LibRTS: A Spatial Indexing Library by Ray Tracing

Liang Geng
The Ohio State University
Columbus, Ohio, USA
geng.161@osu.edu

Rubao Lee
Freelance
Columbus, Ohio, USA
lee.rubao@ieee.org

Xiaodong Zhang
The Ohio State University
Columbus, Ohio, USA
zhang.574@osu.edu

Abstract

The Ray-Tracing (RT) core has become a widely integrated feature in modern GPUs to accelerate ray-tracing rendering. Recent research has shown that RT cores can also be repurposed to accelerate non-rendering workloads. Since the RT core essentially serves as a hardware accelerator for Bounding Volume Hierarchy (BVH) tree traversal, it holds the potential to significantly improve the performance of spatial workloads. However, the specialized RT programming model poses challenges for using RT cores in these scenarios. Inspired by the core functionality of RT cores, we designed and implemented LibRTS, a spatial index library that leverages RT cores to accelerate spatial queries. LibRTS supports both point and range queries and remains mutable to accommodate changing data. Instead of relying on a case-by-case approach, LibRTS provides a general, high-performance spatial indexing framework for spatial data processing. By formulating spatial queries as RT-suitable problems and overcoming load-balancing challenges, LibRTS delivers superior query performance through RT cores without requiring developers to master complex programming on this specialized hardware. Compared to CPU and GPU spatial libraries, LibRTS achieves speedups of up to 85.1x for point queries, 94.0x for range-contains queries, and 11.0x for range-intersects queries. In a real-world application, point-in-polygon testing, LibRTS also surpasses the state-of-the-art RT method by up to 3.8x.

CCS Concepts: • **Computing methodologies** → *Parallel algorithms*; **Ray tracing**; • **Information systems** → **Geographic information systems**; **Multidimensional range search**.

Keywords: GPU, Ray Tracing, Spatial Index, Multidimensional Index

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710850

1 Introduction

Ray tracing is a photorealistic rendering technique widely used in the movie industry and video games. A crucial step in this process involves casting rays and determining their intersections with primitives.[1] To reduce the search space, a Bounding Volume Hierarchy (BVH) tree is constructed over the primitives in the scene. However, software-based BVH traversal does not efficiently utilize computing hardware resources—particularly on GPUs—due to branch divergence and irregular memory access patterns. As a result, ray tracing has traditionally been limited to offline rendering [13].

Recent hardware advancements, particularly the integration of Ray Tracing (RT) cores into modern GPUs, have made real-time ray tracing feasible by accelerating both BVH traversal and ray-primitive intersection tests. This innovation delivers an order-of-magnitude speedup over software-based implementations on GPUs. Although RT cores are specifically engineered for ray-tracing rendering, recent research has shown that non-rendering workloads can also benefit from these accelerators by converting tasks into ray-primitive intersection problems. This approach significantly boosts performance for various applications, including nearest neighbor search, clustering, spatial joins, and database queries [22, 22, 26, 32, 38, 40, 41, 47, 49, 49, 58, 61, 74].

Although numerous examples illustrate how RT cores can be repurposed, the process requires carefully adapting the original problem into an RT workload while minimizing overhead [25]. Furthermore, repurposing RT cores demands that users have expertise in both the target domain and RT technology. As a result, current research efforts typically repurpose RT cores on a case-by-case basis.

Numerous successful efforts to repurpose RT cores have motivated us to develop a library that simplifies the programming challenges involved in using these specialized units. Although RT cores are not designed as general-purpose hardware, we argue that spatial workloads are particularly well-suited to them for two main reasons: (1) The BVH accelerated by RT cores serves as a fast spatial index, despite its strict querying constraints; and (2) spatial workloads naturally resemble rendering tasks, as both operate on 2D/3D geometric data. In contrast, non-spatial data workloads must encode 1D data into 3D primitives [26, 40, 44], incurring additional computational and storage overhead.

[1]In ray-tracing rendering, a primitive is a geometry associated with material/texture. In this paper, a primitive is synonymous with geometry.

Motivated by the inherent spatial indexing capabilities of RT cores and their immense potential for spatial workloads, we present LibRTS, a fast and generic spatial indexing library powered by RT cores. LibRTS supports two fundamental spatial queries—point queries and range queries—and can be used out of the box without requiring any RT programming. Moreover, LibRTS supports insertions, deletions, and updates for dynamically changing geometries. By providing a user-friendly interface, LibRTS boosts the productivity of spatial system developers and delivers high-performance spatial queries, making this powerful hardware more accessible to a broader range of users.

Adapting RT cores into a fast and generic spatial index engine presents several challenges:

1. *The Translation Challenge:* RT cores only support detecting ray-primitive intersections. However, spatial queries are more diverse, often involving points, rectangles, and various predicates that differ from ray-primitive intersection tests.

2. *The Load Balancing Challenge:* Highly skewed geometric distributions can cause severe load imbalances among threads. Because RT cores use a single-ray programming model, these imbalances lead to suboptimal performance.

3. *The Mutability Challenge:* RT cores do not support dynamic insertions or deletions of geometries, making it difficult to handle mutations in spatial data [26, 27].

To address these challenges, we transform the range query into a mathematically equivalent rectangle-diagonal intersection test. We then simulate the diagonal using a ray, effectively creating an RT-efficient ray-box intersection test. To resolve the load imbalance issue, we introduce a technique called *Ray Multicast*, which evenly distributes geometries into k sub-spaces. Consequently, k rays are cast into these sub-spaces, reducing each thread's maximum number of intersections to one-kth of the original. To achieve mutability in LibRTS, we use a technique called *Instancing*. Instead of building a single monolithic BVH, we place newly inserted geometries into separate BVHs and then organize these BVHs into an instance acceleration structure.

Our contributions are summarized as follows:

1. We propose a method to convert range and point queries into RT-friendly problems, allowing them to effectively utilize RT cores (§3.1, §3.2, and §3.3).

2. We are the first to identify and address the load balancing challenge that arises when repurposing RT cores (§3.4).

3. By leveraging the *Instancing* feature, we enable dynamic insertion and deletion of geometries despite limited BVH update support, thus providing mutability for our spatial index (§4).

4. LibRTS is an open-source library offering a user-friendly interface that simplifies the integration of RT cores into spatial data processing applications (§5).

2 Background

2.1 Spatial Index

A spatial index is designed to accelerate queries over geometries, including points, lines, and polygons. Most spatial indexes support range queries, using bounding boxes to retrieve geometries that satisfy a given predicate such as $Contains$ or $Intersects$. For example, we create a spatial index over a set of rectangles R representing building boundaries. Given another set of rectangles S representing flood zones, we can quickly identify buildings at risk of flooding using the predicate $Intersects(r, s)$, where $r \in R$ and $s \in S$.

The primary design goal of spatial indexes is to filter out irrelevant geometries by reducing the search space. K-D trees [10, 11, 21, 73], Quadtrees [20, 71], and grids [1, 2] partition space into non-overlapping subspaces. They are most suitable for indexing geometries without extents, such as points. For indexing complex geometries, such as rectangles or polygons, it is more efficient to use geometry-partitioned indexes, such as R-trees [8, 24, 39, 55, 68–70] and BVHs [15, 23, 43, 57]. Both R-trees and BVHs use bounding boxes to organize geometries. R-trees are widely used in the geospatial community due to their dynamic update capabilities, whereas BVHs are preferred in the graphics community for their fast index construction and efficient handling of ray-primitive intersection queries.

Tree-based indexes, though algorithmically efficient due to their hierarchical structure, can be inefficient on GPUs because of random memory access and branch divergence [5, 6]. In contrast, grid-based indexes map geometries to cells with linear memory space, improving memory efficiency but struggling with skewed data [4, 59, 60]. Consequently, despite their hardware inefficiencies, tree-based indexes remain widely adopted in spatial systems.

A new class of learned spatial indexes learns a cumulative distribution function (CDF) derived from geometry data, enabling constant-time inference of qualifying geometries [16, 31, 35, 36, 56]. However, most learned spatial indexes are designed for disk-based predictions and merely identify the disk pages containing the queried results [34, 56]. Furthermore, training a learned index is time-consuming [37], and few such indexes support geometries with extents [3, 62].

2.2 Ray Tracing

Ray tracing is a technique used to render photorealistic images by simulating light transportation in a scene. It calculates how much radiant energy emitted by light sources reaches the viewer's eyes after reflecting off various surfaces [67]. A crucial step in this process is casting rays from the viewer's perspective into the scene to determine how they interact with the primitives they encounter.

Ray. A ray is defined by a half-line parameterized by an origin O, a direction vector $\vec{d}$, and a parameter t that controls its extent. This definition can be formalized by Equation (1):

$$\mathbb{R}(t) = O + t \cdot \vec{d}, \text{where } t \geq 0 \tag{1}$$

For a given ray, O and $\vec{d}$ are fixed, with t being the variable parameter that controls the distance the ray travels.

Primitive. In ray-tracing rendering, a primitive is a fundamental geometric shape, such as a triangle, sphere, or Axis-Aligned Bounding Box (AABB) that serves as a building block for composing more complex models.

Ray-Primitive Intersection. If the ray intersects a primitive at $t = t_{hit}$, an intersection point at $\mathbb{R}(t_{hit})$ is identified. The parameter t at the intersection point can be calculated by solving the common solution of the ray and the primitive equations. It is often desirable to consider intersections within a certain range, as intersections too close to or too far from the ray's origin may have negligible effects on rendering results. To accommodate this, ray tracing frameworks typically require a search interval $[t_{min}, t_{max}]$ when casting a ray and retain the intersection only if $t_{min} \leq t_{hit} \leq t_{max}$.

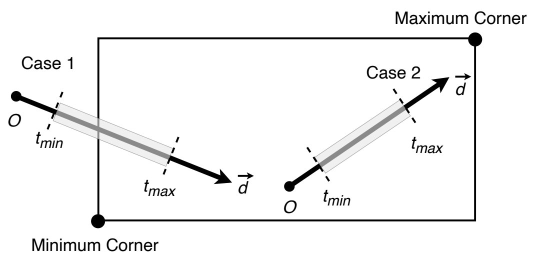


Figure 1. Two valid ray-AABB intersection cases: ray hits the AABB or ray origin is within the AABB

The AABB is an important geometric construct because it can enclose any complex shape. Most ray tracing frameworks, such as NVIDIA OptiX, support using AABBs to implement customized primitives. For ease of demonstration, Figure 1 shows an AABB in 2-D dimensions, defined by its minimum corner (x_{min}, y_{min}) and maximum corner (x_{max}, y_{max}). The ray-AABB intersection test is then performed to identify all potential ray-primitive intersections. As shown in Figure 1, there are two scenarios that qualify as ray-AABB intersections.

Case 1: The origin O is outside an AABB, and the ray intersects the boundaries of the AABB at $\mathbb{R}(t_{hit})$ such that $t_{min} \leq t_{hit} \leq t_{max}$.

Case 2: The origin O is inside an AABB, regardless of whether the ray intersects the AABB.

2.3 Acceleration Structure

Ray tracing rendering requires a data structure to narrow down the primitives that intersect with rays. The BVH tree is predominantly used as an Acceleration Structure (AS) in ray tracing rendering.

Figure 2(a) shows a scene with three triangles and two rays, Ray_1 and Ray_2, illustrating how a BVH accelerates intersection tests. A BVH is a tree where each node N_i is associated with an AABB B_i that encloses the AABBs of its children. For example, B_2 encloses B_4 and B_5, while B_1 encloses all of the other bounding boxes. If Ray_1 intersects B_1, we then check its children (N_2 and N_3) for ray–AABB intersections. Since Ray_1 does not intersect B_2, the child nodes of N_2 are skipped entirely. Subsequently, upon finding that Ray_1 intersects B_3, the triangle inside B_3 is tested, completing the traversal. In contrast, Ray_2 must visit every node in the BVH because it intersects B_2.

In rendering, a complex scene comprises many geometric components, some of which may be identical but positioned differently. For example, Figure 2(b) shows a scene with two helicopters and a plane. Each model has its own coordinate system, known as the local coordinate system. To integrate the models into the scene, a Scale-Rotate-Translate (SRT) transformation is applied to transform the model from the local coordinate system to the world coordinate system. The SRT transformation is usually represented by a 3x4 row-major object-to-world matrix that defines how to scale, rotate, and move a model.

Since a model may appear multiple times in different positions, creating a BVH for each instance is inefficient. The technique called *Instancing* addresses this by allowing a single BVH to be created for a model. To achieve *Instancing*, two types of acceleration structures are used: the Geometry Acceleration Structure (GAS) and the Instance Acceleration Structure (IAS). The BVH serves as the GAS for the model, while the IAS links the GAS with an SRT matrix. Therefore, it is only necessary to create multiple links that connect the IAS to the GAS using different SRT matrices to reuse the same model, resulting in a traversal graph.

Figure 2(c) shows the traversal graph of the scene. The root node is the IAS, which links to *GAS*#1 twice using different SRT matrices. Building the IAS requires a reference to the GAS and an SRT matrix. Since the IAS does not store the primitives, building an IAS is lightweight and very fast. With the traversal graph, the ray will be transformed by the SRT matrix and redirected to the GAS to find the intersections.

2.4 Programming Model for RT Cores

RT cores, introduced with the Turing architecture [13], provide dedicated hardware units for ray-primitive intersection and BVH traversal. Compared to a software-emulated BVH search, which requires thousands of instruction slots per ray, Turing GPUs achieve up to a 10x speedup by offloading tree traversal and ray-primitive intersection tests to the specialized RT cores [50]. Additionally, the ray-primitive performance has doubled with each new generation of GPUs, offering "free" performance gains for RT-enabled applications [50, 51].

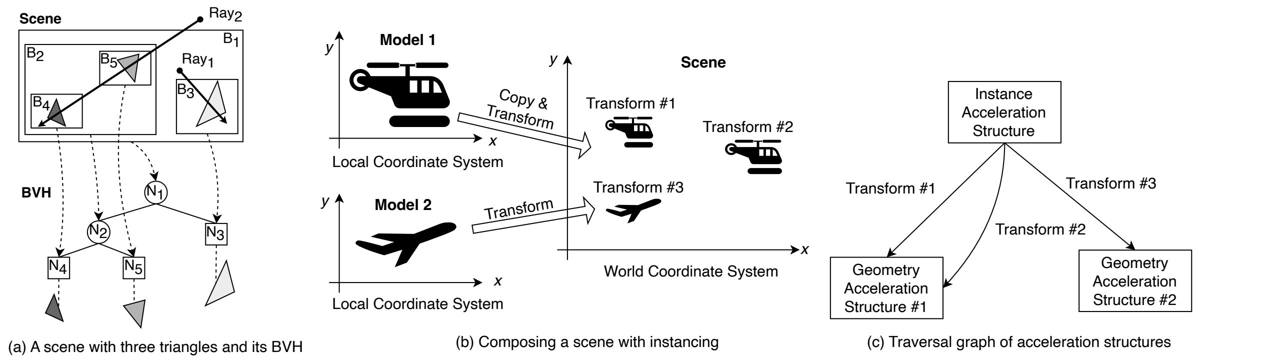

Scene
Ray2
B1
B2
B5
Ray1
B3
B4
BVH
N1
N2
N3
N4
N5
(a) A scene with three triangles and its BVH
Model 1
Copy & Transform
Transform #1
Transform #2
Scene
Local Coordinate System
Model 2
Transform
Transform #3
World Coordinate System
Local Coordinate System
(b) Composing a scene with instancing
Instance Acceleration Structure
Transform #1
Transform #2
Transform #3
Geometry Acceleration Structure #1
Geometry Acceleration Structure #2
(c) Traversal graph of acceleration structures

Figure 2. (a) A scene with three triangles and its BVH. Casting two rays Ray_1 and Ray_2 to find which triangles are being hit by the ray; (b) a scene that comes from two models. Each model has its own coordinate system. By transforming the models, a complex scene can be composed; (c) an example of a traversal graph of a composed acceleration structure for the scene in (b).

NVIDIA OptiX [53] allows users to develop RT programs using a subset of the CUDA language. OptiX requires users to implement several callback functions, known as shaders in CUDA with certain restrictions, such as the unavailability of shared memory and block synchronization instructions. Imposing these restrictions allows OptiX to freely reschedule a ray at any point in time to other lane, thread block or even Streaming Multiprocessor (SM) for efficiency and coherence [53]. OptiX employs a single-ray programming model, where the shaders are executed on a single thread for each ray.

BVH construction is a prerequisite for ray tracing. To build the BVH, the user provides an array of built-in primitives, such as triangles, spheres, curves, or customized AABBs in 3-D Euclidean space. The BVH structure and construction algorithm are opaque and managed by the video driver. Upon completion of the BVH construction, a traversal handle is returned to access the BVH. OptiX allows the user to update the coordinates of primitives in a constructed BVH, called BVH refitting. Recent research shows that updating a BVH is up to three times faster than rebuilding [26]. However, inserting or deleting from a BVH is not supported. In addition to BVH construction, the user must implement a series of shaders to cast rays and handle intersections.

RayGen (RG) shader is the entry point of an RT program, where rays are cast, transferring the control from SMs to RT cores.

IsIntersection (IS) shader is invoked if a ray potentially[2] hits an AABB. This shader is available only when using AABB as a primitive, allowing the user to check whether a ray hits the geometry enclosed by the AABB.

AnyHit (AH) shader is invoked if a ray hits any primitive.

ClosestHit (CH) shader is invoked if a ray hits the primitive closest to the ray origin.

Miss (MS) shader is invoked when the traversal is finished if a ray does not hit any primitive.

[2]The invocation of the *IS* shader does not always indicate the ray hits an AABB.

3 From Spatial Query to Ray-tracing

This section explains how to formulate point queries and range queries using the *Contains* and *Intersects* predicates as ray tracing problems. Additionally, repurposing RT cores for spatial queries introduces a load imbalance challenge. To address this, we present a technique called *Ray Multicast* to balance workloads across threads. For simplicity, the methods are described in 2D, but extending them to 3D is straightforward since OptiX operates natively in 3D space.

3.1 Point Query

Given a set of rectangles R and query points S, a point query $Q(R, S)$ returns a list of pairs (r, s) such that $r \in R, s \in S$, and the predicate $Contains(r, s)$ is true. Definition 1 specifies the $Contains$ predicate, where evaluates to true if the point s lies within the rectangle r. Figure 3 illustrates the point query, where r_1 to r_3 represent the rectangles and s_1 to s_4 denote the query points. In this example, the query point s_2 is within r_1 and r_2, while s_4 is within r_3. Therefore, the query result is $\{(r_1, s_2), (r_2, s_2), (r_3, s_4)\}$.

Definition 1 (Rectangle-Point Containment)**.** *For a given point $p = (x, y)$ and a rectangle $r = \{(x_{min}, y_{min}), (x_{max}, y_{max})\}$, predicate $Contains(r, p)$ is true if $x_{min} \leq x \leq x_{max} \wedge y_{min} \leq y \leq y_{max}$, otherwise, $Contains(r, p)$ is false.*

The idea of accelerating point queries on RT cores involves simulating a point with a short ray. As introduced in §2.2, a segment of a ray is defined by specifying a search range $[t_{min}, t_{max}]$. A point can be simulated using a very short ray by appropriately setting this range. We then build a BVH from the rectangles R, using their AABBs as primitives. For each point $s \in S$, a short ray is cast, with the ray's origin representing the point's location. If the point lies within an AABB, the ray-AABB intersection shader will detect it. These steps are detailed below.

BVH Construction. For each rectangle $r \in R$ on the 2-D plane, an AABB is created that has the same x and y coordinates as r. Since OptiX defines all primitives in 3-D space, we set the z-coordinate to zero. These AABBs are then organized in an array and processed by OptiX, where the BVH is automatically constructed.

Ray Casting. Recall that there are two ray-AABB intersection cases (§2.2). For each point query, we cast a ray from the point as its origin with a small t_{max} value, and the direction can be arbitrary. The occurrence of *Case 2* explicitly indicates that the query point falls within an AABB. As depicted in Figure 3, each point s_i casts a short ray where the ray origin O has the same xy-coordinate as s_i. Point s_2 falls inside r_1 and r_2, resulting in two ray-AABB intersections. Despite not being contained by any rectangle, point s_1 still counts as an intersection due to *Case 1*. This scenario, termed a false positive hit, is mitigated by setting t_{max} to the smallest representable floating point number, such as `FLT_MIN`.

Result Collection. Whenever *Case 1* or *Case 2* occurs, the *IS* shader will be triggered. The *IS* shader allows the user to obtain the ray origin and the index of the primitive being hit by the ray. This information allows us to evaluate the $Contains$ predicate to filter out the occurrence of *Case 1*. Therefore, in the *IS* shader, we filter out the false positives by evaluating the predicate and insert a pair of (r_i, s_i) into a result queue.

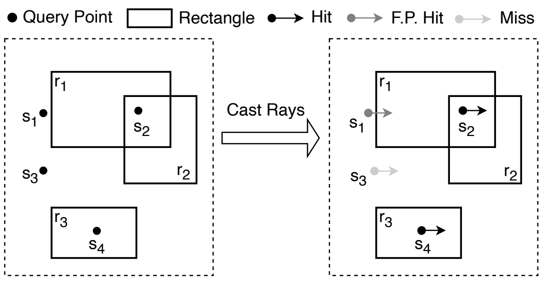


Figure 3. Casting rays from the query points results in three intersection cases: a hit (black); a False Positive (F.P.) hit (grey), and a miss (light grey).

3.2 Range Query with *Contains* Predicate

Given two sets of rectangles R and S, a range query $Q(R, S)$ employs a *Contains* predicate (referred to as *Range-Contains*) to return a list of rectangle pairs (r, s) such that $r \in R$, $s \in S$, and $Contains(r, s)$ is true (see Definition 2).

Definition 2 (Rectangle-Rectangle Containment). *Given two rectangles r_1 and r_2, predicate $Contains(r_1, r_2)$ is true if $r_1.x_{min} \le r_2.x_{min} < r_2.x_{max} \le r_1.x_{max} \wedge r_1.y_{min} \le r_2.y_{min} < r_2.y_{max} \le r_1.y_{max}$, otherwise, $Contains(r_1, r_2)$ is false.*

The RT-accelerated *Range-Contains* method is based on the observation that if $Contains(r, s)$ is true, then the center point s_c of rectangle s must also lie within r, i.e., $Contains(r, s_c)$ is true, where $s_c = \left(\frac{s.x_{min}+s.x_{max}}{2}, \frac{s.y_{min}+s.y_{max}}{2}\right)$. In turn, point s_c is within rectangle r does not always mean r contains s. Based on this, the range query can be reduced to a point query (§3.1) that identifies a list of candidate rectangle-point pairs (r, s_c). Finally, the $Contains$ predicate is evaluated over this list to filter the pairs (r, s) where $Contains(r, s)$ is true.

3.3 Range Query with *Intersects* Predicate

Give two sets of rectangles R and S, a range query with *Intersects* predicate (referred to as *Range-Intersects*) returns all the pairs of rectangles (r, s) such that $Intersects(r, s)$ is true (Definition 3), where $r \in R$ and $s \in S$.

Definition 3 (Rectangle-Rectangle Intersection). *Given two rectangles r_1 and r_2, $Intersects(r_1, r_2)$ is true when $r_1.x_{min} \le r_2.x_{max} \wedge r_1.x_{max} \ge r_2.x_{min} \wedge r_1.y_{min} \le r_2.y_{max} \wedge r_1.y_{max} \ge r_2.y_{min}$, otherwise it is false.*

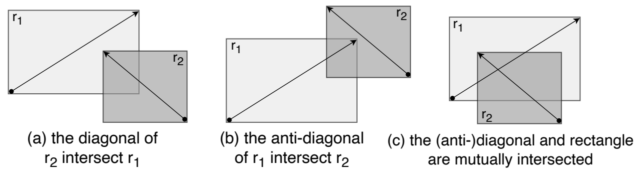

r1
r2
(a) the diagonal of r2 intersect r1
r1
r2
(b) the anti-diagonal of r1 intersect r2
r1
r2
(c) the (anti-)diagonal and rectangle are mutually intersected

Figure 4. Three cases of (anti-)diagonal-rectangle intersections that imply the two rectangles intersect.

A potential approach to formulating the *Range-Intersects* query as an RT problem involves casting rays from the four corners of rectangles R and S, effectively reducing the range query to a point query once again. However, this method generates duplicate query results, as multiple corners may intersect the same rectangle, and deduplication can be computationally expensive. To address this issue, we propose a new formulation technique that can effectively eliminate the duplicated results by transforming the query into a rectangle-diagonal intersection problem.

An important observation regarding the $Intersects$ predicate is that if $Intersects(r_1, r_2)$ is true, then either the diagonal of r_2 intersects r_1 or the anti-diagonal of r_1 intersects r_2 or both cases are true, as demonstrated in Figure 4(a)-(c), respectively. We formally define the diagonal and anti-diagonal of a rectangle in Definition 4.

Definition 4 (Diagonal and Anti-diagonal). *For a rectangle r, its diagonal D_r is a line segment defined by two endpoints (x_{min}, y_{max}) and (x_{max}, y_{min}). The anti-diagonal $\hat{D}_r$ is defined by (x_{min}, y_{min}) and (x_{max}, y_{max}).*

In Definition 5, we define the intersection between a rectangle and its diagonal, which will be used in Theorem 1.

Definition 5 (Rectangle-Line Segment Intersection). *For a given rectangle r and a line segment l, the $Intersects(r, l)$ is true if l intersects any boundary of r.*

Based on the above observation, the $Intersects$ predicate for two rectangles can be transformed into equivalent tests for the intersections of their diagonals, as stated in Theorem 1[3]. As previously noted, a (anti-)diagonal is essentially a line

[3]The proof of the theorem is provided in the supplementary material.

segment, which can be represented as a ray. Therefore, the range query using the *Intersects* predicate can be reformulated into two passes of ray-AABB intersection tests, referred to as *Forward Casting* and *Backward Casting*.

Theorem 1. *Given two rectangles r_1 and r_2 that do not contain each other ($Contains(r_1, r_2)$ is false and $Contains(r_2, r_1)$ is false), if $Intersects(r_2, \hat{D}_{r_1})$ is true or $Intersects(r_1, D_{r_2})$ (Definition 5) is true $\iff$ $Intersects(r_1, r_2)$ is true.*

Forward Casting involves casting rays from the diagonals of S to detect intersections with R. Conversely, *Backward Casting* involves casting rays from the anti-diagonals of R to detect intersections with S. Finally, the results of these two ray-casting passes are merged to produce the final intersection results.

BVH Construction. Differing from the previous queries, the $Range - Intersect$ requires two BVHs for the rectangles R and S, respectively, to support the two ray casting passes.

Ray Casting. Since a ray is a semi-infinite line, we can use a portion of it to simulate a (anti-)diagonal by setting the search range parameters t_{min} and t_{max} (§2.2). For a diagonal defined by two endpoints p_1 and p_2, the ray parameters can be derived as follows [22].

$$\begin{gathered}\text{Ray origin } O = p_1 \\ \text{Ray direction } \vec{d} = p_2 - p_1 \\ t_{min} = 0, \text{and } t_{max} = 1\end{gathered} \tag{2}$$

We can verify the correctness of these parameters by substituting them into Equation (1). Specifically, we have $\mathbb{R}(t) = O+t\cdot\vec{d} = p_1+t\cdot(p_2-p_1)$. When $t_{min} = 0, \mathbb{R}(t_{min}) = p_1$, representing one endpoint of the diagonal. Similarly, when $t_{max} = 1, \mathbb{R}(t_{max}) = p_2$, representing the other endpoint of the diagonal. Thus, the ray effectively captures all intersections along the diagonal.

Note that Theorem 1 has a precondition that rectangles r_1 and r_2 cannot mutually contain each other. However, Definition 3 does consider the case of rectangle containment. Our method still works in this scenario. If r_1 contains r_2, the ray cast from the diagonal of r_2 will intersect r_1 because the origin of r_2 is within r_1 (§2.2, Case 2). The same logic applies if r_2 contains r_1. The time complexity is $O(R \cdot \log S + S \cdot \log R)$ due to the two passes of ray casting.

Result Collection. An intersection can be discovered in both *Forward Casting* and *Backward Casting*, as illustrated in Figure 4(c). To avoid duplicate results, we check whether the intersection can be discovered in both passes. If it can, we retain the intersection only in the forward casting pass.

Algorithm 1 demonstrates the implementation of *Forward Casting* process. The *RayGen* shader is the entry point of an RT program, where casting a ray along the diagonal of the query, with the origin and direction calculated based on the corner points of the query. In Line 9, `optixTrace` casts the ray, carrying the query rectangle ID. After that, the BVH will autonomously traversed on the RT cores guided by the ray. Whenever the traversal reaches to the leaves of the BVH, the *IS* shader will be invoked, allowing users to check whether the ray intersects an primitive, which is an AABB in our scenario.

Note that *IS* only reports potential intersections, so it can be invoked even if the ray does not intersect an AABB. Therefore, in Line 18, we explicitly check whether the diagonal of s intersects r. The diagonal-rectangle intersection test can be efficiently implemented using the well-known "Slab Method" [30, 54]. In Line 19, if an intersection with r is detected, we further verify whether the anti-diagonal of r intersects s to avoid duplicate results (see Figure 4(c)). *Backward Casting* is implemented in a similar way, but it omits the deduplication process.

Algorithm 1: FindIntersections - *Forward Casting*

```
   Input  : Indexed rectangles $R$ and queries $S$
   Input  : BVH traversal handle $h$ built over $R$
   Output : $X$ - Query Results
1  // Entrypoint, invoked when the program starts
2  procedure RayGen
3      $t_{min} = 0$
4      $t_{max} = 1$
5      // Casting rays from $S$
6      for each $s$ in $S$ do
7          $O = (s.x_{max}, s.y_{min})$        // Ray along diagonal
8          $\vec{d} = (s.x_{min}, s.y_{max}) - O$
9          $optixTrace(h, O, \vec{d}, t_{min}, t_{max}, payloads(id_s))$
10     end for
11 end procedure
12 // Invoked when ray potentially hits an AABB
13 procedure IsIntersection
14     $id_r = optixGetPrimitiveIndex()$
15     $id_s = optixGetPayload_0()$
16     $r = R[id_r]$
17     $s = S[id_s]$
18     if the diagonal of $s$ intersects $r$ then
19         if the anti-diagonal of $r$ does not intersect $s$ then
20             $X = X \cup (id_r, id_s)$
21         end if
22     end if
23 end procedure
```

3.4 Ray Multicast for Load Balancing

Recall that OptiX adopts a single-ray programming model (§2.4), meaning that shaders triggered by a single ray are executed by the same thread that casts the ray. Under this design, a load balancing issue arises when some threads handle very few intersections while others handle many intersections. We observed that this imbalance issue notably impacts the *Range-Intersects* query in the *Backward Casting* stage. Various techniques have been proposed to improve intra-GPU load balancing by reassigning workloads to warps or thread blocks [9, 45, 46, 63, 64]. Unfortunately, these techniques are not applicable to RT cores because OptiX does not support thread block or warp synchronization (§2.4). Additionally, the number of primitives a ray intersects is unknown beforehand, rendering workload rebalancing challenging.

To overcome these limitations, we designed a static load-balancing method called *Ray Multicast*. The idea is to evenly split N primitives into k sets, $N_1, N_2, \ldots, N_k$, and distribute them into k non-overlapping regions on the 2-D plane, referred to as sub-space. Then, a ray $\mathbb{R}$ is duplicated into k rays $\mathbb{R}_1, \mathbb{R}_2, \ldots, \mathbb{R}_k$, each responsible for finding intersections within one of the sub-spaces. Under the single-ray model, each thread still casts a single ray but the number of intersections per ray is no more than $\frac{|N|}{k}$. Casting k rays into the k sub-spaces with k threads captures all the intersections without duplications or omissions.

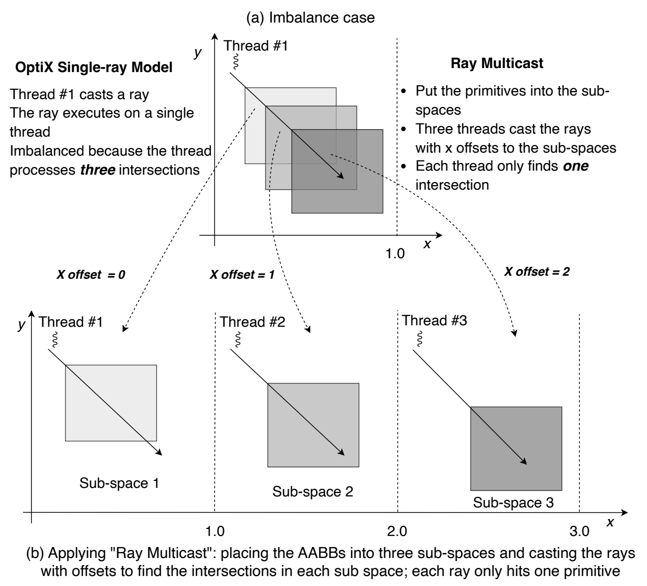

(a) Imbalance case
OptiX Single-ray Model
Thread #1 casts a ray
The ray executes on a single thread
Imbalanced because the thread processes three intersections
Thread #1
Ray Multicast
Put the primitives into the sub-spaces
Three threads cast the rays with x offsets to the sub-spaces
Each thread only finds one intersection
y
x
1.0
X offset = 0
X offset = 1
X offset = 2
Thread #1
Thread #2
Thread #3
Sub-space 1
Sub-space 2
Sub-space 3
1.0
2.0
3.0
(b) Applying "Ray Multicast": placing the AABBs into three sub-spaces and casting the rays with offsets to find the intersections in each sub space; each ray only hits one primitive

Figure 5. Load balancing with *Ray Multicast* technique

Figure 5 demonstrates how *Ray Multicast* works. Assuming we have three AABBs and $k = 3$. Figure 5(a) shows an imbalanced case in which a single ray hits all three AABBs. To construct non-overlapping subspaces, we scale the AABB coordinates to a range of 0 to 1 and place them into sub-spaces with $x \in (0, 1)$, $x \in (1, 2)$, and $x \in (2, 3)$ [4]. In Figure 5(b), three rays are cast with x-coordinates offset by 0, 1, and 2. Each ray therefore intersects only one AABB.

Determining Parameter k. *Ray Multicast* reduces the intersections to $\frac{|N|}{k}$ per thread at the cost of casting k times more rays. Therefore, we need to balance the workload per thread with the cost of ray casting. We use a cost-based model to determine the optimal k. The model estimates the costs in two parts: the ray casting cost (C_R) and the intersection cost (C_I). The total cost (C) is derived by weighting C_R and C_I (Equation (3)). The goal is to find a k to minimize cost C.

$$C = (1 - w) \cdot C_R + w \cdot C_I \tag{3}$$

Assume that there are $|R|$ rays to cast before using Ray Multicast, and that the search cost per ray is $O(\log|N|)$ for a BVH with $|N|$ primitives, as the BVH is a tree-based data structure. When applying *Ray Multicast*, the cost of casting rays is k times higher, as illustrated in Equation (4).

$$C_R = |R| \cdot k \cdot log(|N|) \tag{4}$$

The total number of intersections is $|N| \cdot |R| \cdot s$, where s is the selectivity indicating the percentage of returned intersections from $|N|$ primitives and $|R|$ rays. The intersection cost C_I is the total number of intersections divided by k because the shaders can process the "multicasted" rays in parallel, as shown in Equation (5).

$$C_I = \frac{|N| \cdot |R| \cdot s}{k} \tag{5}$$

To determine s, we use a sampling technique. By sampling a small portion of primitives and rays and performing a brute-force trial run, we can calculate the number of intersections with neglectable overhead. This helps estimate the number of intersections for the entire dataset. Finding the optimal k for the lowest cost C can be accomplished through an exhaustive search, as k must be a power of two for warp efficiency. The effectiveness and overhead of this parameter-finding technique are discussed in §6.5.

4 Update Spatial Index

4.1 Insert

To enable insertions, we avoid building a monolithic BVH for all geometries. Instead, we adopt a two-level acceleration structure. The bottom level consists of the BVHs for each batch of insertions serving as the GASs (§2.3). The top level is an IAS that links to the GASs with an identity SRT matrix. This method eliminates the need to rebuild the BVH for each insertion. Instead, we only need to incrementally build a new BVH and then rebuild the IAS. Rebuilding the IAS is fast, as it merely links the BVHs without storing the geometries.

The *IS* shader needs to be modified to support insertions. In the *IS* shader, `optixGetPrimitiveIndex` can be used to query which primitive is being hit by the ray. The primitive index is renumbered from zero for each BVH. Therefore, when a ray hits a primitive, we have to calculate the global primitive index. We maintain a prefix sum array A for every batched insertion, where $A[i] = A[i-1] + I$ and I is the number of insertions in the i-th batch. OptiX also allows users to query the ID of the BVH with `optixGetInstanceId`. Using the prefix sum, BVH ID, and the primitive index, we can compute the global primitive index in $O(1)$ time.

[4]For the 2-D case, we can also put the geometries into subspaces by specifying the unused z-coordinate.

4.2 Update and Delete

OptiX provides a mechanism to update an acceleration structure, including both GAS and IAS, by specifying the new coordinates of primitives. The existing research shows that updating the BVH is more than three times faster than rebuilding it from scratch [26]. However, the quality of the BVH can degrade when the spatial location of the data changes significantly. Therefore, rebuilding could be necessary if query performance significantly degrades.

LibRTS relies on the updating support of OptiX. Users pass an array of new rectangles along with their IDs. This input is used to update an array of primitives cached in LibRTS. Using the provided IDs and new rectangles, the primitives are updated with their new coordinates, which are subsequently used to refit the BVH.

For deletions, we turn the rectangles being deleted into degenerate cases. For example, we set the x_{min} and x_{max} of AABBs to the same value. Then, we refit the BVH with the updated primitives, just as we do during an update. This setting reduces the extent of the AABB to zero, preventing them from being found by ray casting.

5 Implementation and Interface for Users

LibRTS is a header-only library with additional OptiX shaders that can be easily embedded into users' programs. Users only need to implement a handler to process the query results in a C++ header file. This header file is then included in the OptiX shaders, which are subsequently compiled into Parallel Thread Execution (PTX) code. The PTX code will be loaded during the initialization of LibRTS to create the rendering pipeline.

Algorithm 2 lists the interfaces of LibRTS. To construct an instance of LibRTS, the class requires two template parameters, `COORD_T` which can be *float* or *double*, and `N_DIMS`, which is the number of dimensions of spatial data (2 or 3). To run a query, users need to pass a predicate p, a pointer to the queries in device memory, and the number of queries n. Subsequently, `RTSIndex_handler` will be invoked to handle the query results. LibRTS also includes two built-in handlers, called the "Counting Handler" and the "Collecting Handler," which are responsible for counting the number of query results and storing the query results, respectively.

Algorithm 2: API Summary and an Example

```
template <typename COORD_T, int N_DIMS>
class RTSIndex {
    // Load PTX and initialize the rendering pipeline
    void Init(const char * ptx_root);
    // Run spatial queries on RT cores
    template <typename QUERY_T>
    void Query(Predicate p, QUERY_T *queries, int n, void * arg,
      cudaStream_t stream);
    // Insert new rectangles into the index
    void Insert(rect_t *rectangles, int n, cudaStream_t stream);
    // Delete the rectangles by the IDs
    void Delete(int *ids, int n, cudaStream_t stream);
    // Update the coordinates of the rectangles
    void Update(rect_t *rectangles, int *ids, int n, cudaStream_t
      stream);
}
// Invoked when found qualified results
// "arg" allows users to pass their parameters to the
    handler
__device__ void RTSIndex_handler(Predicate p, void *
  arg, int rect_id, int query_id) {
    // Example: collect results with a queue
    static_cast <Queue*>(arg)->Append(rect_id, query_id);
}
```

6 Evaluation

6.1 Evaluation Setup

Baselines. We endeavor to include the most competitive baselines on both CPUs and GPUs, as listed in Table 1. Building on the work by Lawson et al. [33], who evaluated 20 open-source spatial libraries and concluded that no single solution is optimal for every query—while highlighting CGAL [14] and Boost [12] for their overall performance. Therefore, we include these libraries. We also evaluate ParGeo, a computational geometry library optimized for multicore environments [65, 66]. Recognizing the growing trend of learned-based indexes, we also consider them. However, most learned spatial indexes are limited to point indexing, whereas LibRTS supports indexing geometries with extents. To the best of our knowledge, GLIN [62] is the only learned spatial index capable of handling complex geometries, and thus, we include it in our evaluation.

Ideally, to demonstrate the benefits of LibRTS of hardware-based RT cores, we would compare the performance numbers with enabled and disabled RT cores. However, OptiX does not allow users to manually disable hardware acceleration. To address this, we include Linear BVH (LBVH), a software-emulated BVH on GPUs [28], to illustrate that LibRTS indeed benefits from hardware acceleration. We further assess a real-world application, the Point in Polygon (PIP) test. cuSpatial [52] employs an Octree to accelerate PIP queries. Since a rectangle is a special type of polygon, cuSpatial also supports point queries. Finally, we include RayJoin [22], a state-of-the-art spatial join algorithm designed for RT cores.

Environment. We evaluate the GPU-based libraries on an NVIDIA RTX 3090 and the CPU-based libraries on an HPC server equipped with two AMD EPYC 7713 CPUs (128 cores in total). The cuSpatial version used is 24.08, while LibRTS is implemented with OptiX 8.0 and CUDA 12.3.

Since spatial queries are read-only, distributing them across multiple cores is straightforward. Therefore, for CPU-based baselines, we evenly distribute all queries across all CPU

Table 1. Summaries of the artifacts evaluated in the paper

Artifact	Index Type	Query Type	Platform
Boost [12]	R-Tree	Point, Range	CPU
CGAL [14]	KD-Tree	Point	CPU
ParGeo [65, 66]	KD-Tree	Point	CPU
GLIN [62]	Learned Index	Range	CPU
LBVH [28]	Linear BVH	Point, Range	GPU
cuSpatial [52]	Octree	Point	GPU
RayJoin [22]	BVH on RT cores	PIP	GPU
LibRTS	BVH on RT cores	Point, Range, PIP	GPU

cores to ensure a fair comparison. Lastly, due to the limited FP64 units in RTX GPUs, we implement the queries using FP32 precision.

Datasets. Table 2 shows the real-world geospatial datasets collected from ArcGIS Hub [18] and OpenStreetMap [17]. These datasets are in the form of polygons, for which we create rectangles to enclose the polygons as the input for spatial indexes. We also sampled a subset of OSMParks from the European continent, named EUParks, as a moderately large dataset to accommodate the baseline RayJoin, which runs out of memory when processing the full-scale OSM datasets. Additionally, we use Spider to generate synthetic datasets to evaluate the scalability of LibRTS [29].

Table 2. Real-world datasets collected from ArgGIS Hub and OpenStreetMap.

Dataset	Polygons	Description
USCounty	12.2K	Boundaries of the U.S. Counties
USCensus	248.9 K	U.S. Census block groups
USWater	463.6 K	Boundaries of U.S. water resources
EUParks	1.9 M	Parks and green areas in Europe
OSMLakes	8.3 M	Boundaries of water areas worldwide
OSMParks	11.5 M	Parks and green areas worldwide

Queries. The queries are generated to return a given ratio of the rectangles, which is a well-adopted evaluation method [33, 48, 62]. For the point and *Range-Contains* queries, we ensure each query falls within at least one rectangle. For the *Range-Intersects* query, we generate queries with selectivity levels of 0.01%, 0.1%, and 1%.

Timing. Index construction time is excluded from all query-related experiments, except those presented in §6.9. Another exception applies to the *Range-Intersects* query of LibRTS, which requires constructing a BVH for incoming queries. Since the BVH cannot be pre-determined until the queries are available, its construction time is included in the overall query time to ensure a fair comparison.

6.2 Point Query

From Figure 6(a), we observe that Boost R-Tree performs the best among the CPU-based libraries, except on *EUParks* dataset, where CGAL surpasses Boost. LibRTS outperforms the best CPU baseline by factors ranging from 74.4x (*USAWater*) to 302.1x (*OSMParks*). Although cuSpatial is a GPU-based library, it performs the worst among all baselines. LBVH ranks as the second-best library, benefiting from the advantages of large memory bandwidth and massive parallelism of the GPU. However, LibRTS remains significantly faster than LBVH, demonstrating the advantages of leveraging the hardware-based BVH accelerator - RT cores, achieving speedups of up to 85.1x over LBVH on *OSMLake*.

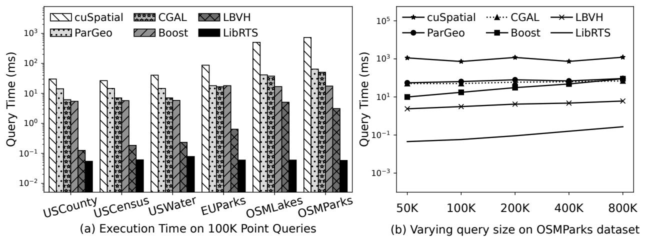


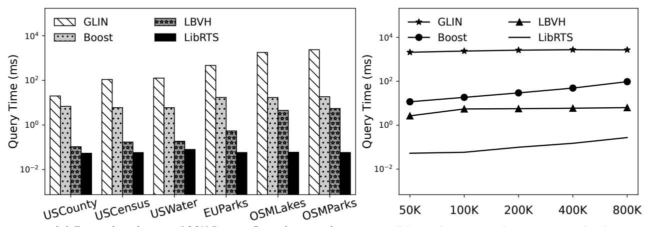
Figure 6. (a) Query time of 100K point queries; (b) Query time by varying the number of queries on OSMParks dataset

Figure 6(b) illustrates how query time varies as the number of queries increases. The three point-based indexes - CGAL, ParGeo, and cuSpatial - exhibit nearly constant search times because they index the query points. In contrast, the running times of the baselines that index rectangles, including LibRTS, increase linearly with the number of query points. Consequently, the performance gap between these two indexing strategies narrows. Nonetheless, LibRTS consistently outperforms all baselines, even for a very large number of queries.

6.3 Range Query with Contains Predicate

Figure 7(a) illustrates the execution time of the *Range-Contains* query. Among the baselines, the learned index GLIN exhibits the longest runtime, followed by the Boost R-Tree. The GPU-based LBVH outperforms Boost by an order of magnitude on the first four smaller datasets. However, when querying the full-scale OSM datasets, LBVH achieves only a 3x speedup over Boost. This is because traversing large datasets generates substantial memory traffic, which exposes the inefficiencies of software-based tree traversal. In contrast, LibRTS leverages hardware-based RT cores, delivering significant performance gains. It achieves speedups ranging from 1.9x on the USCounty dataset to 94.0x on the OSMParks dataset compared to LBVH.

Figure 7(b) illustrates the trends in query time as the query size varies. When the number of queries increases from 50K to 800K, Boost and LibRTS experience a growth in query time of 8.2x and 5.2x, respectively. In contrast, GLIN and LBVH exhibit minimal sensitivity to the increase in queries, with their query times rising by only about 1.3 and 2.4x, respectively. Despite handling more queries, LibRTS consistently outperforms all the baselines, demonstrating its scalability in handling larger query volumes.



(a) Execution time on 100K Range-Contains queries (b) Varying query size on OSMParks dataset

Figure 7. (a) Query time of 100K range queries with the *Contains* predicate; (b) Query time by varying the number of queries on OSMParks dataset

6.4 Range Query with Intersects Predicate

Range-Intersects is the most computationally expensive spatial query due to its intensive index traversal. To avoid exhausting memory by storing query results, we set the number of queries to 10K. Figure 8(a)-(c) shows the query times for different selectivity levels. As shown in Figure 8(a), LBVH is significantly faster than Boost for small datasets. For example, with a selectivity of 0.01%, LBVH achieves speedups of 7.3x, 5.4x, and 3.5x over Boost on USCounty, USCensus, and USWater datasets, respectively. However, for EUParks dataset, the running times of Boost and LBVH are nearly identical, although they are running on different hardware platforms. For larger datasets such as OSMLakes and OSMParks, LBVH underperforms compared to the Boost R-tree. The performance disparity arises because a significant portion of the index must be traversed for larger datasets or high-selectivity queries. In such scenarios, the software-implemented LBVH suffers from random memory access patterns and branch divergence caused by excessive tree traversal, leading to suboptimal performance. In contrast, LibRTS mitigates these issues by leveraging dedicated RT cores [25], resulting in speedups of 1.3x to 2.3x over the best baseline performance. As selectivity increases, the performance gap between LibRTS and the baselines widens (Figures 8(b)-(c)). At a selectivity of 0.1%, LibRTS achieves speedups of up to 6.8x (USWater). When selectivity increases to 1%, LibRTS delivers speedups up to 11.0x (USCensus).

Figure 8(d) illustrates query times as the number of queries increases under a selectivity of 0.1%. Initially, Boost and LBVH exhibit comparable performance for 10K queries. As the number of queries increases, LBVH surpasses Boost in performance. However, LibRTS consistently outperforms all the baselines, maintaining its performance advantage even as the query count increases.

6.5 Effectiveness of Load Balacing

In §3.4, we introduce the *Ray Multicast* technique for improved load balancing, which requires the parameter k to find the lowest ray casting cost. Figure 9(a) shows query times for *Range-Intersects* queries with varying k. As k increases, the query times decrease due to the intersection processing is evenly distributed to k threads. For example, LibRTS takes 24.26ms on `USCensus` when $k = 1$. When increasing k from 2 to 16, the running time consistently decreases from 11.9ms to 3.1ms, resulting in a 7.8x speedup compared to not using load balancing. When k is greater than 16, the overhead of casting rays offsets the benefits of more balanced workloads. In Figure 9(a), the red circles represent the predicted parameters by our cost-based model. For example, the model predicates k should be 32 for `USCensus` dataset, which is very close to the optimal k. For OSMLakes and EUParks datasets, our method correctly infers the optimal k.

Figure 9(b) breaks down the total query time into four components, which shows that the majority of time is spent in the backward cast stage. As discussed in §3.4, our prediction algorithm samples a small portion of datasets to estimate selectivity, which is then used to determine the optimal k. During selectivity estimation, we employ a brute-force approach to calculate the number of intersections on the samples. Therefore, the cost of this estimation depends only on the number of geometries and queries, independent of their distributions. To sum up, the prediction time is negligible compared to the total query time, highlighting the efficiency of our parameter-tuning technique.

6.6 Update Performance

Figure 10(a) shows the index construction time of the baselines and LibRTS. Note that all the CPU-based spatial indexes do not support parallel construction. LBVH builds the index on the GPU by sorting the geometries with their Morton codes. LibRTS uses the BVH construction provided by OptiX. Interestingly, although GLIN is the least performant baseline, it has a significantly lower buildup cost than Boost R-tree and even LBVH for large datasets. LBVH is 1.4x faster than LibRTS on `USCounty` dataset. While LibRTS is 3.7x to 4.5x faster than LBVH on the larger datasets, indicating OptiX has a better construction implementation. Figure 10(b) shows the number of insertions and deletions completed in one second with different batch sizes. For a 1K batch, LibRTS achieves 1.4M insertions per second and 49.5M deletions per second. Deletion is fast because it only degenerates rectangles to exclude them from searches and then refits the IAS. As batch size increases, the throughput significantly improves due to better GPU utilization.

6.7 Sensitivity to Updates

Although OptiX supports updating BVH, query performance can degrade due to suboptimal geometry subdivision compared to building the BVH from scratch. We evaluated the performance under different update ratios when updating both the size and position of geometries: (1) Moving rectangles along the x and y axes, (2) Enlarging rectangles up to 10 times, and (3) Shrinking rectangles approaching zero. Using the largest real-world dataset `EuropeParks`, we measured query times with an increasing update ratio.

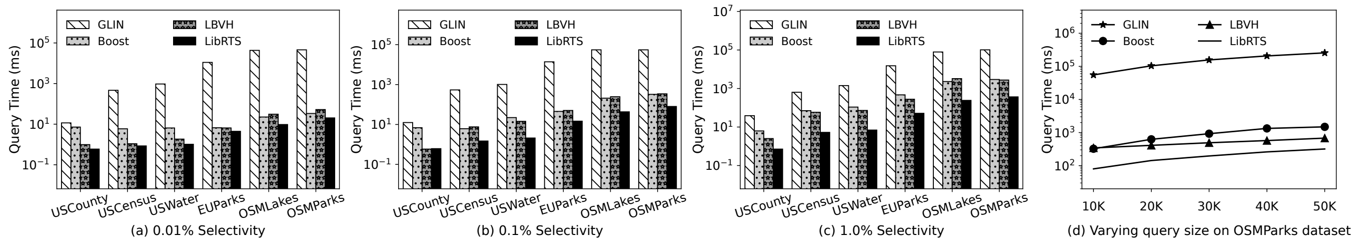


Figure 8. Query time of 10K *Range-Intersects* queries by varying selectivity.

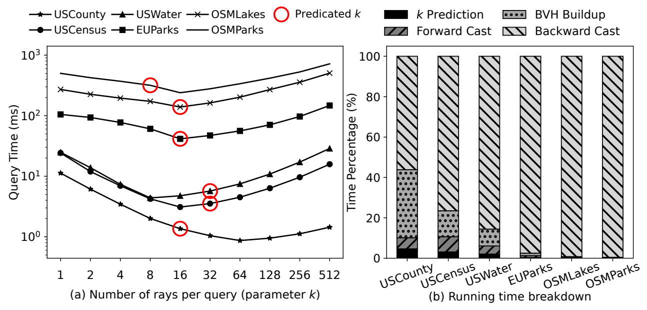


Figure 9. Demonstration of the effectiveness of *Ray Multicast* on *Range-Intersects* 50K queries and a selectivity 0.1%. (a) The changing of query times by varying the number of rays per query, with red circles indicating the predicted number of rays by our method; (b) Query time breakdown

Figure 10(c) shows the performance slowdown of an updated BVH compared to a freshly built BVH. Updating 0.02% of rectangles results in a 44% slowdown for point query and a 39% slowdown for *Range-Contains* query, while the *Range-Intersects* query slows down by only 3%. *Range-Intersects* is less susceptible to updates because the query traverses a large portion of the BVH, so a less efficient BVH has little impact on performance. With a 0.2% update ratio, point and *Range-Contains* queries experience 2.3x and 2.4x slowdowns respectively, whereas the *Range-Intersects* slows down by 1.08x. Interestingly, from 2% to 20% update ratios, query performance does not further deteriorate. We found that the number of query results surges when the update ratio increases from 0.2% to 20%, indicating that the BVH is intensively traversed. Therefore, the heavily updated BVH with a suboptimal structure does not further deteriorate performance.

6.8 Scalability

To evaluate LibRTS's scalability, we built the index on more rectangles using Spider to generate synthetic datasets with uniform and Gaussian ($\mu = 0.5, \sigma = 0.1$) distributions [29]. Figure 11(a) shows that query times increase linearly with more rectangles. The running time includes both searching the BVH and storing results. Although BVH search time complexity is logarithmic, more rectangles lead to a linear increase in result size. For example, 10K point queries on 10M uniform rectangles return 9.7M query results in 4.2ms, while 50M rectangles return 48.6M results in 20.8ms. Both query time and result size increase about fivefold. Due to space limitation, we do not show *Range-Contains* query but the result is very similar to the point query. Figure 11(b) shows the *Range-Intersects* query for the two synthetic datasets. The Gaussian datasets have clustered rectangles and queries, producing more query results and taking longer time. However, the query times still increase linearly with the number of rectangles, demonstrating that LibRTS is scalable across different distributions of spatial datasets.

6.9 Real-world Application: PIP

Figure 12 presents the PIP performance of the three artifacts for 100K query points. cuSpatial constructs the index based on query points. Since RayJoin adopts a planar map format, it requires the construction of the BVH at the line segment level, which limits RayJoin's ability to process the full OSM datasets due to excessive memory consumption.

In contrast, LibRTS is a generic index capable of indexing polygons using bounding boxes. Due to less effective indexing, cuSpatial is significantly slower than the RT-based approaches. Although RayJoin outperforms LibRTS on the smaller `USCounty` dataset, LibRTS surpasses RayJoin on the three larger datasets, with speedups of 1.9x, 1.1x, and 3.8x, respectively. Notably, RayJoin's performance is hindered by the high cost of BVH construction, which accounts for up to 98.7% of its total runtime. This inefficiency arises because decomposing polygons into individual line segments exponentially increases the number of AABBs, leading to significant overhead in BVH construction for large datasets.

7 Related Work

This section discusses RT cores for non-rendering workloads.

Neighbor Search Evangelou et al. proposed an inverse mapping to formulate a radius search as a ray tracing problem [19]. RTNN extended the radius search to k Nearest Neighbor (kNN) search with query scheduling optimizations [74]. TrueKNN further supports kNN search with an arbitrary radius [49]. JUNO offers an approximate nearest neighbor search in high-dimensional space [38]. Arkade supports kNN search in non-Euclidean space [41]. These works focus on neighbor search and do not support range queries.

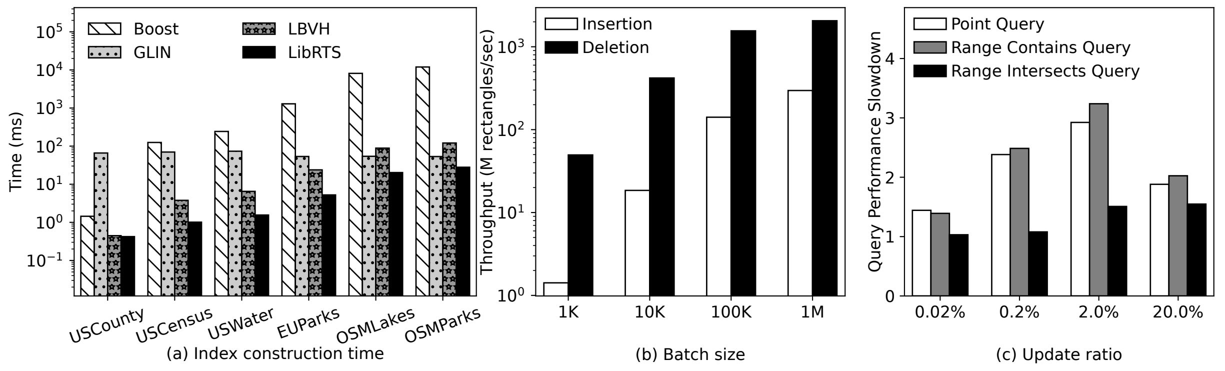


Figure 10. Index buildup costs, update rate by varying batch size, and query performance sensitivity to update ratios

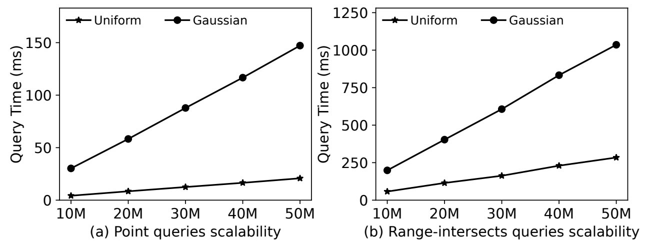


Figure 11. Scalability results. The number of queries is fixed to 10K for all the queries. The query times are reported by varying the number of rectangles indexed by LibRTS.

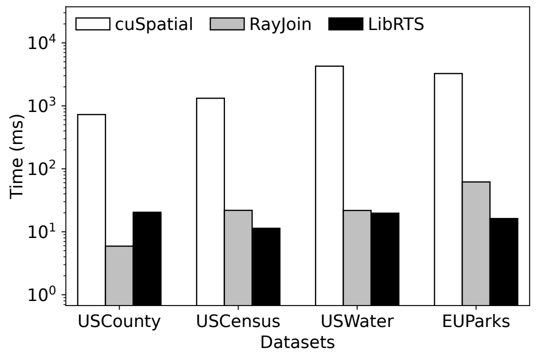


Figure 12. The end-to-end execution time of 100K point in polygon queries

Database Workloads RTIndeX turns RT cores into a B-tree [26]. RTScan leverages RT cores to accelerate index scans [40]. cgRX generalizes RTIndeX to achieve updateability and more performant queries [27]. RT-DBSCAN is an RT-based DBSCAN software [49]. Meneses et al. explore range minimum queries [44] on RT cores. These works focus on 1-D data, while spatial data is in 2-D/3-D space.

Spatial Workloads Wald et al. investigate using RT cores for tet-mesh point location [61] and unstructured mesh point location [47]. Laass formulated the PIP query as a ray tracing problem by transforming polygons into a 3D representation [32]. RayJoin adapts a different formulation for PIP and supports the Line Segment Intersection query [22], but it does not support the point and range queries. LibRTS does not require formulating queries on a case-by-case basis and supports updating geometries.

Scientific Workloads Simulating Monte Carlo particle transport has been explored by Salmon et al. [58]. Maul et al. proposed utilizing RT cores to accelerate the generation of synthetic X-ray attenuation images [42]. Zhao et al. suggested leveraging RT cores for particle-based simulations [72]. These works are remotely related to our research.

Generalization of RT Cores Numerous studies have been proposed on the generalization of RT cores. Ha et al. [25] introduced the Tree Traversal Accelerator (TTA), which extends Ray-Tracing Accelerators (RTA) for general tree traversal applications such as B-tree searches and radius search algorithms by modifying the architecture of RTA. Similarly, Barnes et al. [7] proposed the Hierarchical Search Unit (HSU) along with a set of new instructions for the HSU to accelerate a broad class of hierarchical search problems.

8 Conclusion

In this paper, we successfully adapted RT cores to function as a general-purpose spatial index. Despite the strict limitations of the programming model, our transformation method effectively enables RT cores to handle range queries. Additionally, the *Ray Multicast* technique enhances workload balance within the existing RT framework. Furthermore, The adoption of *Instancing* enables geometry insertion and deletion. This work enhances the accessibility of RT cores for spatial developers, broadening their utility in spatial data processing and paving the way for future research in repurposing specialized hardware for diverse computational tasks.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. The work is supported in part by the U.S. National Science Foundation under grants MRI-2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC-2310510.

References

[1] Danial Aghajarian and Sushil K. Prasad. 2017. A Spatial Join Algorithm Based on a Non-uniform Grid Technique over GPGPU. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Redondo Beach, CA, USA) *(SIGSPATIAL '17)*. Association for Computing Machinery, New York, NY, USA, Article 56, 4 pages.

[2] Varol Akman, Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1989. Geometric computing and uniform grid technique. *Computer-Aided Design* 21, 7 (1989), 410–420.

[3] Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G Aref. 2024. A Survey of Learned Indexes for the Multi-dimensional Space. *arXiv preprint arXiv:2403.06456* (2024).

[4] Samuel Audet, Cecilia Albertsson, Masana Murase, and Akihiro Asahara. 2013. Robust and efficient polygon overlay on parallel stream processors. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Orlando, Florida) *(SIGSPATIAL'13)*. Association for Computing Machinery, New York, NY, USA, 304–313.

[5] Muhammad A Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D Owens. 2019. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th symposium on principles and practice of parallel programming*. 145–157.

[6] Muhammad A Awad, Serban D Porumbescu, and John D Owens. 2022. A GPU Multiversion B-Tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 481–493.

[7] Aaron Barnes, Fangjia Shen, and Timothy G. Rogers. 2024. Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1027–1040.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

[10] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[11] Guy E Blelloch and Magdalen Dobson. 2022. Parallel Nearest Neighbors in Low Dimensions with Batch Updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 195–208.

[12] Boost. 2024. Boost C++ Libraries. http://www.boost.org/. Last accessed 2024-08-07.

[13] John Burgess. 2020. Rtx on—the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.

[14] CGAL. 2024. Computational Geometry Algorithms Library. https://www.cgal.org. Last accessed 2024-08-07.

[15] James H Clark. 1976. Hierarchical geometric models for visible surface algorithms. *Commun. ACM* 19, 10 (1976), 547–554.

[16] Ding, Jialin and Minhas, Umar Farooq and Yu, Jia and Wang, Chi and Do, Jaeyoung and Li, Yinan and Zhang, Hantian and Chandramouli, Badrish and Gehrke, Johannes and Kossmann, Donald and Lomet, David and Kraska, Tim. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[17] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[18] Esri. 2023. *ArcGIS Hub*. Retrieved Feb 21, 2023 from https://hub.arcgis.com

[19] Iordanis Evangelou, Georgios Papaioannou, Konstantinos Vardis, and Andreas A Vasilakis. 2021. Fast radius search exploiting ray-tracing frameworks. *Journal of Computer Graphics Techniques Vol* 10, 1 (2021), 25–48.

[20] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4 (1974), 1–9.

[21] Tim Foley and Jeremy Sugerman. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 15–22.

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

[23] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the 5th High-Performance Graphics Conference*. 81–88.

[24] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[25] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M. Aamodt. 2024. Generalizing Ray Tracing Accelerators for Tree Traversals on GPUs. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1041–1057.

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

[27] Justus Henneberg, Felix Schuhknecht, Rosina Kharal, and Trevor Brown. 2024. More Bang For Your Buck (et): Fast and Space-efficient Hardware-accelerated Coarse-granular Indexing on GPUs. *arXiv preprint arXiv:2406.03965* (2024).

[28] Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics*. 33–37.

[29] Puloma Katiyar, Tin Vu, Ahmed Eldawy, Sara Migliorini, and Alberto Belussi. 2020. Spiderweb: a spatial data generator on the web. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. 465–468.

[30] Timothy L Kay and James T Kajiya. 1986. Ray tracing complex scenes. *ACM SIGGRAPH computer graphics* 20, 4 (1986), 269–278.

[31] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[32] Moritz Laass. 2021. Point in Polygon Tests Using Hardware Accelerated Ray Tracing. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*. 666–667.

[33] Margaret Lawson, William Gropp, and Jay Lofstead. 2022. Exploring Spatial Indexing for Accelerated Feature Retrieval in HPC. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 605–614.

[34] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.

[35] Guanli Liu, Jianzhong Qi, Christian S Jensen, James Bailey, and Lars Kulik. 2023. Efficiently learning spatial indices. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1572–1584.

[36] Jiesong Liu, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li, Huanchen Zhang, Jidong Zhai, Hechen Zhang, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. G-Learned Index: Enabling Efficient Learned Index on GPU. *IEEE Transactions on Parallel and Distributed Systems* 35, 6 (2024), 950–967.

[37] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. 2024. How Good Are Multi-dimensional Learned Indices? An Experimental Survey. *arXiv preprint arXiv:2405.05536* (2024).

[38] 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* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 549–565.

[39] Lijuan Luo, Martin DF Wong, and Lance Leong. 2012. Parallel implementation of R-trees on the GPU. In *17th Asia and South Pacific Design Automation Conference*. IEEE, 353–358.

[40] 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. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1460–1472.

[41] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing*. 14–25.

[42] J Maul, Sarah Said, N Ruiter, and Torsten Hopp. 2021. X-ray synthesis based on triangular mesh models using GPU-accelerated ray tracing for multi-modal breast image registration. In *Simulation and Synthesis in Medical Imaging: 6th International Workshop, SASHIMI 2021, Held in Conjunction with MICCAI 2021, Strasbourg, France, September 27, 2021, Proceedings 6*. Springer, 87–96.

[43] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 683–712.

[44] Enzo Meneses, Cristóbal A Navarro, Héctor Ferrada, and Felipe A Quezada. 2024. Accelerating range minimum queries with ray tracing cores. *Future Generation Computer Systems* 157 (2024), 98–111.

[45] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[46] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[47] Nate Morrical, Ingo Wald, Will Usher, and Valerio Pascucci. 2020. Accelerating unstructured mesh point location with RT cores. *IEEE transactions on visualization and computer graphics* 28, 8 (2020), 2852–2866.

[48] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. 2022. Waffle: A Workload-Aware and Query-Sensitive Framework for Disk-Based Spatial Indexing. *Proceedings of the VLDB Endowment* 16, 4 (2022), 670–683.

[49] 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*. 289–300.

[50] NVIDIA 2018. *NVIDIA TURING GPU ARCHITECTURE*. NVIDIA. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[51] NVIDIA 2020. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. NVIDIA. https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[52] NVIDIA. 2024. cuSpatial. https://docs.rapids.ai/api/cuspatial/stable/. Last accessed 2024-08-07.

[53] NVIDIA. 2024. NVIDIA OptiX 8.0 – Programming Guide. https://raytracing-docs.nvidia.com/optix8/guide/index.html. Last accessed 2024-08-07.

[54] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically based rendering: From theory to implementation*. MIT Press.

[55] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 618–627.

[56] Jianzhong Qi, Guanli Liu, Christian S Jensen, and Lars Kulik. 2020. Effectively learning spatial indices. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2341–2354.

[57] Steven M Rubin and Turner Whitted. 1980. A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*. 110–116.

[58] Justin Salmon and Simon McIntosh-Smith. 2019. Exploiting hardware-accelerated ray tracing for Monte Carlo particle transport with OpenMC. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 19–29.

[59] Jaewoo Shin, Ahmed R Mahmood, and Walid G Aref. 2019. An investigation of grid-enabled tree indexes for spatial query processing. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 169–178.

[60] Darius Šidlauskas, Simonas Šaltenis, Christian W Christiansen, Jan M Johansen, and Donatas Šaulys. 2009. Trees or grids? Indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems*. 236–245.

[61] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. 2019. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. *High Performance Graphics (Short Papers)* 7 (2019), 13.

[62] Congying Wang, Jia Yu, and Zhuoyue Zhao. 2023. GLIN: A (G) eneric (L) earned (In) dexing Mechanism for Complex Geometries. In *Proceedings of the 11th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 1–12.

[63] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 38–52.

[64] Yangzihao Wang, Andrew 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*. 1–12.

[65] Yiqiu Wang, Rahul Yesantharao, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. ParGeo: A Library for Parallel Computational Geometry. In *30th Annual European Symposium on Algorithms (ESA 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 244)*, Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 88:1–88:19.

[66] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2022. Pargeo: A library for parallel computational geometry. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 450–452.

[67] Turner Whitted. 1980. An improved illumination model for shaded display. *Commun. ACM* 23, 6 (jun 1980), 343–349.

[68] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2019. Catfish: Adaptive RDMA-enabled r-tree for low latency and high throughput. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 164–175.

[69] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. 2022. An RDMA-enabled In-memory Computing Platform for R-tree on Clusters. *ACM Transactions on Spatial Algorithms and*

Systems (TSAS) 8, 2 (2022), 1–26.

[70] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on gpus using r-trees. In *Proceedings of the 2Nd ACM SIGSPATIAL international workshop on analytics for big geospatial data*. 23–31.

[71] Jianting Zhang and Simin You. 2013. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *International Journal of Geographical Information Science* 27, 11 (2013), 2207–2226.

[72] Shiwei Zhao, Zhengshou Lai, and Jidong Zhao. 2023. Leveraging ray tracing cores for particle-based simulations on GPUs. *Internat. J. Numer. Methods Engrg.* 124, 3 (2023), 696–713.

[73] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–11.

[74] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 76–89.

A Artifact Evaluation

A.1 Availability

The user can either download the artifact evaluation package from Zenodo https://doi.org/10.5281/zenodo.14209767 or Github https://github.com/RTSpatial/PPoPPAE. If you clone the repository on GitHub, do not forget to use "–recursive" option to download the dependencies.

A.2 Hardware Requirements

To run the experiments, you need a machine equipped with an NVIDIA RTX Series GPU, ideally an RTX 3090 (as used in the paper). The GPU should have at least 24 GB of VRAM to run all experiments successfully. Non-RTX GPUs, such as the A100 or H100, should not be used to reproduce the results, as they don't have RT cores. Additionally, the machine should have high-performance CPUs for evaluating CPU-based baselines and sufficient RAM to load the datasets (64 GB should be adequate).

Note that we do not explicitly specify which GPU to use in the code, so the first available GPU will be used in the experiments. If your RTX GPU is not the first one, you should make it accessible to the benchmark program by setting "export CUDA_VISIBLE_DEVICES=x", where x is the index of your RTX GPU.

A.3 Software Requirements

The evaluation scripts are developed for Linux only. In addition, the user should make sure the following programs are available.

- bash
- wget
- unzip
- tar
- md5sum
- gcc (>=7.5)
- Conda (>=22.11)
- CMake (>=3.27)
- CUDA (>=12)
- NVIDIA Driver (>=535)

A.4 Evaluation

If the above software requirements are met, simply navigate to the root of the codebase and execute "./runme.sh". This script will build and install dependencies, download datasets, build and run baselines and finally, draw the figures in the paper. Once the script completes, Figures 7–13 will be generated and can be found in "figures" folder within the root directory of the codebase.

Since this paper evaluates both CPU and GPU-based spatial libraries and your machine with a GPU may not have a powerful CPU, so you want to run the CPU-based baselines on a different machine. You can control whether to evaluate CPU/GPU-based baselines by setting variables "AE_RUN_CPU" and "AE_RUN_GPU" in "common.sh". Additionally, you may also set "AE_BUILD_GPU" to 'OFF' if your machine does not have installed CUDA.