optimal k and (2) the subsequent AABB construction from those, used for the BVH, together with the number of primitives and their size. Overall, the conversion cost scales primarily with the number of vertices ($|V|$), rather than edges ($|E|$), due to vertex-parallel extraction of neighbor sets. For example, Twitter and Weibo have similar $|E|$ and size, but Weibo has twice as many $|V|$, which also doubles the computation time and storage. UASP construction is fast, even for large graphs like Weibo and Twitter, generating 1GB of data in less than a second.

AABB construction is slower since it requires more computation to calculate the boundaries. Memory footprint grows proportionally to the UASP size ($3-4\times$). The resulting BVH size is compact, with 60 MB for Kron, while 2.3 GB for Weibo, which is a relatively large graph. We observe that the preprocessing overhead is relatively small compared to the total runtimes of the algorithms. With Weibo, UASP, and AABB construction together taking about 0.87 s, while subsequent BFS, PR, or SSSP executions take 0.3–7.2 s. Hence, preprocessing accounts for less than 15 % of total runtime, and it can be amortized across multiple algorithm invocations.

BVH Build. A detailed BVH breakdown is shown in Figure 10. NAIVE is a AABB per edge implementation. Since this leads to high VRAM consumption for the BVH, it is limited to small graphs. RT-BFS and RT-TC are specialized for BFS and TC from [94]. As RTIndeX is not graph-based, we loaded the matrix data for each graph in memory and created a multidimensional index for each vertex's entry. GraphRTX consistently produces the smallest BVH memory footprint, ranging between 60 MB and 2.3 GB, depending on graph size. The other approach arises with a higher BVH memory footprint ($2-3\times$), for the same graph. For RT-BFS and RT-TC, this overhead stems from the edge-centric approach in both models, while RTIndex is not optimized for graph data. The NAIVE approach allocates the most memory due to its per-edge model, leading to memory allocation problems on larger graphs.

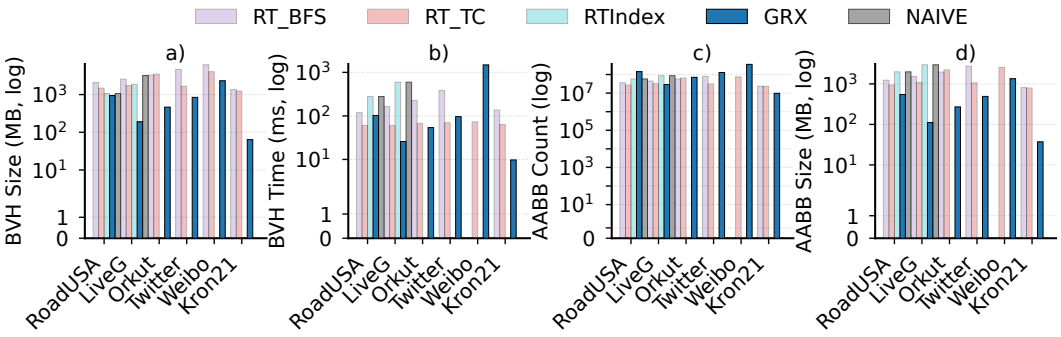


Fig. 10. BVH: a) size, b) time; Primitive c) count, d) size.

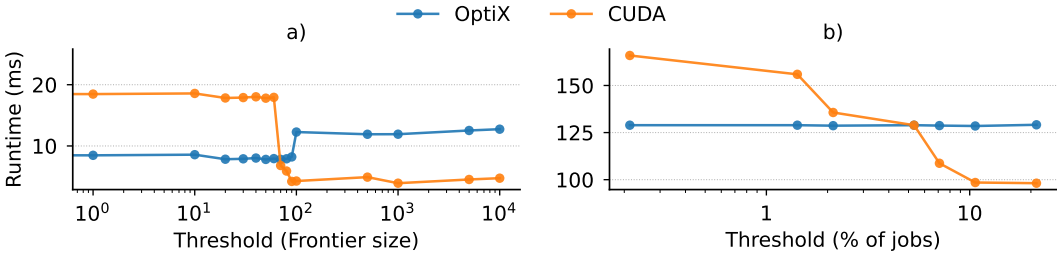


Fig. 11. Cooperative RT-SIMT execution on LiveG: a) BFS and b) PR

For BVH construction, GraphRTX exhibits greater scalability. BVH build times range from 9–100 ms for small graphs (Kron21, LiveGraph) to about 1.5 s for Weibo, despite constructing over 350 million primitives. Segmentation effectively aggregates edges into neighbor sets, reducing primitive counts compared to alternative approaches, which suffer from higher build times. Unlike RT-TC and RT-BFS, which are algorithm-specific pipelines requiring costly model conversion (e.g., from adjacency lists to triangle meshes) between invocations, GraphRTX maintains a unified BVH representation. This allows all algorithms to use a single BVH. Consequently, the UASP model provides a lightweight, versatile graph representation that avoids the overhead of specialized pipelines for the execution of graph algorithms using RT.

Cooperative Thresholds. Figure 11 illustrates the total runtime of hybrid BFS execution as a function of the RT offloading threshold. This threshold determines when work is dispatched to the RT engine versus the CUDA traversal kernel. For BFS, the runtime decreases rapidly as the threshold increases, reaching a minimum of approximately 7.8 ms for frontier sizes between 20 and 70. Beyond this range, the runtime increases sharply due to limited RT throughput. In contrast, pure CUDA execution initially suffers from higher kernel launch overhead, which diminishes as the frontier size increases (around 70 frontiers), improving resource utilization. For PR and BC, the best RT performance is achieved at low thresholds. Similar to BFS, the CUDA kernels exhibit higher launch overhead and limited utilization at small workloads, which improves as the number of processed jobs increases. Overall, these results demonstrate that cooperative RT–SIMT execution effectively balances the two approaches through cooperative workload switching, leveraging their complementary strengths to reduce total runtime significantly.

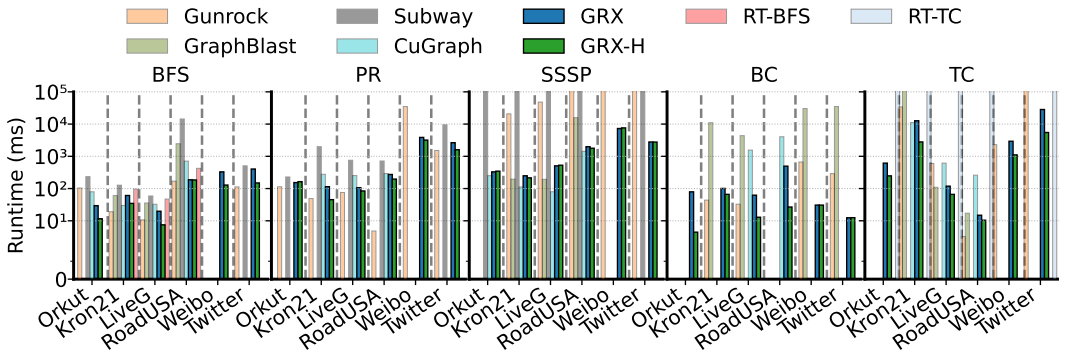


Fig. 12. GPU-based graph frameworks on different datasets. Not shown bars indicate OOM.

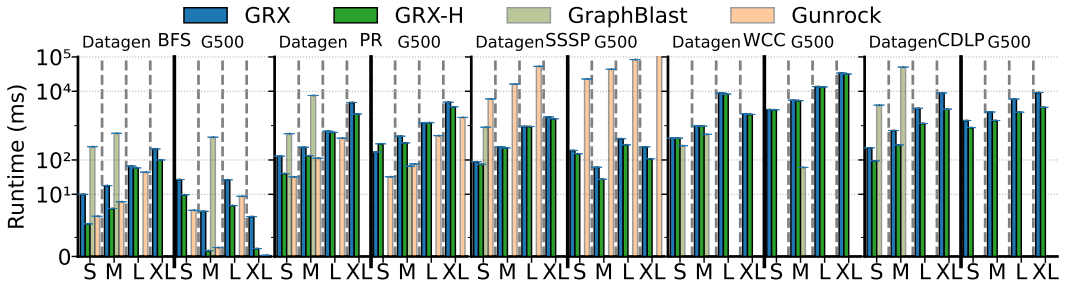


Fig. 13. LDBC Graphalytics on the datagen & graph500 at S-XL scaling. Not shown bars indicate OOM.

6.2 Analytics

Ray Tracing in Operators. We evaluate GraphRTX primitives (Figure 5), showing the evolution of active rays (left axis) and per-ray processing cost (right axis) for selected operators.

For BFS, each UASP traces a ray to traverse the adjacency range. The number of rays grows with the number of frontiers in each level. The cost of each ray decreases with the number of entries in the frontier, leading to more efficient processing when ray-level parallelism is utilized. Lower ray numbers reduce the efficiency. PR exhibits nearly constant behavior across iterations, while also exhibiting a constant number of rays, which is equal to the UASP count. This is because PR uses rays only as dispatch mechanisms for the aggregation kernels, which are dominated by memory bandwidth and arithmetic accumulation rather than ray traversal.

In TC, rays are used for orchestrating set intersection between neighbor sets, which generate variable workloads per ray depending on vertex degrees. Ray-level parallelism is most effective during early, high-density phases. Operator characteristics determine RT hardware utilization: operators issuing many short, independent rays (BFS, SSSP, TC) gain substantial speedups from concurrency and spatial coherence, while compute- or memory-bound workloads see limited benefit, with rays acting mainly as a coordinator.

GPU-based frameworks. The runtimes of all graph algorithms are shown in Figure 12 (*O2*). On average, GraphRTX matches Gunrock's performance while being 2–10× faster than GraphBLAST, Subway, and cuGraph, demonstrating the potential of RT. For example, on Twitter (BFS), GraphRTX completes in 404 ms versus 112 ms for Gunrock, and hybrid execution further reduces traversal time by up to 4× via parallel frontier scheduling.

Table 2. GPU throughput across algorithms and systems.

Algo	System	Graph	L1(%)	L2(%)	Memory(%)	Compute(%)
BFS	GRX	LiveG	40.49	59.20	44.98	12.24
BFS	GRX-H	LiveG	**45.09**	**66.68**	**66.68**	12.24
BFS	Gunrock	LiveG	27.08	39.80	40.20	22.00
PR	GRX	LiveG	**28.30**	**42.51**	42.51	9.00
PR	GRX-H	LiveG	12.90	32.50	71.30	**18.70**
PR	Gunrock	LiveG	13.80	37.20	**92.20**	7.80
SSSP	GRX	Kron21	14.5	25.4	63.1	**7.53**
SSSP	GRX-H	Kron21	18.4	**40.08**	**49.4**	**2.8**
SSSP	Gunrock	Kron21	**49.1**	1.8	2.05	0.02
BC	GRX	RoadUSA	3.15	2.47	5.06	7.53
BC	GRX-H	RoadUSA	**22.4**	**36.2**	**36.2**	**8.43**
BC	Gunrock	RoadUSA	7.28	1.45	4.75	0.09
TC	GRX	LiveG	18.27	24.59	24.59	21.62
TC	GRX-H	LiveG	**38.20**	18.62	**38.23**	**69.00**
TC	Gunrock	LiveG	23.45	**37.78**	37.78	7.56

For Gunrock, this is primarily due to higher utilization of SIMT cores when processing dense graphs. Similar trends appear on RoadUSA, where limited parallelism benefits from hybrid scheduling. Encoding edge weights as spatial primitives (z-dimension) significantly accelerates SSSP through efficient weighted filtering. Whenever the graph is dense, the higher parallel processing of SIMT with CUDA (e.g., Gunrock) is more beneficial than Optix.

Compared to RT-based BFS baselines, GraphRTX delivers consistently higher performance. On RoadUSA, GraphRTX (187 ms) is over 2× faster than RT1 and RT2 (422 ms, 424 ms), with similar improvements on Twitter and Kron21. For computation-heavy algorithms such as PR and BC, GraphRTX is slightly slower than Gunrock but consistently faster than other GPU frameworks, and on large graphs (Weibo, Twitter), it even surpasses Gunrock due to improved locality and reduced synchronization overheads.

On denser graphs, gains are smaller as these workloads are memory-bandwidth bound. On SSSP, CuGraph performs slightly better than GRX and GRX-H (329/346 ms vs. 250ms which is mainly due to its SIMT optimized implementation by NVIDIA. In contrast, TC shows the largest benefits, with GraphRTX achieving 3–10× speedups from dense computation and coherent ray intersections; on sparse graphs like RoadUSA, performance converges with baselines. Importantly, GraphRTX processes all datasets without OOM, unlike several baselines, highlighting its compact memory footprint. Against RT-based systems, GraphRTX achieves orders-of-magnitude improvements, for example, on Kron21 (12.7 s vs. 397 s), LiveGraph (119 ms vs. 675 ms), and Twitter (28 ms vs. 8762ms), demonstrating efficiency with a single unified model.

Profiling. Table 2 summarizes GPU profiling results. L1/L2 metrics reflect RT-induced cache locality during BVH traversal, while memory and compute utilization indicate bandwidth and SM efficiency. GRX-H shows the most balanced resource usage by alternating between RT traversal and CUDA compute. For BFS, higher L1/L2 throughput indicates improved locality and fewer traversal stalls from the UASP model. In PR, GRX-H boosts utilization via concurrent CUDA aggregation, whereas Gunrock's high memory footprint and low compute utilization highlight global access overheads. For SSSP, BC, and TC, GRX-H sustains higher activity through pipelined traversal and intersection, outperforming both memory-bound RT and divergent Gunrock kernels. The cooperative model improves cache reuse, bandwidth, and memory utilization.

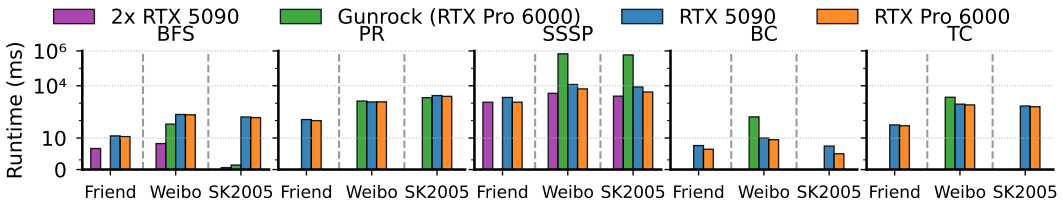


Fig. 14. Scaling graph sizes and hardware.

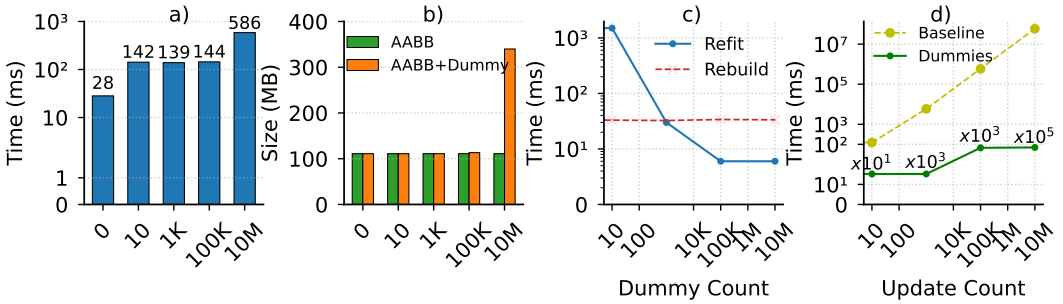


Fig. 15. Update dummy a) overhead and b) time.

LDBC Graphalytics. We evaluate our system on the LDBC Graphalytics benchmark (S–XL), with results shown in Figure 13. On datagen graphs, Gunrock is up to 1.5× faster than GRX-H for BFS and PR at small scales, but this advantage diminishes as graph sizes grow. Compared to GraphBlast, GRX consistently achieves lower runtimes and completes all scales without OOM, demonstrating strong architectural scalability. For WCC, GRX, and GRX-H match GraphBlast when normalized to the same iteration count. On aggregation-intensive kernels such as TC and CDLP, GRX-H delivers 2×–4× speedups over GRX. On Graph500, GRX and GRX-H maintain stable scalability across increasing sizes. GRX-H achieves competitive BFS performance relative to Gunrock and clearly outperforms GraphBlast on medium and large scales. For PR and WCC, GRX consistently reports lower runtimes than both baselines. While GraphBlast fails on several Graph500 configurations, GRX completes all scales, confirming its scalability across graph sizes.

Varying Devices. We evaluate larger graphs on other GPUs (Figure 14) (*O3*). Gunrock ran only on the RTX 6000 Pro and encountered OOM on the RTX 5090. GraphRTX scales efficiently across all datasets and hardware: on Friendster (65M vertices, 1.8B edges), all operators complete within seconds, with BFS and SSSP achieving sub-second runtimes. Computation-bound algorithms (PR, TC) also scale well, with minor GPU-to-GPU variation from arithmetic saturation. While Gunrock is faster on BFS and comparable on PR, GraphRTX maintains stable performance on large datasets without exceeding memory limits.

Multi-GPU. Next, we evaluate running the traversal-based algorithms on multi-GPUs (*O3*). For this, we use the same datasets and 2x NVIDIA RTX 5090 GPUs. The runtimes for the algorithms, especially for BFS, can be reduced drastically (104 ms vs 24ms on Friendster), which is mostly due to the increased parallelism and low synchronization approach via BVH partitions. The SSSP algorithm runtimes can also be decreased with multi-GPU, although there is a lower speedup than with BFS (2136 ms vs 1134 ms on Friendster).

6.3 Updates

We evaluated the update performance of GraphRTX to assess how efficiently the BVH adapts to updates without full reconstruction (*O4*).

Dummy Conversion. Figure 15a shows that the UASP-to-primitive conversion time remains nearly constant (28 ms) even as the number of inserted dummy elements increases by several orders of magnitude, confirming that preprocessing scales sublinearly with update size. Figure 15b shows that memory usage also grows modestly: the primitive representation stays stable up to 100 K updates, and even at 10 M insertions, total memory increases by less than 3×, demonstrating effective memory management.

Update costs. In Figure 15c, we compare the per-update cost of incremental BVH updates with full rebuilds. The per-update cost drops rapidly as the number of updates increases and stabilizes at only a few microseconds per update, while full rebuilds remain constant in the tens of milliseconds range. Figure 15d summarizes the total runtime impact: incremental updates outperform complete rebuilds by up to three orders of magnitude once the update batch exceeds 100 K elements. These results are similar for any kind of update, since it would process the same workflow.

Influence on performance. We evaluate the impact of refitted dummies (via insert updates), now represented as regular UASPs, on graph algorithm performance. Results are shown in Figure 16 using the LiveG dataset and representative traversal (BFS, SSSP) and computation-heavy (PR, BC) algorithms. We vary the number of incorporated dummies from 10% to 90% of total graph nodes. Overall, the added dummies incur only modest overhead, increasing runtimes by 1–4%. Traversal-based algorithms exhibit a gradual runtime increase with more nodes, while computation-based algorithms maintain a nearly constant 3–4% overhead, as most computation is unaffected by additional BVH nodes.

Influence on graph parameters. Lastly, we evaluate the performance impact of substantial graph modifications: sparse-to-dense (edge insertions) and dense-to-sparse (edge deletions) transformations. For insertions, we identify pairs of neighbors-of-neighbors and connect them directly, thereby increasing the size of common neighborhoods. For deletions, we compute the top-k neighbors of each node and remove 30% of these edges connecting them. The results are shown in Figure 17, using LiveGraph for the sparse-to-dense scenario and Kron21 for the dense-to-sparse scenario. Similar to the previous benchmarks, there is only a small overhead of 6% on algorithmic runtime when substantial graph topology changes occurred. This is mostly due to internal BVH optimization when refitting data in the BVH.

This demonstrates that GraphRTX can efficiently maintain spatial structures for evolving graphs.

6.4 Efficiency & Limitations

Energy. An advantage of RT is reduced energy consumption for BVH traversal on fixed-function hardware. We measure end-to-end energy by combining CPU package energy (Intel RAPL via `perf stat`) and GPU energy (integrated `nvidia-smi` power samples), aligned to the same execution window. For BFS, the OptiX-based implementation consumes ~392J (328.7J CPU, ~63,J GPU), compared to ~960J for CUDA (Gunrock) (662J CPU, ~300J GPU), yielding up to 6% lower energy consumption. This demonstrates that using RT cores can reduce end-to-end energy consumption.
Limitations. While traversal-based algorithms and dynamic updates show promising results, limitations arise for aggregation-heavy workloads and small or large, dense graphs. As shown in Figure 5, for small graphs (e.g., $|E| < 1$M) or batch sizes, ray-level parallelism incurs underutilization of the RT hardware. For large, dense graphs, performance is limited by ray-level parallelism: insufficient ray batches underutilize RT cores, increase scheduling overhead, and reduce memory-level parallelism, raising per-ray latency.

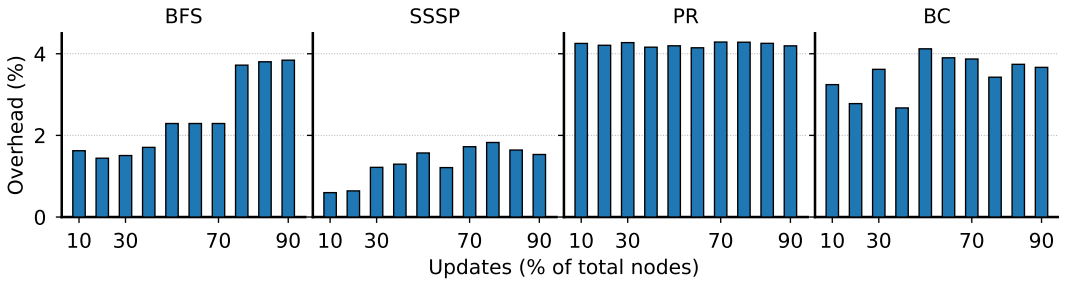


Fig. 16. Overhead on algorithms after dummy updates.

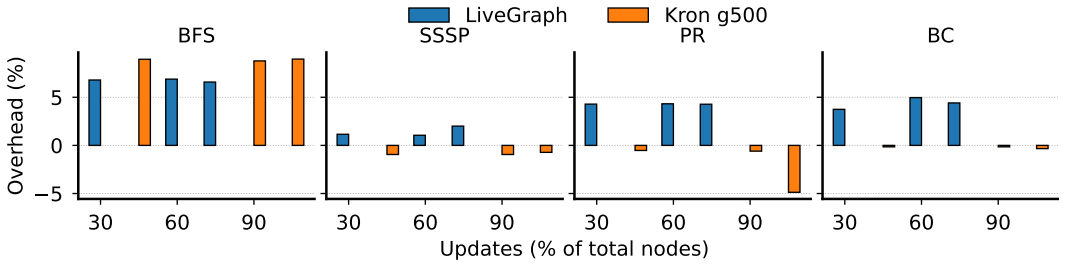


Fig. 17. Overhead due to changing graph topology.

Aggregation-heavy kernels are often bounded by arithmetic throughput rather than traversal efficiency, limiting the benefit of RT offloading and causing potential contention between SMs and RT units. Finally, the restricted programmability of the RT pipeline confines applicability to algorithms naturally expressed as traversal queries, while workloads requiring complex control flow or coordination are better suited to conventional GPU execution. Therefore, the most benefit from offloading to RT cores can be achieved when graphs are sparse but sufficiently large, and on traversal-based workloads.

7 Conclusion

We present `GraphRTX`, a framework for graph analytics that leverages RT cores. Unlike prior RT-based techniques with operator-specific models and costly BVH rebuilds, `GraphRTX` uses a unified spatial representation that supports diverse algorithms (BFS, PR, SSSP, BC, WCC, CDLP, TC) and remains extensible. Its hierarchical BVH design enhances memory efficiency, scalability, and update flexibility, enabling large-graph analysis even on GPUs with limited VRAM. Our experiments show that `GraphRTX` achieves low preprocessing overhead and competitive, often superior, runtime performance on sparse graphs. On dense or larger graphs, however, SIMT cores still offer higher throughput due to their greater parallel processing capacity. While ray-level parallelism provides limited concurrency, our cooperative SIMT–RT execution mitigates this and improves overall device utilization. With RT cores now standard across GPU generations, we see growing potential for their use in accelerating future graph analytics. Further opportunities lie in combining RT and SIMT execution with tensor cores, where leveraging each accelerator for its optimal workload could overcome long-standing GPU inefficiencies in graph processing.

References

[1] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael B. Abu-Ghazaleh, and Rajiv Gupta. 2023. CommonGraph: Graph Analytics on Evolving Data. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 133–145. doi:10.1145/3575693.3575713

[2] Michael Bauer, Henry Cook, and Brucek Khailany. 2011. CudaDMA: optimizing GPU memory bandwidth via warp specialization. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, Scott Lathrop, Jim Costa, and William Kramer (Eds.). ACM, 12:1–12:11. doi:10.1145/2063384.2063400

[3] Michael Bauer, Sean Treichler, and Alex Aiken. 2014. Singe: leveraging warp specialization for high performance on GPUs. (2014), 119–130. doi:10.1145/2555243.2555258

[4] Alexander Baumstark, Philipp Götze, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2021. Instant Graph Query Recovery on Persistent Memory. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 10:1–10:4. doi:10.1145/3465998.3466011

[5] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2010. Efficient algorithms for large-scale local triangle counting. *ACM Trans. Knowl. Discov. Data* 4, 3 (2010), 13:1–13:28. doi:10.1145/1839490.1839494

[6] Norbert Beckmann and Bernhard Seeger. 2009. A revised r*-tree in comparison with related index structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 799–812. doi:10.1145/1559845.1559929

[7] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. (2017), 235–248. doi:10.1145/3018743.3018756

[8] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6 (2024), 237:1–237:26. doi:10.1145/3698812

[9] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. 65:1–65:12 pages. doi:10.1145/2063384.2063471

[10] Federico Busato and Nicola Bombieri. 2015. BFS-4K: An Efficient Implementation of BFS for Kepler GPU Architectures. *IEEE Trans. Parallel Distributed Syst.* 26, 7 (2015), 1826–1838. doi:10.1109/TPDS.2014.2330597

[11] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 1–7. doi:10.1109/HPEC.2018.8547541

[12] Jiaping Cao, Le Xu, Man Lung Yiu, Jianbin Qin, and Bo Tang. 2025. GPH: An Efficient and Effective Perfect Hashing Scheme for GPU Architectures. *Proc. ACM Manag. Data* 3, 3 (2025), 165:1–165:26. doi:10.1145/3725406

[13] Zheng Chen, Feng Zhang, Jiawei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. 2023. CompressGraph: Efficient Parallel Graph Analytics with Rule-Based Compression. *Proc. ACM Manag. Data* 1, 1 (2023), 4:1–4:31. doi:10.1145/3588684

[14] Yuze Chi, Licheng Guo, and Jason Cong. 2022. Accelerating SSSP for Power-Law Graphs. In *FPGA '22: The 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Virtual Event, USA, 27 February 2022 - 1 March 2022*, Michael Adler and Paolo Ienne (Eds.). ACM, 190–200. doi:10.1145/3490422.3502358

[15] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. doi:10.1109/MM.2021.3061394

[16] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. *Proc. VLDB Endow.* 17, 6 (2024), 1405–1417. doi:10.14778/3648160.3648179

[17] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[18] Harish Doraiswamy, Huy T. Vo, Cláudio T. Silva, and Juliana Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 1086–1097. doi:10.1109/ICDE.2016.7498315

[19] Nhat Tan Duong, Quang Anh Pham Nguyen, Anh Tu Nguyen, and Huu-Duc Nguyen. 2012. Parallel PageRank computation using GPUs. In *Symposium on Information and Communication Technology 2012, SoICT '12, Halong City, Quang Ninh , Viet Nam, August 23-24, 2012*, Eric Castelli, Khanh Tran Duc, Chi Mai Luong, and Viet Tran (Eds.). ACM, 223–230. doi:10.1145/2350716.2350751

[20] Per Fuchs, Domagoj Margan, and Jana Giceva. 2023. Sortledton: a Universal Graph Data Structure. *SIGMOD Rec.* 52, 1 (2023), 17–25. doi:10.1145/3604437.3604442

[21] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: eXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed*

Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019, Jon B. Weissman, Ali Raza Butt, and Evgenia Smirni (Eds.). ACM, 121–131. doi:10.1145/3307681.3326606

[22] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 55–64. http://www.adms-conf.org/2021-camera-ready/gao_adms21.pdf

[23] Isaac Gelado and Michael Garland. 2019. Throughput-oriented GPU memory allocation. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 27–37. doi:10.1145/3293883.3295727

[24] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2024. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the 38th ACM International Conference on Supercomputing, ICS 2024, Kyoto, Japan, June 4-7, 2024*, Kenji Kise, Valentina Salapura, Murali Annavaram, and Ana Lucia Varbanescu (Eds.). ACM, 124–136. doi:10.1145/3650200.3656610

[25] Liang Geng, Rubao Lee, and Xiaodong Zhang. 2025. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 396–411. doi:10.1145/3710848.3710850

[26] Hemant Kumar Giri, Mridul Haque, and Dip Sankar Banerjee. 2020. HyPR: Hybrid Page Ranking on Evolving Graphs. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*. IEEE, 62–71. doi:10.1109/HIPC50609.2020.00020

[27] Andrew S Glassner. 1989. *An introduction to ray tracing*. Morgan Kaufmann.

[28] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[29] Oded Green. 2021. ButterFly BFS - An Efficient Communication Pattern for Multi Node Traversals. *CoRR* abs/2103.13577 (2021). arXiv:2103.13577 https://arxiv.org/abs/2103.13577

[30] 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 *57th IEEE/ACM International Symposium on Microarchitecture, MICRO 2024, Austin, TX, USA, November 2-6, 2024*. IEEE, 1041–1057. doi:10.1109/MICRO61859.2024.00080

[31] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 56:1–56:13. doi:10.1109/MICRO.2016.7783759

[32] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *Proc. VLDB Endow.* 16, 13 (2023), 4268–4281. doi:10.14778/3625054.3625063

[33] Justus Henneberg, Felix Martin Schuhknecht, Rosina Kharal, and Trevor Brown. 2025. More Bang for Your Buck(et): Fast and Space-Efficient Hardware-Accelerated Coarse-Granular Indexing on GPUs. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 1320–1333. doi:10.1109/ICDE65448.2025.00103

[34] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 736–748. doi:10.1145/3448016.3452815

[35] Yu-Ching Hu, Yuliang Li, and Hung-Wei Tseng. 2022. TCUDB: Accelerating Database with Tensor Processors. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1360–1374. doi:10.1145/3514221.3517869

[36] 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, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE / ACM, 14:1–14:12. http://dl.acm.org/citation.cfm?id=3291675

[37] Jianfeng Huang, Cao Yihao, Ren Shubing, Baohua Wu, and Dongjing Miao. 2025. BACH: Bridging Adjacency List and CSR Format using LSM-Trees for HGTAP Workloads. *Proc. VLDB Endow.* 18, 5 (2025), 1509–1521. doi:10.14778/3718057.3718076

[38] Intel Corporation. 2024. Intel Embree: High Performance Ray Tracing. https://www.embree.org/. Accessed: 2025-10-07.

[39] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328. doi:10.14778/3007263.3007270

[40] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022, Taormina, Italy, May 16-19, 2022*. IEEE, 71–80. doi:10.1109/CCGRID54584.2022.00016

[41] Muhammad Attahir Jibril, Hani Al-Sayeh, Alexander Baumstark, and Kai-Uwe Sattler. 2023. Fast and Efficient Update Handling for Graph H2TAP. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 723–736. doi:10.48786/EDBT.2023.60

[42] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://vldb.org/cidrdb/2023/kuzu-graph-database-management-system.html

[43] Seunghwa Kang, Chuck Hastings, Joe Eaton, and Brad Rees. 2023. cuGraph C++ primitives: vertex/edge-centric building blocks for parallel graph computing. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023 - Workshops, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 226–229. doi:10.1109/IPDPSW59300.2023.00045

[44] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 239–252. doi:10.1145/2600212.2600227

[45] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. 2017. GPU-Accelerated Graph Clustering via Parallel Label Propagation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li (Eds.). ACM, 567–576. doi:10.1145/3132847.3132960

[46] Alexander Krolik, Clark Verbrugge, and Laurie J. Hendren. 2023. rNdN: Fast Query Compilation for NVIDIA GPUs. *ACM Trans. Archit. Code Optim.* 20, 3 (2023), 41:1–41:25. doi:10.1145/3603503

[47] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 29:1–29:40. doi:10.1145/3364180

[48] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[49] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David P. Luebke, and Dinesh Manocha. 2009. Fast BVH Construction on GPUs. *Comput. Graph. Forum* 28, 2 (2009), 375–384. doi:10.1111/J.1467-8659.2009.01377.X

[50] Dean De Leo and Peter Boncz. 2019. Packed Memory Arrays - Rewired. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 830–841. doi:10.1109/ICDE.2019.00079

[51] Dean De Leo and Peter Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. doi:10.14778/3447689.3447708

[52] Zhila Nouri Lewis and Yi-Cheng Tu. 2022. G-PICS: A Framework for GPU-Based Spatial Indexing and Query Processing. *IEEE Trans. Knowl. Data Eng.* 34, 3 (2022), 1243–1257. doi:10.1109/TKDE.2020.2992440

[53] Ang Li, Weifeng Liu, Linnan Wang, Kevin J. Barker, and Shuaiwen Leon Song. 2018. Warp-Consolidation: A Novel Execution Model for GPUs. In *Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12-15, 2018*. ACM, 53–64. doi:10.1145/3205289.3205294

[54] Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang. 2017. High Performance Detection of Strongly Connected Components in Sparse Graphs on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2017, Austin, TX, USA, February 5, 2017*, Quan Chen and Zhiyi Huang (Eds.). ACM, 48–57. doi:10.1145/3026937.3026941

[55] Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 403–416. doi:10.1145/2882903.2882959

[56] 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, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 549–565. doi:10.1145/3620665.3640360

[57] 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. *Proc. VLDB Endow.* 17, 6 (2024), 1460–1472. doi:10.14778/

3648160.3648183

[58] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 135–146. doi:10.1145/1807167.1807184

[59] Tobias Maltenberger, Ilin Tolovski, and Tilmann Rabl. 2025. Efficiently Joining Large Relations on Multi-GPU Systems. *Proc. VLDB Endow.* 18, 11 (2025), 4653–4667. doi:10.14778/3749646.3749720

[60] 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, ICS 2024, Kyoto, Japan, June 4-7, 2024*, Kenji Kise, Valentina Salapura, Murali Annavaram, and Ana Lucia Varbanescu (Eds.). ACM, 14–25. doi:10.1145/3650200.3656601

[61] Hunter McCoy and Prashant Pandey. 2024. Gallatin: A General-Purpose GPU Memory Manager. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, Michel Steuwer, I-Ting Angelina Lee, and Milind Chabbi (Eds.). ACM, 364–376. doi:10.1145/3627535.3638499

[62] Adam McLaughlin and David A. Bader. 2014. Scalable and High Performance Betweenness Centrality on the GPU. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, Trish Damkroger and Jack J. Dongarra (Eds.). IEEE Computer Society, 572–583. doi:10.1109/SC.2014.52

[63] Adam McLaughlin and David A. Bader. 2018. Accelerating GPU betweenness centrality. *Commun. ACM* 61, 8 (2018), 85–92. doi:10.1145/3230485

[64] Daniel Meister, Paritosh Kulkarni, Aaryaman Vasishta, and Takahiro Harada. 2024. HIPRT: A Ray Tracing Framework in HIP. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3 (2024), 44:1–44:18. doi:10.1145/3675378

[65] Daniel Meister, Paritosh Kulkarni, Aaryaman Vasishta, and Takahiro Harada. 2024. HIPRT: A Ray Tracing Framework in HIP. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3 (2024), 44:1–44:18. doi:10.1145/3675378

[66] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proc. ACM Manag. Data* 3, 1 (2025), 12:1–12:26. doi:10.1145/3709662

[67] Vani Nagarajan, Rohan Gangaraju, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. 2025. RT-BarnesHut: Accelerating Barnes-Hut Using Ray-Tracing Hardware. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 43–56. doi:10.1145/3710848.3710885

[68] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th International Conference on Supercomputing, ICS 2023, Orlando, FL, USA, June 21-23, 2023*, Kyle A. Gallivan, Efstratios Gallopoulos, Dimitrios S. Nikolopoulos, and Ramón Beivide (Eds.). ACM, 289–300. doi:10.1145/3577193.3593738

[69] Neo4j, Inc. 2020. Neo4j Graph Data Science Library: Scalable Graph Algorithms for Analytics and ML. https://neo4j.com/product/graph-data-science/. Accessed: 2025-10-06.

[70] Yuyao Niu and Marc Casas. 2025. BerryBees: Breadth First Search by Bit-Tensor-Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*. ACM, 339–354. doi:10.1145/3710848.3710859

[71] NVIDIA Corporation. 2022. cuSpatial: GPU-Accelerated Spatial and Spatio-Temporal Data Analytics Library. https://developer.nvidia.com/cuspatial. Accessed: 2025-10-07.

[72] Yunho Oh, Keunsoo Kim, Myung Kuk Yoon, Jong Hyun Park, Yongjun Park, Murali Annavaram, and Won Woo Ro. 2019. Adaptive Cooperation of Prefetching and Warp Scheduling on GPUs. *IEEE Trans. Computers* 68, 4 (2019), 609–616. doi:10.1109/TC.2018.2878671

[73] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. 2017. Multi-GPU Graph Analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. IEEE Computer Society, 479–490. doi:10.1109/IPDPS.2017.117

[74] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David P. Luebke, David K. McAllister, Morgan McGuire, R. Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (2010), 66:1–66:13. doi:10.1145/1778765.1778803

[75] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David P. Luebke, David K. McAllister, Morgan McGuire, R. Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (2010), 66:1–66:13. doi:10.1145/1778765.1778803

[76] Keshav Pingali. 2014. High-speed graph analytics with the galois system. In *Proceedings of the first workshop on Parallel programming for analytics applications, PPAA 2014, Orlando, Florida, USA, February 16, 2014*, Manoj Kumar, Joefon Jann, and Priya Nagpurkar (Eds.). ACM, 41–42. doi:10.1145/2567634.2567648

[77] Adam Polak. 2016. Counting Triangles in Large Graphs on GPU. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 740–746. doi:10.1109/IPDPSW.2016.108

[78] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 472–488. doi:10.1145/2517349.2522740

[79] Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast PageRank Computation on a GPU Cluster. In *Proceedings of the 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2012, Munich, Germany, February 15-17, 2012*, Rainer Stotzka, Michael Schiffers, and Yannis Cotronis (Eds.). IEEE, 450–456. doi:10.1109/PDP.2012.78

[80] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 12:1–12:16. doi:10.1145/3342195.3387537

[81] Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. 2016. *Large-Scale Graph Processing Using Apache Giraph*. Springer. doi:10.1007/978-3-319-47431-1

[82] Albert Segura, José-María Arnau, and Antonio González. 2023. Irregular accesses reorder unit: improving GPGPU memory coalescing for graph-based workloads. *J. Supercomput.* 79, 1 (2023), 762–787. doi:10.1007/S11227-022-04621-1

[83] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (2017), 107–120. doi:10.14778/3151113.3151122

[84] Xuri Shi, Kai Zhang, X. Sean Wang, Xiaodong Zhang, and Rubao Lee. 2024. RTCUDB: Building Databases with RT Processors. arXiv:2412.09337 doi:10.48550/ARXIV.2412.09337

[85] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. (2013), 135–146. doi:10.1145/2442516.2442530

[86] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, and Lei Chen. 2024. Galaxybase: A High Performance Native Distributed Graph Database for HTAP. *Proc. VLDB Endow.* 17, 12 (2024), 3893–3905. doi:10.14778/3685800.3685814

[87] Kai Wang, Don Fussell, and Calvin Lin. 2021. A fast work-efficient SSSP algorithm for GPUs. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27-March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 133–146. doi:10.1145/3437801.3441605

[88] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. 2018. A Comparative Study on Exact Triangle Counting Algorithms on the GPU. *CoRR* abs/1804.06926. arXiv:1804.06926 http://arxiv.org/abs/1804.06926

[89] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 11:1–11:12. doi:10.1145/2851141.2851145

[90] Yangzihao Wang, Yuechao Pan, Andrew A. 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 Trans. Parallel Comput.* 4, 1 (2017), 3:1–3:49. doi:10.1145/3108140

[91] Ziming Wang, Kai Zhang, Yangming Lv, Yinglong Wang, Zhigang Zhao, Zhenying He, Yinan Jing, and X. Sean Wang. 2024. RTOD: Efficient Outlier Detection With Ray Tracing Cores. *IEEE Trans. Knowl. Data Eng.* 36, 12 (2024), 9192–9204. doi:10.1109/TKDE.2024.3453901

[92] Hao Wen and Wei Zhang. 2019. Improving Parallelism of Breadth First Search (BFS) Algorithm for Accelerated Performance on GPUs. In *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. IEEE, 1–7. doi:10.1109/HPEC.2019.8916551

[93] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance Extreme Computing Conference, HPEC 2018, Waltham, MA, USA, September 25-27, 2018*. IEEE, 1–7. doi:10.1109/HPEC.2018.8547566

[94] 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 (2025), 1–25. doi:10.1145/3727108

[95] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 15–29. doi:10.1109/HPCA47549.2020.00012

[96] Carl Yang, Aydin Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1 (2022), 1:1–1:51. doi:10.1145/3466795

[97] Carl Yang, Aydin Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1 (2022), 1:1–1:51. doi:10.1145/3466795

[98] Boseon Yu, Hyunduk Kim, Wonik Choi, and Dongseop Kwon. 2011. Parallel Range Query Processing on R-Tree with Graphics Processing Unit. In *IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing, DASC 2011, 12-14 December 2011, Sydney, Australia*. IEEE Computer Society, 1235–1242. doi:10.1109/DASC.2011.200

[99] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR. *Proc. ACM Manag. Data* 2, 6 (2024), 243:1–243:28. doi:10.1145/3698818

[100] Stefan Zellmann, Jürgen P. Schulze, and Ulrich Lang. 2021. Binned k-d Tree Construction for Sparse Volume Data on Multi-Core and GPU Systems. *IEEE Trans. Vis. Comput. Graph.* 27, 3 (2021), 1904–1915. doi:10.1109/TVCG.2019.2938957

[101] Hongrui Zhang, Yunan Zhang, and Hung-Wei Tseng. 2025. RTSpMSpM: Harnessing Ray Tracing for Efficient Sparse Matrix Computations. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 359–373. doi:10.1145/3695053.3731072

[102] Yongliang Zhang, Yuanyuan Zhu, Hao Zhang, Congli Gao, Yuyang Wang, Guojing Li, Tianyang Xu, Ming Zhong, Jiawei Jiang, Tieyun Qian, Chenyi Zhang, and Jeffrey Xu Yu. 2025. TGraph: A Tensor-centric Graph Processing Framework. *Proc. ACM Manag. Data* 3, 1 (2025), 81:1–81:27. doi:10.1145/3709731

[103] Yongliang Zhang, Yuanyuan Zhu, Hao Zhang, Congli Gao, Yuyang Wang, Guojing Li, Tianyang Xu, Ming Zhong, Jiawei Jiang, Tieyun Qian, Chenyi Zhang, and Jeffrey Xu Yu. 2025. TGraph: A Tensor-centric Graph Processing Framework. *Proc. ACM Manag. Data* 3, 1 (2025), 81:1–81:27. doi:10.1145/3709731

[104] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distributed Syst.* 25, 6 (2014), 1543–1552. doi:10.1109/TPDS.2013.111

[105] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *Proc. VLDB Endow.* 16, 9 (2023), 2172–2185. doi:10.14778/3598581.3598590

[106] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 76–89. doi:10.1145/3503221.3508409

Received October 2025; revised January 2026; accepted February 2026