RayJoin: Fast and Precise Spatial Join

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@cse.ohio-state.edu

ABSTRACT

Real-time spatial data analysis is a fundamental requirement for many critical applications in this digital era. However, such a requirement in practice is often hindered by the low performance of spatial join queries on conventional parallel systems. Specifically, all existing spatial join methods, including both classical plane sweeping algorithms and various grid- or tree-based index-assisted algorithms, have unacceptably long execution times and thus fail to deliver real-time performance. In this paper, we present RayJoin, a new and effective approach that utilizes the ray tracing hardware in modern GPUs (e.g., NVIDIA RT Cores) as accelerators to overcome the bottlenecks in spatial join processing and push the performance to an unprecedented level. Specifically, RayJoin consists of a high-performance and high-precision spatial join framework that accelerates two vital spatial join queries: line segment intersection (LSI) and point-in-polygon test (PIP). Besides these ray tracing-backed algorithms, RayJoin also contains new solutions to address two challenging technical issues: (1) how to meet the high precision requirement of spatial data analysis with the insufficient precision support by the underlying hardware, and (2) how to reduce the high buildup cost of the hardware-accelerated index, namely Bounding Volume Hierarchy (BVH), while maintaining optimal query performance. Our evaluation results show that RayJoin achieves speedups from 3.0x to 28.3x over any existing highly optimized methods in high precision. To the best of our knowledge, RayJoin stands as the sole solution capable of meeting the real-time requirements of diverse workloads, taking under 460ms to join millions of polygons.

CCS CONCEPTS

• **Information systems** → *Join algorithms*; **Spatial-temporal systems**; • **Computing methodologies** → **Ray tracing**.

KEYWORDS

Ray Tracing, Hardware Acceleration, Spatial Join, Spatial- and Geo-Databases

ACM Reference Format:
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 '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656610

ICS '24, June 04–07, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656610

1 INTRODUCTION

Spatial join queries are pivotal in Geographic Information Science (GIS) and geo-databases. A spatial join query aims to identify pairs of geometries from multiple datasets that satisfy a given predicate, such as "intersect" or "contain." [7, 32, 45]. For example, it can determine the locations of bridges by querying intersections between a road network and a hydrographic map. However, the increasing volume of GIS data collected from various sources, including satellites and mobile devices, poses challenges for timely data analytics [4, 10, 37, 51]. This requirement is particularly acute in domains like interactive urban planning, emergency management, and risk assessment [58, 72, 79].

Various techniques have been explored to address the challenges of large-scale data and low-latency processing. The plane-sweep algorithm is favored for its low complexity [12, 47, 66] but suffers from limited parallel processing capabilities [5, 6, 41]. Rasterization-based methods offer parallelism but require intensive preprocessing [16, 65, 79]. Learned spatial indexes, though efficient in query performance and storage, struggle with high training costs and accuracy concerns [18, 29, 52, 53, 59, 82]. Consequently, GIS researchers and practitioners often resort to conventional indexing techniques, such as grids and trees, which offer numerous advantages [4, 6, 10, 17, 62–64, 71, 73–77, 80].

However, the conventional spatial indexes mentioned above have performance limitations from achieving real-time processing [61]. While tree-based methods offer a unique advantage in logarithmic time complexity, they suffer from inherent shortcomings, including branches, irregular memory accesses, and memory space overhead for storing pointers. These drawbacks notably undermine the efficiency of spatial join operations, particularly on GPUs. In contrast, grids convert spatial queries into hardware-friendly scan operations within a limited search space, thereby maximizing memory access efficiency. However, skewed data distributions may result in imbalanced workloads that hinder the effectiveness of parallel processing [43, 44, 69].

Over the years, algorithmic improvements in conventional indexing methods seriously mismatch with the execution models of general-purpose computer architecture, including CPUs and GPUs, leading to persistent performance bottlenecks and associated trade-offs in designing databases and analytical systems [13, 31, 78, 81]. In fact, our evaluation shows that none of the existing spatial join methods satisfy the real-time processing requirement (§5.7), defined as the completion time within a second. Recently, the GPU advancement of dedicated ray-tracing units, such as NVIDIA's RT Cores, opens a new opportunity to accelerate spatial join queries through hardware-accelerated Bounding Volume Hierarchy (BVH) tree traversal. Moreover, the processing capability of RT Cores is doubled per generation, making RT-based methods automatically benefit from the advancing of this new hardware [48, 49]. Utilizing

the technology advancement, we have designed and implemented a new and effective acceleration engine for spatial joins with RT Cores, which is called RayJoin, to address the limitations of all existing approaches.

Leveraging RT Cores for spatial join queries presents three non-trivial challenges for us to address. First, to harness the dedicated hardware effectively, it is crucial to devise an efficient method that formulates spatial join queries as ray tracing problems. In RayJoin, we showcase two critical spatial join queries, line segment intersection (LSI) and point-in-polygon (PIP) test. We transform these two queries into RT-friendly algorithms using the *AnyHit* and *ClosestHit* shaders supported by RT Cores. Specifically, we have leveraged these queries to develop a polygon overlay application. Second, the precision in RT Cores is limited to single-precision floating-point (FP32) for high-performance rendering. However, this limitation poses a significant challenge in meeting the requirements of GIS applications. To address this, we have proposed a conservative representation that stores polygons in low precision while still delivering exact query results. In computer arithmetic, a conservative method ensures the precision of computation results, even at a cost of performance. RayJoin achieves both high precision and high performance. Finally, the BVH construction cost, in both time and space, is another bottleneck to be resolved to achieve high performance. We have noticed that the construction time is often a magnitude higher than the query time. In response, we have introduced a grouping technique that greatly reduces the construction cost by reducing the number of primitives used in the BVH. This effectively reduces construction costs while keeping high query performance.

Our experiments show that RayJoin has achieved 3.0x to 28.3x speedups over any existing polygon overlay solutions. It only takes about 460ms on the full-fledged polygon overlay analysis to join 1.6M polygons with 303K polygons, meeting real-time spatial join query requirements. In this paper, we have made a strong case for using RT Cores to accelerate various spatial queries in geodatabases, and our contributions are as follows:

- We have developed an effective approach that utilizes RT Cores to accelerate LSI and PIP queries, substantially outperforming conventional spatial join techniques on both CPUs and GPUs. To the best of our knowledge, RayJoin is the fastest spatial join solution (§3.1).
- Within the existing GPUs that only support FP32 ray-tracing, we have developed a method to effectively achieve double-precision floating-point (FP64) while maintaining performance levels similar to the underlying hardware (§3.2).
- We have proposed an algorithm that greatly reduces the BVH construction time and memory cost while keeping high query performance (§3.3).
- We have comprehensively compared and analyzed RayJoin with many existing spatial join solutions by a consistent benchmark with synthetic and real-world datasets (§5), making a strong case for RT Cores to attain fast and precise spatial join.
- RayJoin is an open-source software project for widespread public usage.[1]

[1]Source code: https://github.com/pwrliang/RayJoin

2 BACKGROUND

2.1 Spatial Join Queries and Spatial Indexes

Spatial join finds pairs of spatial objects (r, s) from two datasets R and S that satisfy a spatial predicate θ, where $r \in R, s \in S$. Formally,

$$R \bowtie_\theta S := \{(r, s) \mid r \in R, s \in S, \theta(r, s) = true\}$$

Spatial objects come in several forms, including points, line segments, polylines, and polygons. Predicates, such as intersect, contain, overlap, etc., can be combined to perform complex spatial queries. Figure 1 provides an example of joining two sets of polygons R and S, which is also called polygon overlay analysis[2]. The polygon overlay is achieved by combining the results of LSI and PIP queries. The LSI query generates points at the intersections where the boundaries of datasets R and S meet. The PIP query determines the polygon $r \in R$ where a query point $s \in S$ lies and vice versa. The output from the above two steps is combined to obtain the resulting polygon (on the right of Figure 1).

Performing spatial join queries on a large volume of complex geometries is challenging due to the high computational cost [3]. Exhaustive search compares all pairs of geometries, resulting in $O(|R| \cdot |S|)$ time complexity, which is impractical for large datasets [40]. Thus, spatial indexes are necessary to improve query performance by reducing search space.

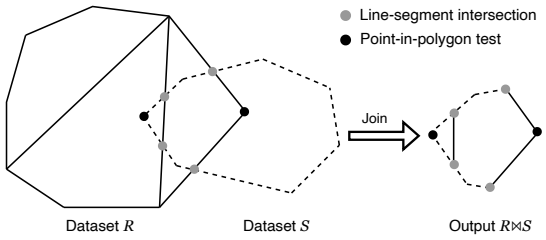


Figure 1: Example: polygon overlay analysis through spatial join. Dataset R contains the union of multiple simple polygons with common boundaries. Dataset S has a simple polygon. The output is the overlapped regions from two datasets.

This section introduces two typical tree-based and grid-based spatial indexes, namely BVH and uniform grid. To clarify, in graphics, BVH is a general term to describe a hierarchical data structure that partitions geometries with bounding volume. For example, the R-tree can be categorized into BVH because it uses Minimum Bounding Rectangle (MBR) as the bounding volume [19].

BVH. The BVHs organize geometries in tree-like data structures. The internal node in the BVH is associated with an axis-aligned bounding box (AABB)[3], which tightly encloses the union of the children's AABBs. Leaf nodes hold the primitives[4], which can be points, lines, triangles, or any shapes. AABB approximates the primitive boundary, making the intersection test more efficient. Building a BVH can be efficiently converted into a GPU-friendly

[2]To understand dataset R, considering the provinces/states (the three simple polygons) share the common boundaries forms into a country (the whole polygon).
[3]AABB is a synonym of MBR in this paper.
[4]We use the word "primitive" interchangeably with "geometry." In the context of BVH, we prefer the word "primitive" because they are conventionally used together in computer graphics.

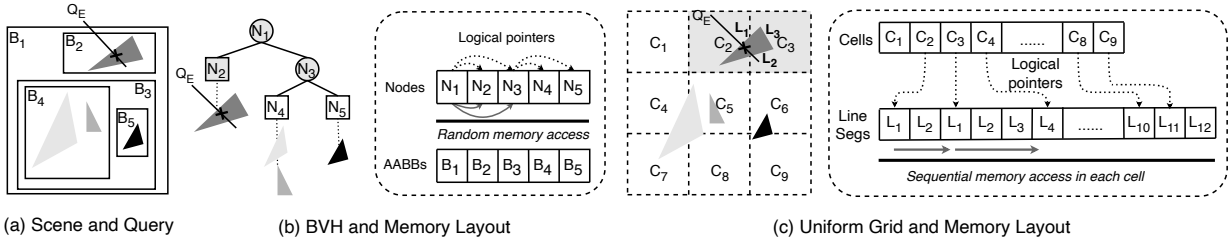


Figure 2: (a): given a group of geometric objects and a query Q_E, finding which geometry intersects with the query. (b) and (c): demonstrate the acceleration of the query with two spatial indexes and their memory layouts and access patterns.

sorting problem. For example, Linear Bounding Volume Hierarchy (LBVH) organizes geometries in the order of their Morton Code, balancing the construction time and index quality [26].

Figure 2 (a) depicts a query Q_E being performed on a scene composed of four triangles. Figure 2 (b) illustrates how to accelerate the query with the BVH. The query aims to identify all the triangles intersecting the query. This process begins by performing an intersection test between Q_E and the AABB B_1 of the root node. If an intersection is detected, further tests are conducted between Q_E and the nodes N_2 and N_3. The search proceeds until we narrow it down to the leaf node N_2. Then, we only need to execute the intersection test against the triangle bounded by B_2. The right sub-tree rooted at N_3 can then be pruned as it does not intersect with the query.

Tree-based indexes offer logarithmic time complexity of searching, which is a key advantage. However, they can be memory inefficient. As shown in Figure 2 (b), tree nodes are stored in the "*Nodes*" array. Logical pointers connect parent nodes to their child nodes. Each query starts by visiting the root node and following the logical pointers to its descendants, resulting in a Depth-First Search (DFS). The pointer-chasing during DFS leads to irregular memory access patterns and branches, preventing coalesced memory access optimization and resulting in cache line inefficiencies. As a result, numerous memory transactions are generated during the traversal. Determining the traversal path also results in branch divergence on the GPU, causing very few threads to be active in each warp.

Uniform Grid. The uniform grid is a spatial index that partitions the space into equal-sized cells. The width of the grid is called the resolution. Figure 2 (c) represents a 3x3 uniform grid built for the scene. The triangles are split into line segments for efficient retrieval. The grid data structure consists of "Cells" and "Line Segs" array. The "Cells" array maintains the beginning offsets that point to the positions in the "Line Segs" array, where the line segments within the cell are organized in a contiguous memory region.

Accelerating the query Q_E can be achieved through the uniform grid, as demonstrated in Figure 2 (c). The cells that overlap with the query are collected (shaded with gray). Then, we scan line segments in the cells to determine whether they intersect with the query. With the grid, the line segments spatially distant from the query are pruned. The finer the grid, the more line segments are pruned, leading to improved query efficiency, but the construction cost is also increased due to duplicated line segments. Thus, the resolution of the grid must be carefully tuned.

The uniform grid offers several advantages, including GPU-friendly scan operations and a low construction cost. If geometries are distributed uniformly, the grid-based indexes should be ideal candidates for GPUs. Unfortunately, this is not true in real-world applications. For example, geometries representing buildings are dense in metropolitan cities but usually sparse in deserts, which forms an uneven grid, limiting the parallelism of GPUs and leading to load imbalance and low occupancy.

2.2 Ray-Tracing and Its Programming Model

Ray tracing is a technique for rendering photorealistic images. The fundamental principle of ray tracing is identifying the primitives hit by rays, which is time-consuming, typically requiring an Accelerating Structure (AS) to reduce search space. Currently, BVH is the most widely utilized AS in ray tracing frameworks, as discussed in §2.1. Despite their efficiency, tree-based indexes are not optimal for use on GPUs. As a result, researchers have proposed various hardware-based solutions to accelerate BVH traversal. For example, NVIDIA RTX series GPUs provide dedicated ray tracing units to accelerate both BVH traversal and ray-primitive intersection test [8]. Several ray-tracing interfaces are available for the ray-tracing hardware, including NVIDIA OptiX, Vulkan, and DirectX Raytracing. This paper focuses on OptiX as it is interoperable with CUDA.

OptiX supports the "Multiple Instruction, Multiple Data" (MIMD) subset of CUDA [50]. Each thread casts a ray, and the execution of the ray could be rescheduled at runtime to a different lane, warp, and even Streaming Multiprocessor (SM) for execution efficiency. Thus, shared memory and warp/block synchronization intrinsics are not available. A ray is allowed to carry per-ray data via registers to communicate to shaders. Shaders are user-defined CUDA programs to cast rays and handle ray-primitive intersection events.

Before casting rays, we have to build the BVH from an array of primitives. The BVH structure is hardware-specific and handled by the video driver. Thus, users have no control over the BVH construction process but only supply primitives. Constructing a BVH is expensive. The construction time is linearly correlated to the number of primitives [83]. Upon completion of the BVH construction, a traversal handle is returned, which is used to access the BVH. The BVH traversal occurs on the RT Cores, which share the same device memory with the SMs.

Figure 3 illustrates the execution flow of OptiX (shapes shaded in grey are non-programmable procedures). OptiX does not allow explicit BVH traversal and instead begins by generating a set of rays. The entry point of a ray tracing program is the RayGen (RG) shader, where the traversal handle, origins, and directions of the rays are provided to generate rays and identify hits. The BVH is

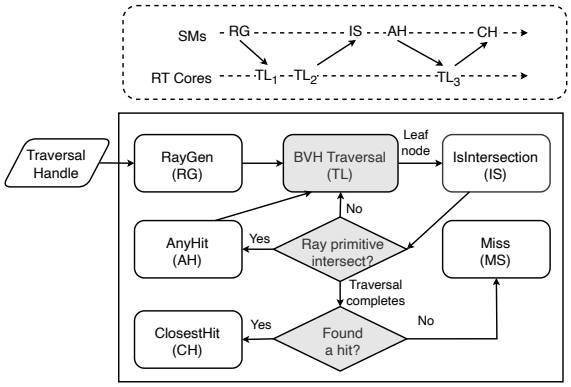


Figure 3: Top: Switching between the RT Cores and SMs; Bottom: Execution flow of OptiX.

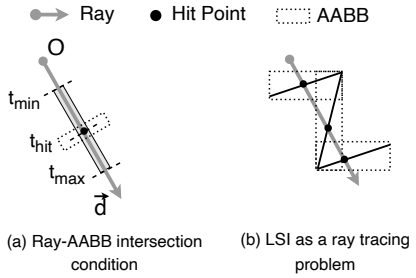


Figure 4: Formulating LSI as a ray tracing problem. By simulating the line segment with a ray, intersections are identified by ray-AABB intersection tests

autonomously traversed, guided by the rays. The traversal stage is non-programmatic and transparent to the user. Figure 3 top shows the execution flow of the query on Figure 2 (b), where TL_i corresponds to the traversal of BVH node N_i. If leaf nodes are reached during the traversal, the IsIntersection (IS) shader is invoked to determine if a ray intersects with a primitive. When the IS shader reports an intersection, the AnyHit (AH) shader is called, allowing the user to execute rendering tasks. At the end of the traversal, the ClosestHit (CH) shader is called, which returns the closest primitive hit by the ray. The Miss (MS) shader is invoked if the ray does not hit any primitive. In OptiX, the coordinates of primitives and ray parameters are defined in the FP32 for performance reasons. This limitation makes implementing high-precision applications challenging. Thus, this issue must be addressed to accommodate broad applications.

3 SPATIAL JOIN OPERATIONS ON RT CORES

3.1 From Spatial Join to Ray-tracing

Ray Definition. A ray $\mathbb{R}$ is a semi-infinite line characterized by its origin O, a direction vector $\vec{d}$, and a parameter t. The parameter t allows the ray to extend indefinitely in the direction of $\vec{d}$ starting from O. Formally, it is defined in Equation (1):

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

When casting a ray, users are asked to provide a range parameter $[t_{min}, t_{max}]$ to filter the intersections on a segment of the ray (Figure 4 (a)). If the ray hits a primitive (such as an AABB), a hit event is reported only if $t_{min} \leq t_{hit} \leq t_{max}$, where $O + t_{hit} \cdot \vec{d}$ is the intersection point.

An essential step in ray tracing is finding ray-primitive intersections. Interestingly, this process bears the similarity of spatial join problems. For example, a complex polygon overlay query can be decomposed into two sub-problems, LSI and PIP [37], which can then be formulated as two RT-friendly problems.

Casting Rays for LSI. Given a base map R and a query map S, LSI finds all the intersections from $R \bowtie_{intersect} S$. This can be achieved by ray tracing. We first construct a BVH from an array of AABBs, where each AABB encloses a line segment from R. A line segment from S can be represented by a ray. Subsequently, we cast rays generated from S and collect all intersections (Figure 4 (b)). The following ray parameters O, $\vec{d}$, t_{min}, and t_{max} defines a line segment $s \in S$, where p_1 and p_2 are the two endpoints of s.

$$O = p_1, \vec{d} = p_2 - p_1, t_{min} = 0, t_{max} = 1 \tag{2}$$

Putting the parameter $t = t_{min}$ into equation (1), $\mathbb{R}(t_{min}) = p_1 + t_{min} \cdot (p_2 - p_1) = p_1$, which is an endpoint of s. Carrying $t = t_{max}$, we have $\mathbb{R}(t_{max}) = p_1 + t_{max} \cdot (p_2 - p_1) = p_2$, which is another endpoint. Thus, we can see that any intersection point p_x on s can be represented by $p_x = R(t) = O + t \cdot (p_2 - p_1)$, where $t_{min} \leq t \leq t_{max}$, and the ray-AABB intersection test identifies these hit points.

Algorithm 1 illustrates the above idea, which is orchestrated by OptiX's callbacks. *RayGen* shader is the entry of the algorithm that computes O and $\vec{d}$ using the endpoints. At Line 8, $optixTrace$ shoots a ray carrying the line segment ID (per-ray data) from S to find intersections. If the ray hits an AABB, a user-defined function *IsIntersection* is triggered to verify the intersection and calculate the coordinate of intersection point p_x and the ray parameter t_{hit}. This step is necessary because a ray hitting an AABB does not guarantee the ray intersects the line segment enclosed by the AABB. If p_x exists, we call $optixReportIntersection$, which reports the intersections and extra information with attributes to OptiX.

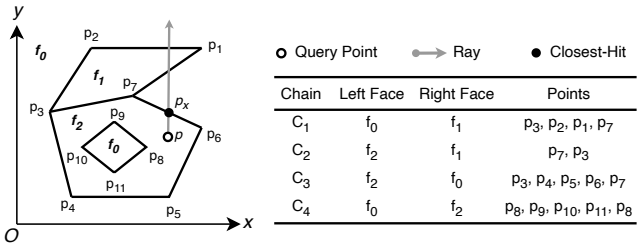
Upon the reported intersections, if t_{hit} falls within the interval $[t_{min}, t_{max}]$, the *AnyHit* shader is automatically triggered. Remember that each line segment corresponds to an AABB, and we have verified the ray indeed intersects the line segment enclosed by the AABB. We can now safely put the intersection into result set Q. At line 27, we invoke the $optixIgnoreIntersection$ function to ask OptiX to continue searching.

Casting Rays for PIP. To begin with, we want to clarify that the classical PIP test determines whether a point lies inside or outside a polygon. This can be addressed with the crossing number algorithm [60]. However, we are solving a more complex PIP problem here. Given a query point and a group of polygons, we want to know which polygon the point falls in. Specifically, this problem is also called "Point Location" in some literature [56]. This operation is the building block of complex spatial queries like polygon overlay [37].

Figure 5 presents an example polygon alongside its in-memory representation, which will be used to demonstrate how the PIP

Algorithm 1: Formulate LSI as a ray-tracing problem

```
Input  : R - a set of line segments from base map
Input  : S - a set of line segments from query map
Input  : h - the BVH traversal handle built from R
Output : Q - results of intersections
1  procedure RayGen // Entrypoint, invoked when the program starts
2      t_min, t_max = {0, 1}
3      for each s in S do                    // For each line segment from S
4          p1, p2 = GetEndpoints(s)          // Get two endpoints from line segment s
5          O = p1                                                  // Ray origin
6          d = p2 − p1                                          // Ray direction
7          // Cast a ray carrying line segment ID of s
8          optixTrace(h, O, d, t_min, t_max, payloads(id_s))
9      end for
10 end procedure
11
12 procedure IsIntersection   // Invoked when ray potentially hits an AABB
13     O = optixGetRayOrigin()                                      // Ray origin
14     d = optixGetRayDirection()                                // Ray direction
15     id_r = optixGetPrimitiveIndex()      // Line segment id from R
16     id_s = optixGetPayload_0()           // Line segment id from S
17     p = R[id_r] ⋈_intersects S[id_s]   // Calculate intersection point
18     if p ≠ φ then
19         t_hit = (p_x.x − O.x) / d.x        // Calculate t_hit with x-coordinate
20         optixReportIntersection(t_hit, attributes(id_r, id_s, p))
21     end if
22 end procedure
23
24 procedure AnyHit   // Invoked when intersections being reported
25     id_r, id_s, p_x = optixGetAttribute_{0...2}()                // From optixReportIntersection
26     Q = Q ∪ (id_r, id_s, p_x)                // Collect the intersection
27     optixIgnoreIntersection() // Ask OptiX to continue searching
28 end procedure
```



Chain	Left Face	Right Face	Points
C_1	f_0	f_1	p_3, p_2, p_1, p_7
C_2	f_2	f_1	p_7, p_3
C_3	f_2	f_0	p_3, p_4, p_5, p_6, p_7
C_4	f_0	f_2	$p_8, p_9, p_{10}, p_{11}, p_8$

(a) PIP query with ray-tracing (b) Chain representation

Figure 5: Formulating PIP as a ray tracing problem and the chain representation of the left polygon. Notably, C_4 is a hole, so it only involves two adjacent faces, resulting in a "closed" point sequence (the first point also appears as the last point).

works with ray tracing. The point sequence $(p_1, p_2, \ldots, p_7, p_1)$ consists of the polygon's outline. The line segment (p_7, p_3) splits the polygon into two faces[5] f_1 and f_2. The diamond-shaped polygon $(p_8, p_9, p_{10}, p_{11}, p_8)$ on the bottom is a hole with face f_0. A chain is a polyline that belongs to the same adjacent faces. For instance, the polyline (p_3, p_2, p_1, p_7) should be organized as a chain because the line segment (p_3, p_2), (p_2, p_1), and (p_1, p_7) share the same left face f_0, and right face f_1. The order of points determines the adjacent faces of a polyline. For example, in chain C_2, the left and right faces are f_2 and f_1, respectively. If the point sequence is (p_3, p_7), the left face is f_1 and the right face is f_2. The chain representation saves memory by storing common boundaries only once and facilitates the PIP algorithm by providing neighborhood information.

The idea of PIP is to shoot a ray from the query point vertically upwards and to identify the closest line segment hit by the ray [14, 37]. This method works both on convex and non-convex polygons, even polygons with holes. The simulation of ray casting is traditionally achieved with grids or trees, which have many shortcomings, as mentioned in §2. Since the PIP problem is essentially a ray tracing problem, solving it with RT Cores is natural. The algorithm works as follows. Figure 5 (a) shows a query p, locating which polygon the query falls in. We shoot a vertical ray from p upwards to identify the closest line segment hit by the ray, which is (p_6, p_7) in chain C_3. Then, we use the chain table (Figure 5 (b)) to determine the face where the query point is located. Since the x-coordinate of point p_6 is greater than that of p_7, we take the left face f_2 of chain C_3, as the polygon ID.

Algorithm 2 describes the implementation of PIP with OptiX. Again, the *RayGen* shader is the entry point of the algorithm, starting from casting vertical rays from every point $p \in S$. In *IsIntersection* shader, we calculate the distance t_{hit} between the query point p and the hit point p_x, which is calculated by solving the common solution of line equations (Line 15). Since the ray may hit multiple line segments, we report each t_{hit} to OptiX by invoking *optixReportIntersection*, asking OptiX to find out the closest hit. When a ray hits the closest line segment, the *ClosestHit* shader will be called, where we determine which polygon the query falls in. optixGetPayload tells the ID of the query point id_p, and optixGetPrimitiveIndex returns the closest line segment ID id_l. Finally, we retrieve the line segment l and query point p by the IDs. To tell which polygon p falls in, we check the ray casting from the left or right face of l.

3.2 High Arithmetic Precision Beyond the Hardware Support

Ray-tracing frameworks like OptiX typically use FP32 for optimal performance [28]. In real-time rendering, such as in video games, the difference in visual quality between lower and higher precision is generally imperceptible to the human eye. However, precision levels are critical in areas like spatial databases. For example, FP32 coordinates offer meter-level distance precision, while FP64 coordinates achieve nanometer-level precision [25], crucial for scientific simulations and engineering analysis. Without improving the precision, the applicability of ray-tracing technology may be severely limited. Precision challenges in graphics, such as self-intersection, and robustness of ray-primitive intersection test, have been well-studied [22, 39]. However, the issue we address here stems from floating-point downcasting and differs from these well-known problems. In this section, we present a simple yet effective method to achieve exact query results for FP64 or higher precision data sources within the constraints of limited hardware support.

To tackle the precision challenges, it is crucial to understand the source of imprecision. Figure 6 represents floating-point numbers

[5]The term "face" defines a planar region enclosed by the boundaries. The hole and region that is not enclosed are defined as the exterior face f_0.

Algorithm 2: Formulate PIP as a Ray-Tracing Problem

Input : R - a set of line segments from base map
Input : S - a set of points from the query map
Input : h - the BVH traversal handle built from R
Output : Q - Polygon IDs; $Q[i]$ contains the polygon ID of query $S[i]$

```
procedure RayGen      // Entrypoint, invoked when program starts
    t_min, t_max = {0, ∞}
    for each p in S do
        O = p                                   // Query point as origin
        d⃗.xyz = {0, 1, 0}                       // Towards upward of y-axis
        optixTrace(h, O, d⃗, t_min, t_max, payloads(id_p))   // Shoot a ray, which carries the point id from S
    end for
end procedure

procedure IsIntersection   // Invoked when ray potentially hits an AABB
    id_l = optixGetPrimitiveIndex()     // Line segment id from R
    id_p = optixGetPayload_0()          // Point id from S
    p = S[id_p]
    l = make_vertical_line(p)           // Generate a vertical line segment crossing p
    p_x = R[id_l] ⋈_intersects l        // Intersection point of the line segment from R and l
    if p_x ≠ φ then
        t_hit = p_x.y − p.y             // Distance between the ray origin and the hit point
        optixReportIntersection(t_hit)  // Report intersection to OptiX
    end if
end procedure

procedure ClosestHit // Invoked when a ray hit the closest line segment
    id_l = optixGetPrimitiveIndex()     // Line segment id from R
    id_p = optixGetPayload_0()          // Query point id from S
    l = R[id_l]
    p_1, p_2 = GetEndpoints(l)
    if p_1.x < p_2.x then               // Check the line segment direction
        Q[id_p] = GetRightFace(l)
    else
        Q[id_p] = GetLeftFace(l)
    end if
end procedure
```

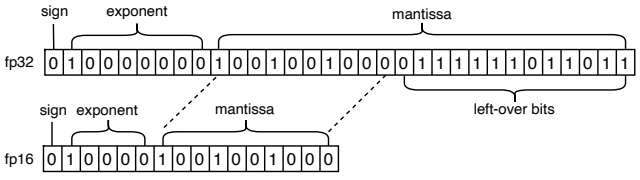


Figure 6: Downcasting a 32-bit floating-point to 16-bit.

in binary format[6], consisting of three components: sign, exponent, and mantissa [1]. When downcasting, the sign and exponent bits are preserved, ensuring no loss of information. However, the mantissa undergoes truncation. Based on the default rounding rules, known as "roundTiesToEven" [1], the mantissa bits of the low-precision type are left as-is if the discarded bits are below $10\ldots0$, while they are incremented by one if the discarded bits are above $10\ldots0$. In the case of the discarded bits being exactly $10\cdots0$, one of the rounding rules is selected that makes the least significant bit of the mantissa bits zero. Consequently, any downcasted floating-point number can have a value that is less than, greater than, or equal to the value represented by the higher precision type.

[6]For ease of representation, we demonstrate the process of downcasting from 32 bits to 16 bits instead of 64 bits to 32 bits.

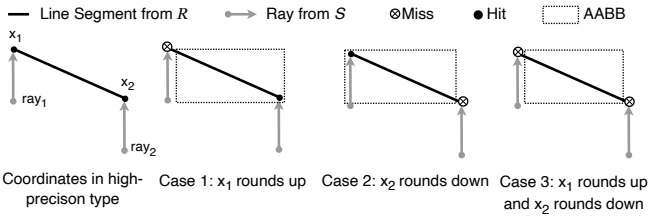


Figure 7: Low-precision AABB failed to enclose the line segment in high-precision, leading to geometric inconsistencies.

The process of downcasting high-precision data sources to low-precision primitives and rays introduces geometric inconsistencies due to the loss of mantissa bits. Figure 7 shows cases that a line segment from dataset R intersects two rays casting from dataset S, resulting in two intersections at x-coordinate x_1 and x_2 where $0 < x_1 < x_2$. For the sake of simplicity, let's assume that the precision issue only affects the x-axis when constructing the AABB. In Case 1, x_1 is rounded up, causing the left boundary of the AABB to fail to enclose the left endpoint of the line segment, leading to a false negative outcome. In Case 2, rounding down of x_2 causes ray_2 to miss the AABB, as the right endpoint falls outside the box. In Case 3, neither ray_1 nor ray_2 intersects the bounding box, resulting in the failure to detect any intersections. These examples highlight the hidden issues that can arise when employing ray-tracing techniques in GIS applications.

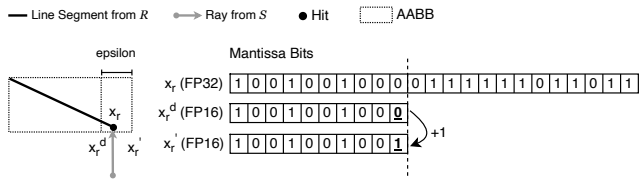


Figure 8: Conservative Representation: adding 1 to mantissa bits to compute a conservative right boundary

An intuitive solution is to create slightly larger AABBs, which can accommodate the line segment after downcasting. However, determining the appropriate size of the enlarged AABB is not trivial. Simply enlarging the AABBs by a small epsilon may still lead to intersections missing, while excessively enlarging them can result in many false positive hits, hurting performance. Formally, let x_l denote the x-coordinate of the line segment's left endpoint in high precision. We need to find the maximum value expressible in FP32 as the downcasted left AABB boundary $x_l^{'}$, satisfying $x_l^{'} \leq x_l$. Similarly, x_r represents the right endpoint in high precision. We aim to find the minimum value $x_r^{'}$ as the downcasted right boundary, ensuring $x_r^{'} \geq x_r$. The same consideration applies to the y-axis.

The solution is to create the low-precision AABB that is just barely large enough to fully enclose the high-precision line segment by tweaking mantissa bits. In the scenario depicted in Figure 8, x_r

is the x-coordinate of the line segment's right endpoint in high precision. Upon downcasting, this value is rounded down, resulting in x_r^d, which is less than x_r. To determine the smallest possible value $x_r^{'}$ as the new right boundary, we only need to add 1 to the mantissa bits after downcasting. This ensures that $x_r^{'}$ just exceeds x_r. Additionally, to compensate for errors caused by downcasting of the ray origin, we add 1 again to the mantissa bits. To sum up, we add two to the mantissa bits of the right boundary and subtract two from the mantissa bits of the left boundary. Adjusting the mantissa bits may not always be necessary, such as if the x_r is rounding up. However, we perform this operation unconditionally to avoid checking all the rounding cases, minimize branches, and enhance execution efficiency on GPUs. Hence, we refer to this approach as Conservative Representation (CR).

3.3 Taming High BVH Construction Overhead

The number of AABBs used to build the BVH equals the line segments in the dataset because each line segment is bounded by an individual AABB (§3.1). This bounding strategy introduces a long construction time and huge memory consumption[7] because the buildup cost is linearly correlated to the number of primitives [83]. One may impulsively use the Ramer–Douglas–Peucker algorithm to simplify polylines and then create an AABB for each simplified polyline [54]. Yet, this algorithm's recursive and sequential nature renders it inefficient for GPU execution. Therefore, we introduce a GPU-optimized method, termed Adaptive Grouping (AG), to group spatially close line segments into the same AABB.

We first introduce the grouping strategy. For two AABBs B_i and B_j, the union operator $\cup$ is defined as $B = B_i \cup B_j$, where B is their MBR, and $|B|$ represents its area. Considering n line segments $L_1, L_2, \ldots, L_n$ each enclosed by AABBs $B_1, B_2, \ldots, B_n$, we only merge adjacent AABBs in the sequence, such as $B_i, B_{i+1}, \ldots, B_j$ (where $1 \le i < j \le n$). This facilitates representing a line segment subset with a simple range $R[i, j]$ instead of explicitly storing each segment's ID. With the range R, we linearly scan the line segments bounded by B to find which line segment is hit by the ray. Merging B_i and B_j occurs only if the area expansion ratio $\frac{|B_i \cup B_j|}{max(|B_i|,|B_j|)} \le s$, where s is the merging threshold. We will discuss how the parameter s impacts performance and how to find an optimal s in §5.5. This approach is inspired by R-Tree's heuristic insertion algorithm [17], which effectively curtails the occurrence of false positives by limiting AABB's dead space.

Figure 9 demonstrates AG's execution. Initially, each line segment (L_0 to L_7) is in a unique group (G_0 to G_7), with a merging threshold $s = 1.5$. In the first iteration (Figure 9 (b)), L_0 and L_1 form G_0 as their area expansion ratio (1.4) is below s. However, L_2 and L_3 remain separate since $\frac{|B_1 \cup B_2|}{max(|B_1|,|B_2|)} = 2 > s$. The second iteration includes L_2 in G_0 and L_6 in G_2, but excludes L_7 from G_2 due to a ratio of 1.8, above the threshold. The final iteration merges L_3 into G_0, resulting in three groups: G_0 with B_0 and range $R_0[0, 3]$, G_1 with B_1 and range $R_1[4, 6]$, and G_2 with B_2 and range $R_2[7, 7]$.

AG can be efficiently implemented on the GPU. We divide L segments into $\lceil \frac{L}{b} \rceil$ partitions, where b is the thread block size.

[7] One may argue that building the BVH is a one-off cost. A fast buildup allows users to work on exploratory and interactively workloads, e.g., try out different datasets and run some ad-hoc queries quickly.

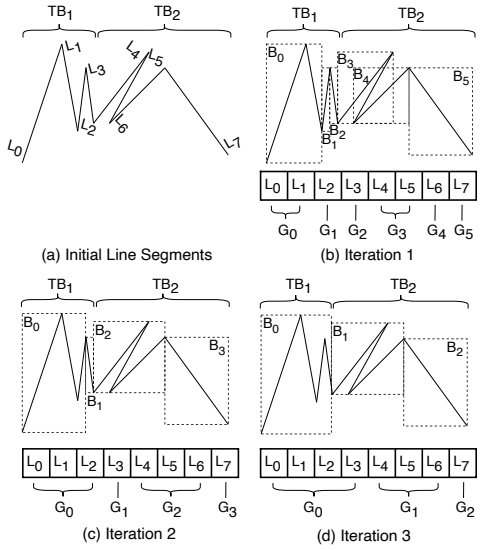


Figure 9: Execution process of Adaptive Grouping. Initially, a thread block (TB) processes a fixed number of line segments. Spatially close line segments are iteratively put together.

Each thread block processes partition i containing line segments $L_{i*b}, L_{i*b+1}, \ldots, L_{(i+1)*b}$. In the first iteration, thread j in thread block i attempts to merge two adjacent AABBs B_{i*b+j} and $B_{i*b+j+1}$, which encloses line segments L_{i*b+j} and $L_{i*b+j+1}$ respectively. The thread either produces a merged AABB $B_{i*b+j} \cup B_{i*b+j+1}$ and a range $R[i*b+j, i*b+j+1]$ if the area expanding ratio is within the threshold s. Otherwise, the AABBs remain unaltered. Subsequent iterations merge two adjacent AABBs from the previous iteration, continuing until no further merges are possible.

4 RAYJOIN DEPLOYMENT IN PRACTICE

4.1 Implementation Details

Data Storage. The dataset is stored in a planar graph format on host memory and is subsequently decomposed into line segments on device memory for further processing. Each line segment references endpoint IDs instead of storing the coordinates to conserve memory and improve the locality. We pre-compute line equation coefficients from the endpoints. This pre-calculation enables the LSI and PIP queries to access coefficients directly, avoiding redundant calculations. The line segments are then fed into the AG to create AABBs and ranges, followed by applying CR to determine the new boundaries of the AABBs, ensuring precise results.

Coordinates Representation. We employ the fixed-point notation to resolve the accumulated errors from the floating-point arithmetic operations. Given that GPUs do not possess built-in support for fixed-point arithmetic, we convert the floating-point numbers to integers through scaling. This allows all subsequent operations to be performed exclusively on integers, effectively circumventing the issue of precision loss. Scaling to integers also benefits performance because NVIDIA RTX GPUs tend to have very limited FP64 units. Furthermore, we store the intersection points with rational numbers produced by LSI, thereby eliminating the need for fractional coordinates.

Degenerate Cases Handling. In large-scale datasets, degenerate cases such as overlapped line segments for LSI or points on line segments for PIP may occur. These cases must be resolved for correctness. A well-known method for handling degenerate cases is Simulation of Simplicity (SOS) [9]. SOS eliminates all degeneracies by simulating the addition of an infinitesimal value to the coordinates. We employ SOS to handle degenerate cases.

Eliminating Invocation Overhead. Algorithm 1 and 2 conceptually introduce the implementation of RayJoin, where we utilize AnyHit and ClosestHit shaders to collect results. However, employing these shaders may introduce invocation overhead. To mitigate this, both AnyHit and ClosestHit shaders are disabled. We combine the logic of intersection testing and result collection into a single *IsIntersection* shader, thereby eliminating the overhead [46].

4.2 RT Cores Acceleration Enabled Applications

We also discovered a subset of *ST* functions in geodatabases that may benefit from RT Cores [21]. We only present the ideas of translating them into ray tracing problems. Implementing these operators with RayJoin in geo-database systems is beyond the scope of this paper, and we leave it as future work.

ST_Touches. Given two set of geometric objects A and B, the function *ST_Touches* returns true if *Condition* 1: A and B have at least on common point, and *Condition* 2: the common points cannot lie in the interior of A and B. Supposing A are polygons, we build a BVH from A. If B are points, we cast very short rays as approximations of the points to search hits. If B are polygons or polylines, we use Algorithm 1. This fetches all the potential touches, which can be filtered respected with *Condition 1*. The *Condition 2* can be verified with Algorithm 2.

ST_Intersects. This function returns true if the intersection of geometry A and B is not an empty set. We illustrate the idea due to limited space: a complex geometric object can always be split into primitives (point, line segment). We build a BVH with the primitives decomposed from A. Then we shoot rays depending on the type of primitives from B (a very short ray as a point or a long ray as a line segment). This allows us to collect potentially intersected geometries. Then, we filter the results that satisfy the intersection condition.

ST_Intersection. This function returns the intersected geometry from set A and B. This is one of the most complicated spatial functions and is very compute-intensive. Yet, it benefits from RayJoin. Assuming A and B are both polygons, which is the most complicated case, we (1) build two BVHs from A and B, (2) run the Algorithm 1 to find all intersections, (3) run Algorithm 2 to determine which polygon the points fall in, and (4) create the resulting polygon by connecting the points of intersection and the points that lie in the interior of polygons. This process is exactly the polygon overlay analysis, which is evaluated in §5.7.

5 EVALUATION

5.1 Evaluation Settings

Baselines. Table 1 lists the artifacts we have evaluated. We implemented the uniform grid and LBVH-based solutions on the GPU, which support all the queries [26]. cuSpatial is a GPU-accelerated geospatial analytical platform from NVIDIA [2]. RasterJoin is a raster-based solution on the GPU for spatial aggregation [79]. PSSL is a state-of-the-art plane-sweep-based algorithm for LSI on GPUs [15]. To the best of our knowledge, GLIN is the only learned spatial index that supports indexing geometries with extends [68]. With OpenMP, we use GLIN to implement a parallel LSI query. EPUG-Overlay is a parallel polygon overlay program with a two-level uniform grid [37]. Kinetica and PostGIS are geospatial databases [20]. RasterIntervals uses a raster-based method to accelerate polygon overlay query [16].

Datasets. We collected a diverse set of real-world datasets shown in Table 2, including US-based maps such as *County*, *Zipcode*, *Block*, and *Water* from ArcGIS Hub [11] and global maps, such as *Lakes* and *Parks* from OpenStreetMap (OSM) [10]. We partitioned the OSM datasets based on continents/regions. The spatial join operations were exclusively performed within datasets originating from the same geographic region. $R \bowtie S$ represents the join operation, where R is a base map and S is a query map. The spatial index is built from R, and queries are generated from S. For the scalability experiments, we use Spider to generate synthetic datasets [27]. The polygon size and segment length are set to 0.001 and 10, respectively. The Gaussian distribution is set to $\mu = 0.5, \sigma = 0.1$.

Parameters. Grid-based solutions need a predefined resolution, and trying all the possible resolutions to get the best is prohibitively expensive. By trials, we found that 15000x15000 for the uniform grid yields low running time for all datasets. For EPUG-Overlay, the first-level grid is set to 4000x4000, and the second-level grid is set to 40x40, a typical value used in [37]. RayJoin only needs one parameter, the merging threshold s, in the adaptive grouping algorithm, which is fixed to 3.5. We discuss how to find an optimal parameter in §5.5.

Environments. RayJoin is implemented in OptiX 8.0 and CUDA 12.3. We evaluate GPU-based solutions on a workstation with Intel Core i9-12900, NVIDIA RTX 3090, and 64 GB memory. cuSpatial version is 23.12.00. For the CPU-based solutions, we employed them on AWS EC2 using an r5.4xlarge instance with 16 vCPUs and 128 GB of RAM. The version of PostGIS is PostGIS-3.3, and Kinetica is v7.1.9.10-ga1. Note that Kinetica has GPU acceleration but it does not support our GPU model and requires a license purchase for self-hosted installation, so we have to evaluate its CPU version on AWS Marketplace Platform.

5.2 LSI Performance

Table 3 top shows the running time of LSI queries. For the conventional spatial indexes, LBVH is consistently better than the uniform grid. Our analysis attributes the grid-based method's inferior performance to skew distribution issues. For example, in the $County \bowtie Zipcode$, almost 100% of cells have workloads (number of intersection tests) under 223K per cell, accounting for 70.97% of total workloads. Yet, about 30% of the workloads are concentrated in just 90 cells (the total number of cells is 225M). PSSL's performance sits between the grid-based and tree-based methods, with its limited parallelism likely hindering the best utilization of the GPU. Notably, GLIN emerges as the least efficient method, with the majority of its processing time devoted to the query stage.

RayJoin demonstrates superior performance across all baselines. Focusing solely on processing time, RayJoin achieves speedups

Table 1: Evaluated artifacts, queries, implementation notes, and measured operations. (H2D: Host to Device)

Artifact	Evaluated Queries	Implementation Notes	Preprocessing Timing	Processing Timing
Uniform Grid	LSI, PIP, Polygon Overlay	Uniform Grid in CUDA	H2D, Build Grid	Search Index, Compute Overlay
LBVH	LSI, PIP, Polygon Overlay	LBVH in CUDA	H2D, Build LBVH	Search Index, Compute Overlay
cuSpatial	PIP	Quadtree-based in CUDA	Build Quadtree	Join Quadtree with MBR, Point in Polygon
RasterJoin	PIP	Rasterization-based in OpenGL	H2D, Polygon Triangulation	Rendering
PSSL	LSI	Plane-sweep-based in CUDA	H2D, Compute Worklist	Compute Intersections
GLIN	LSI	Learned Spatial Index	Bulk Loading	Search Index, Compute Intersections
EPUG-Overlay	Polygon Overlay	Two-level uniform grid in OpenMP	Build Grid	Search Index, Compute Overlay
Kinetica	Polygon Overlay	In-memory spatial database	-	SQL Execution
PostGIS	Polygon Overlay	Disk-based spatial database	-	SQL Execution
RasterIntervals	Polygon Overlay	Rasterization-based in OpenMP	Generate Raster Intervals	MBR Filter, Raster Intervals Filter, Refinement
RayJoin	LSI, PIP, Polygon Overlay	HW-accelerated BVH in CUDA+OptiX	H2D, AG, Build BVH	Ray Tracing, Compute Overlay

Table 2: Statistics of real-world datasets

Dataset	Line Segs	Polygons	Description
County	1.0M	3.1K	Boundaries of the U.S. Counties
Zipcode	23.9M	32.2K	ZIP Code areas for the USPS
Block	29.3M	239.2K	Census block groups in the U.S.
Water	25.6M	463.6K	Major water features in the U.S.
LKAF	1.8M	18.2K	Water areas in Africa
PKAF	1.3M	25.7K	Parks or green areas in Africa
LKAS	10.3M	151.6K	Water areas in Asia
PKAS	11.9M	172.6K	Parks or green areas in Asia
LKAU	1.2M	14.5K	Water areas in Australia
PKAU	567.1K	12.8K	Parks or green areas in Australia
LKEU	27.9M	654.8K	Water areas in Europe
PKEU	65.9M	1.9M	Parks or green areas in Europe
LKNA	69.3M	1.6M	Water areas in North America
PKNA	26.9M	303.0K	Parks or green areas North America
LKSA	2.4M	32.6K	Water areas in South America
PKSA	3.2M	49.5K	Parks or green areas in South America

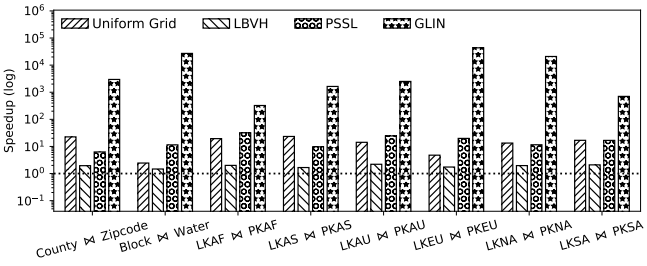


Figure 10: Summary of Rayjoin's LSI speedups over baselines by considering total execution time.

ranging from 1.7x to 6.0x over the eight datasets compared to the best baseline performances. When including factors like loading time, adaptive grouping, and BVH construction costs, RayJoin still achieves speedups between 1.5x and 2.4x. We summarize the speedups in Figure 10.

5.3 PIP Performance

The bottom half of Table 3 reveals that the uniform grid surpasses LBVH for large datasets, although its performance is hindered by skewed data. The grid's advantage lies in its ability to partition space, allowing for the termination of searches as soon as the closest line segment to a query point is located. In contrast, LBVH requires examining all intersecting line segments to identify the closest one. RasterJoin, however, shows underperformance compared to both grid and LBVH and runs time out on $LKNA \bowtie PKNA$ in the polygon triangulation stage. For instance, in the $County \bowtie Zipcode$ dataset, RasterJoin spent approximately 6878ms on point rendering and 5371ms on polygon rendering, a likely consequence of rendering excessive primitives in large datasets. Notably, NVIDIA's spatial library fell short of the LBVH and ran out-of-memory (OOM) on the $LKEU \bowtie PKEU$ dataset.

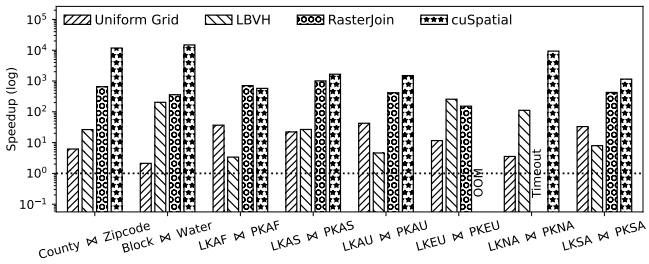


Figure 11: Summary of Rayjoin's PIP speedups over baselines by considering total execution time.

PIP is an RT-favorable workload because only the closest hit needs to be found. This property allows us to leverage the ClosestHit shader supported by the ray-tracing hardware to reduce the amount of work instead of traversing all the geometries. If considering the processing time only (ray-tracing), RayJoin exhibited from 4.8x to 74.2x speedups over the best of baselines. As shown in Figure 11, by considering the total running time, RayJoin achieved from 2.1x to 22.2x speeds over the best performance of baselines.

5.4 Evaluation of Precision

The conservative representation ensures results identical to those of the program using FP64 or higher. To validate its precision, we compared the results of RayJoin to the grid implementation in FP64 on LKNA ⋈ PKNA. The LSI error rate is defined by $\frac{|Missed\ Intersections|}{|Intersections|}$, and the PIP error rate is defined by $\frac{|Wrongly\ Identified\ Point|}{|Query\ Points|}$. Executing LSI and PIP queries with OptiX's default FP32 precision resulted in 745 intersections missing (out of 1,251,343) for LSI and 524 incorrect responses (out of 69,273,661) for PIP, yielding error rates of 0.0595% and 0.0008%, respectively. When using the conservative representation to create the AABBs, both LSI and PIP produce

Table 3: Performance numbers of LSI and PIP queries. The cell represents "*Processing Time* (*Preprocessing Time*) in milliseconds." (All methods are GPU-based except GLIN.)

Query	Artifact	County ⋈ Zipcode	Block ⋈ Water	LKAF ⋈ PKAF	LKAS ⋈ PKAS	LKAU ⋈ PKAU	LKEU ⋈ PKEU	LKNA ⋈ PKNA	LKSA ⋈ PKSA
LSI	Uniform Grid	1316 (81)	210 (127)	90 (46)	916 (65)	37 (34)	568 (200)	2577 (194)	122 (63)
	LBVH	61 (60)	80 (123)	2 (12)	15 (54)	3 (9)	115 (166)	89 (319)	6 (17)
	PSSL	147 (242)	1172 (406)	30 (193)	175 (231)	41 (80)	2554 (636)	1766 (631)	69 (115)
	GLIN	183668 (73)	3732430 (1055)	2208 (60)	68352 (306)	12459 (59)	6991691 (1037)	4333675 (2836)	7632 (101)
	RayJoin	**17 (45)**	**48 (91)**	**1 (6)**	**4 (37)**	**1 (4)**	**27 (134)**	**30 (177)**	**1 (10)**
PIP	Uniform Grid	341 (74)	139 (117)	257 (36)	1113 (63)	178 (34)	2785 (157)	624 (180)	414 (49)
	LBVH	1709 (61)	24424 (122)	14 (12)	1360 (55)	15 (9)	64661 (166)	25029 (325)	93 (17)
	RasterJoin	12248 (31196)	21657 (21641)	1016 (4647)	36755 (16626)	155 (1916)	10925 (27593)	Timeout	969 (4986)
	cuSpatial	787691 (101)	1795798 (74)	4599 (11)	87942 (43)	7493 (9)	OOM	2120454 (70)	16109 (27)
	RayJoin	**21 (46)**	**29 (92)**	**1 (6)**	**15 (38)**	**1 (4)**	**117 (136)**	**51 (176)**	**4 (10)**

completely the same answers compared to the FP64 program, and we have verified the correctness of all datasets. These experiments confirm the analysis in §3.2, and our solution eliminates all the false negative cases due to the precision loss.

5.5 Effectiveness of Adaptive Grouping and Parameter Tuning

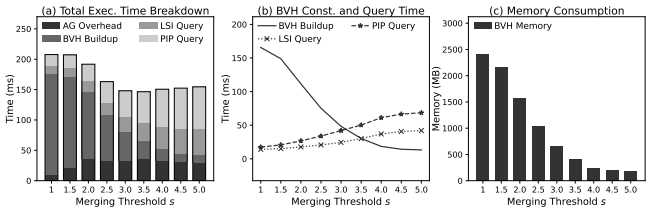


Figure 12: Execution time breakdown on $LKNA \bowtie PKNA$ datasets by varying merging threshold s

Figure 12 (a) illustrates the variation in BVH construction time, query time, and AG overhead when joining the *LKNA* with *PKNA* dataset (Table 2) as parameter s changes. When s is set to 1, AG is effectively disabled because two line segments are allowed to be put together only if the area of the merged bounding box does not grow, leading to high query performance but also high BVH construction time and memory usage – 166ms and 2403MB, respectively (Figure 12 (b-c)). Here, LSI and PIP queries take just 14ms and 18ms.

As s increases, construction costs drop significantly, but query times for both LSI and PIP begin to rise. At $s = 3.5$, AG achieves a 29% reduction in total execution time (147ms) and drastically lowers memory usage to about 408MB, which is just 17% of the usage without AG. Beyond $s = 3.5$, total time starts to increase, reaching 155ms at $s = 5.0$. This rise is attributed to AG merging spatially distant bounding boxes, causing a higher false positive rate and more intersection tests.

Figure 12 (b) demonstrates a linear correlation between construction and query times with s, particularly in the range of $1.5 \leq s \leq 4$. This trend is consistent across various datasets. Optimal parameter selection could be determined through linear regression modeling of the merging threshold-execution time relationship, though this process is omitted here due to space constraints.

5.6 Scalability

We generated synthetic datasets with uniform and Gaussian distributions. The BVH is built from dataset R with 5M polygons. A series of datasets S, ranging from 1M to 5M polygons, were generated to evaluate RayJoin's scalability. To better understand the scalability of RayJoin, adaptive grouping was disabled, and BVH construction time was not included; only query time was reported.

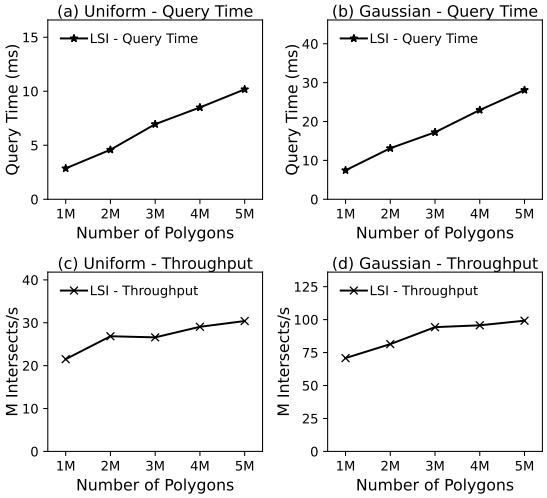


Figure 13: LSI scalability: R is fixed to 5M polygons, varying S from 1M to 5M

Figure 13 (a)-(d) shows LSI query time and throughput for increasing polygon numbers. Under uniform distribution, 1M polygons result in a 2.9ms query time and 62K intersections. At 3M polygons, the time rises to 6.9ms with 185K intersections, indicating RayJoin's linear scalability, aligning well with the time complexity of $O(|S|log|R|)$. Gaussian distributions yield similar patterns but with longer times due to more intersections. When the number of queries increases from 1M to 2M polygons, LSI throughput stabilizes at 29M intersections/s. In the Gaussian dataset, throughput peaks at 99M intersections/s.

For 1M uniform polygons, 8M PIP queries are issued, taking about 5.9ms (Figure 14 (a)). At 2M polygons and 16M queries, the time increases to 12.1ms. In Gaussian distributions, query times are slightly longer due to more ray-primitive intersections. RayJoin

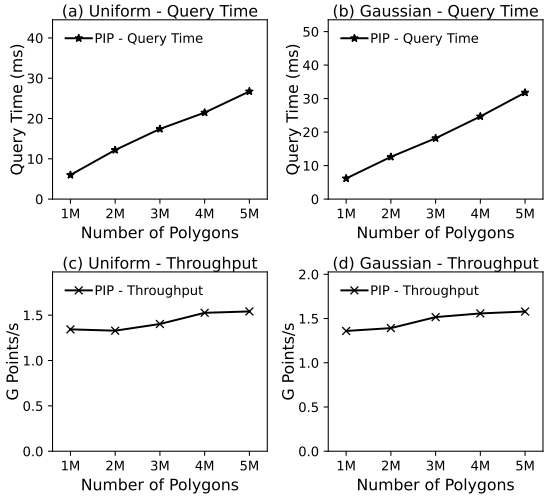


Figure 14: PIP scalability: R is fixed to 5M polygons, varying S from 1M to 5M

shows linear scalability with query numbers. Figure 14 (c), (d) indicates a peak PIP throughput of around 1.5G points/s for both distributions, affirming RayJoin is highly scalable as throughput remains consistent despite increased queries.

5.7 Polygon Overlay

The polygon overlay query identifies and outputs the overlapped area of intersected polygons from two maps, which can be achieved by combining the LSI and PIP query results (§2.1). The overlay processing is extremely time-consuming due to the complexity of the overlay algorithm and the irregularity of the execution.

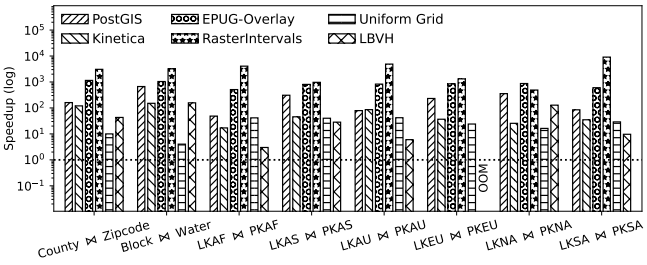


Figure 15: Summary of RayJoin's polygon overlay speedups over baselines by considering total execution time.

Table 4 shows the polygon overlay running time. It is evident that none of the existing solutions other than RayJoin take under one second for all datasets, even the highly optimized GPU solutions (uniform grid and LBVH). The longest execution time of PostGIS is about 234s on the $Block \bowtie Water$ dataset. Kinetica is faster than PostGIS because it is memory-based, only taking about 53s. EPUG-Overlay typically takes more than hundreds of seconds for fairly large datasets. Although RasterIntervals is fast at the processing stage, it can spend thousands of seconds on preprocessing. The uniform grid takes up to 9.4s for processing $LKEU \bowtie PKEU$, and LBVH even runs OOM on this dataset. LBVH takes up to 58.4s to process the $LKNA \bowtie PKNA$ dataset due to irregular random memory access patterns of tree-based structure and the inability to terminate the PIP search early.

RayJoin outperformed all the state-of-the-art solutions on both CPUs and GPUs. By leveraging the RT Cores, RayJoin efficiently transforms the polygon overlay problem into two RT-friendly problems. The RT Cores overcome the long memory access latency and irregular memory access patterns while benefiting from the logarithmic complexity of tree-based indexes, resulting in extremely low query time. As summarized in Figure 15, RayJoin achieved from 3.0x to 28.3x speed up over the best of the baselines.

6 RELATED WORK

Spatial Indexing on Advanced Hardware. Geometric Performance Primitives (GPP) is a robust polygon overlay library on GPUs with a uniform grid [6]. Liu et al. proposed a filtering technique to accelerate spatial join with fine-grained tiles [33, 34]. Rayhan et al. designed an R-tree that leverages SIMD instructions [55]. SwiftSpatial is a spatial join accelerator on FPGAs [23]. Note that we target the same problem but with different hardware. In short, RayJoin is an index-based solution uniquely exploiting ray-tracing hardware's capability to deliver real-time spatial join.

Rasterization-based Techniques. RasterJoin is a rasterization-based method for spatial aggregation by leveraging the rendering pipeline [79]. RasterIntervals is the state-of-the-art polygon overlay method with the approximation technique to reduce the refinement cost [16]. PixelBox employs a rasterization-based method to compute the Jaccard Similarity [70]. IDEAL is a vector-raster hybrid method for PIP and point-to-polygon distance calculation [65]. Compared to rasterization-based methods, RayJoin does not need any preprocessing steps to build up rasterized representations.

Parallel Plane Sweep Algorithms. McKenney et al. proposed a parallel plane sweep algorithm on CPUs[41] that dynamically segments the input to spatial operations and executes them in parallel. PSSL is the plane sweep-based solution for the LSI query on the GPU [15]. The plane sweep algorithms are challenging to parallelize, as demonstrated by our evaluations, which clearly indicate that they exhibit significantly lower performance than other solutions.

Accelerating Non-graphics Workload with RT Cores. An early work on repurposing RT Cores, proposed by Salmon et al., involves accelerating Monte Carlo particle transport simulations [57]. RTNN adopted spatial reordering and query partitioning optimizations for K nearest neighbor (KNN). TrueKNN supports arbitrary radius KNN-search on RT Cores [46]. JUNO is the first work supporting high-dimensional KNN search on RT Cores [35]. A method presented in [38] supports non-Euclidean distance measures for KNN. A method proposed in [30] accelerates the PIP test by transforming polygons into 3D representations, which is different from our method and has accuracy limitations. Accelerating Tet-Mesh Point Location with RT Cores is discussed in [67]. Literature [24] explores using RT Cores to serve as a B-Tree. RTScan leverages RT Cores to accelerate index scans [36]. Accelerating Range Minimum Queries with RT Cores is discussed in [42]. None of these works address the spatial join problem that our paper targets, which presents unique challenges such as problem formulation, precision preservation, and performance optimization.

Table 4: Polygon overlay execution time. The cell represents "Processing Time (Preprocessing Time) in seconds." (∗: GPU-based)

Artifact	County ⋈ Zipcode	Block ⋈ Water	LKAF ⋈ PKAF	LKAS ⋈ PKAS	LKAU ⋈ PKAU	LKEU ⋈ PKEU	LKNA ⋈ PKNA	LKSA ⋈ PKSA
PostGIS	30.58	233.78	0.97	27.78	0.79	92.58	163.42	2.54
Kinetica	22.81	52.75	0.34	4.10	0.86	14.69	11.75	1.05
EPUG-Overlay	165.71 (52.64)	250.21 (114.46)	4.94 (5.26)	37.41 (35.22)	4.74 (3.54)	177.61 (161.91)	234.53 (166.83)	8.30 (9.75)
RasterIntervals	2.94 (580.40)	10.54 (1114.52)	0.14 (81.20)	5.29 (82.03)	0.12 (48.72)	15.47 (511.82)	21.52 (203.45)	0.45 (272.04)
Uniform Grid*	1.82 (0.08)	1.29 (0.13)	0.78 (0.05)	3.54 (0.07)	0.38 (0.03)	9.40 (0.20)	7.29 (0.20)	0.81 (0.06)
LBVH*	8.07 (0.11)	54.82 (0.18)	0.05 (0.02)	2.46 (0.09)	0.05 (0.01)	OOM	58.40 (0.47)	0.26 (0.03)
RayJoin*	**0.12 (0.07)**	**0.23 (0.12)**	**0.01 (0.01)**	**0.04 (0.05)**	**0.01 (0.01)**	**0.20 (0.20)**	**0.25 (0.21)**	**0.02 (0.01)**

7 CONCLUSION

We have designed and implemented RayJoin, a high-performance spatial join engine for real-time applications. By leveraging RT Cores, RayJoin effectively resolves the longstanding performance bottlenecks associated with conventional spatial join methods running on traditional architectures. Additionally, our solution addresses the underlying hardware precision concerns to ensure accurate results subject to retaining high performance, which can be widely used in other RT-based spatial data processing applications. Through intensive experiments, we have demonstrated that RayJoin achieves super high processing speeds, with spatial join operations completed within milliseconds. Our research outcome opens up new possibilities for real-time applications that rely on fast and accurate spatial processing.

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] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229
[2] 2023. cuSpatial. https://docs.rapids.ai/api/cuspatial/stable/
[3] Laila Abdelhafeez, Amr Magdy, and Vassilis J Tsotras. 2023. SGPAC: generalized scalable spatial GroupBy aggregations over complex polygons. *GeoInformatica* (2023), 1–28.
[4] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. 2013. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1009–1020.
[5] Mikhail J Atallah and Michael T Goodrich. 1986. Efficient plane sweeping in parallel. In *Proceedings of the second annual symposium on Computational geometry*. 216–225.
[6] 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*. 304–313.
[7] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial joins: what's next? *SIGSPATIAL Special* 11, 1 (2019), 13–21.
[8] John Burgess. 2020. Rtx on—the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.
[9] Herbert Edelsbrunner and Ernst Peter Mücke. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (tog)* 9, 1 (1990), 66–104.
[10] 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.
[11] Esri. 2023. *ArcGIS Hub*. Retrieved Feb 21, 2023 from https://hub.arcgis.com
[12] Ulrich Finke and Klaus Hinrichs. 1993. A spatial data model and a topological sweep algorithm for map overlay. In *International Symposium on Spatial Databases*. Springer, 162–177.
[13] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2021. NestGPU: Nested query processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1008–1019.
[14] William Randolph Franklin, Venkateshkumar Sivaswami, David Sun, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. 1994. Calculating the area of overlaid polygons without constructing the overlay. *Cartography and Geographic Information Systems* 21, 2 (1994), 81–89.
[15] Roger Frye and Mark McKenney. 2022. Per segment plane sweep line segment intersection on the GPU. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*. 1–4.
[16] Thanasis Georgiadis and Nikos Mamoulis. 2023. Raster Intervals: An Approximation Technique for Polygon Intersection Joins. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–18.
[17] 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.
[18] Ali Hadian and Thomas Heinis. 2019. Considerations for handling updates in learned index structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.
[19] Herman Johannes Haverkort. 2004. *Results on geometric networks and data structures*. Ph. D. Dissertation.
[20] Kinetica DB Inc. 2023. Kinetica: The Database for Time & Space. https://www.kinetica.com/
[21] ISO/IEC. 2021. ISO/IEC 13249-3:2016. https://www.iso.org/standard/60343.html
[22] Thiago Ize. 2013. Robust BVH ray traversal. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (2013), 12–27.
[23] Wenqi Jiang, Martin Parvanov, and Gustavo Alonso. [n. d.]. SwiftSpatial: Spatial Joins on Modern Hardware. ([n. d.]). https://arxiv.org/pdf/2309.16520.pdf
[24] Benjamin Kahl. 2023. Hardware Acceleration of Progressive Refinement Radiosity using Nvidia RTX. *arXiv preprint arXiv:2303.14831* (2023).
[25] Charles FF Karney. 2011. Transverse Mercator with an accuracy of a few nanometers. *Journal of Geodesy* 85 (2011), 475–485.
[26] 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.
[27] 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.
[28] S. Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *Proceedings of High Performance Graphics* (Lyon, France) *(HPG '14)*. Eurographics Association, Goslar, DEU, 29–40.
[29] 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.
[30] 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.
[31] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2999–3013.
[32] Yun Li, Yixiang Fang, Reynold Cheng, and Wenjie Zhang. 2019. Spatial pattern matching: A new direction for finding spatial objects. *SIGSPATIAL Special* 11, 1 (2019), 3–12.
[33] Yiming Liu and Satish Puri. 2020. Efficient filters for geometric intersection computations using gpu. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. 487–496.
[34] Yiming Liu, Jie Yang, and Satish Puri. 2019. Hierarchical filter and refinement system over large polygonal datasets on cpu-gpu. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 141–151.
[35] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2023. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and

Ray-Tracing Core Mapping. *arXiv preprint arXiv:2312.01712* (2023).
[36] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. [n. d.]. RTScan: Efficient Scan with Ray Tracing Cores. ([n. d.]).
[37] Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Wenli Li. 2015. Fast exact parallel map overlay using a two-level uniform grid. In *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. 45–54.
[38] Durga Mandarapu, Vani Nagarajan, and Milind Kulkarni. 2023. Generalized Neighbor Search using Commodity Hardware Acceleration. *arXiv preprint arXiv:2311.09168* (2023).
[39] Adam Marrs, Peter Shirley, and Ingo Wald. 2021. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Springer Nature.
[40] Mark McKenney, Gabriel De Luna, Schiller Hill, and Logan Lowell. 2011. Geospatial overlay computation on the GPU. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 473–476.
[41] Mark McKenney and Tynan McGuire. 2009. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 392–395.
[42] Enzo Meneses, Cristóbal A Navarro, Héctor Ferrada, and Felipe A Quezada. 2023. Accelerating Range Minimum Queries with Ray Tracing Cores. *arXiv preprint arXiv:2306.03282* (2023).
[43] Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. 2023. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 191–204.
[44] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.
[45] Kyriakos Mouratidis, Jing Li, Yu Tang, and Nikos Mamoulis. 2014. Joint search by social and spatial proximity. *IEEE Transactions on Knowledge and Data Engineering* 27, 3 (2014), 781–793.
[46] 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.
[47] Jürg Nievergelt and Franco P Preparata. 1982. Plane-sweep algorithms for intersecting geometric figures. *Commun. ACM* 25, 10 (1982), 739–747.
[48] 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.
[49] 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.
[50] NVIDIA. 2024. NVIDIA OptiX 8.0 Programming Guide. https://raytracing-docs.nvidia.com/optix8/guide/optix_guide.240111.A4.pdf
[51] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673.
[52] Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. 2020. The Case for Learned Spatial Indexes. In *AIDB@VLDB 2020, 2nd International Workshop on Applied AI for Database Systems and Applications, Held with VLDB 2020, Monday, August 31, 2020, Online Event / Tokyo, Japan*, Bingsheng He, Berthold Reinwald, and Yingjun Wu (Eds.).
[53] 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.
[54] Urs Ramer. 1972. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing* 1, 3 (1972), 244–256.
[55] Yeasir Rayhan and Walid G Aref. 2023. SIMD-ified R-tree Query Processing and Optimization. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*. 1–10.
[56] Marcel Roeloffzen. 2014. *Point Location*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–5. https://doi.org/10.1007/978-3-642-27848-8_587-1
[57] 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.
[58] Philip J Schneider and Barbara A Schauer. 2006. HAZUS—Its development and its future. *Natural Hazards Review* 7, 2 (2006), 40–44.
[59] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3950–3962.
[60] Moshe Shimrat. 1962. Algorithm 112: position of point relative to polygon. *Commun. ACM* 5, 8 (1962), 434.
[61] 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.
[62] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM RUM-tree: a log structured merge R-tree for update-intensive spatial workloads. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 2285–2290.
[63] Darius Šidlauskas, Simonas Šaltenis, and Christian S Jensen. 2012. Parallel main-memory indexing for moving-object query and update workloads. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 37–48.
[64] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. 2016. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1565–1568.
[65] Dejun Teng, Furqan Baig, Qiheng Sun, Jun Kong, and Fusheng Wang. 2021. IDEAL: a Vector-Raster Hybrid Model for Efficient Spatial Queries over Complex Polygons. In *2021 22nd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 99–108.
[66] Jan W van Roessel. 1991. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems* 18, 1 (1991), 49–67.
[67] 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.. In *High Performance Graphics (Short Papers)*. 7–13.
[68] 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.
[69] 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.
[70] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. 2012. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 5. NIH Public Access, 1543.
[71] Randall T Whitman, Michael B Park, Sarah M Ambrose, and Erik G Hoel. 2014. Spatial indexing and analytics on hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*. 73–82.
[72] Andrew Wooler. 2021. Air Force's Digital Directorate Awards Kinetica Contract with $100M Ceiling for Real-Time Intelligence. https://www.kinetica.com/blog/air-force-contract/
[73] 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.
[74] 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.
[75] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 international conference on management of data*. 1071–1085.
[76] Simin You, Jianting Zhang, and Le Gruenwald. 2015. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*. IEEE, 34–41.
[77] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 1–4.
[78] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828.
[79] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freire. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.
[80] Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. 2023. Adaptive Indexing of Objects with Spatial Extent. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2248–2260.
[81] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1226–1237.
[82] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. 2021. SPRIG: A learned spatial index for range and kNN queries. In *17th International Symposium on Spatial and Temporal Databases*. 96–105.
[83] 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.