Accelerating Spatial Data Processing by Repurposing Ray Tracing Cores

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Liang Geng, B.E., M.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2026

Dissertation Committee:

Dr. Xiaodong Zhang, Advisor

Dr. Suren Byna

Dr. Hanqi Guo

Dr. Martin Kong (Moreno)

© Copyright by

Liang Geng

2026

Abstract

Fast spatial data processing is vital for location-based services, autonomous navigation, and geospatial analytics. However, processing large-scale spatial datasets is computationally intensive due to complex geometric computation and high volume of geometries. While spatial systems rely on tree-based indexes to prune the search space, and turn to GPUs for parallel compute, traversing tree structures on GPUs is inherently inefficient due to pointer chasing, branch divergence, and irregular memory accesses. Consequently, traditional GPU-accelerated spatial processing struggles to achieve high performance.

Recent GPU architectures have introduced Ray Tracing (RT) cores, designed to accelerate Bounding Volume Hierarchy (BVH) traversal and ray-primitive intersection tests with dedicated circuits, delivering order-of-magnitude speedups over software-based approaches. Recent research shows that RT cores can also benefit non-rendering workloads by reformulating them as ray tracing tasks. Given the similarity in execution patterns, spatial data processing tasks may also benefit from using RT cores. This dissertation presents a comprehensive suite of hardware-accelerated solutions spanning core spatial operators to end-to-end database engines.

We first present RayJoin, a real-time spatial join framework using RT cores. RayJoin accelerates two fundamental queries: Line Segment Intersection (LSI) and Point-In-Polygon (PIP) tests. It also tackles two challenges: (1) achieving high-precision results despite hardware precision limitations, and (2) reducing BVH construction costs while maintaining high query performance. Evaluations

show RayJoin achieves 3.0× to 28.3× speedups over existing highly optimized methods under high-precision settings.

Some spatial workloads demand fast point and range queries supported by spatial indexing. To address this, we propose LibRTS, a general-purpose spatial indexing library that leverages RT cores. By reformulating queries as RT-friendly tasks and addressing load-balancing challenges, LibRTS achieves high performance without requiring developers to manage RT-specific programming complexities. LibRTS achieves speedups of up to 85.1× for point queries, 94.0× for range-contains queries, and 11.0× for range-intersects queries. In a real-world application, the PIP test, LibRTS also surpasses the state-of-the-art RT method by up to 3.8×.

Third, we introduce X-HD, an RT-accelerated solution for computing the Hausdorff Distance (HD), a critical spatial measurement used in computer vision, medical imaging, geospatial analysis, and CAD. We reformulate HD computation as a nearest neighbor search to leverage hardware-accelerated BVH traversal. We apply a grid-based partitioning strategy that groups spatially proximate points into the same cell and adopt pruning techniques such as early-break. As a result, X-HD outperforms the state-of-the-art industrial software ITK by 5.3× on average. Compared to a GPU-optimized HD solution, X-HD achieves a speedup of up to 6.4×.

Finally, to bridge the gap between individual spatial operator acceleration and an industrial spatial database system, we present RayBooster, an end-to-end RT-acceleration solution integrated into Apache SedonaDB. RayBooster introduces a GPU-friendly memory layout for random geometry access, a monolithic BVH construction scheme to reduce index build-up costs, and a unified RelateEngine evaluating DE-9IM topological matrices as its core components. RayBooster achieves up to 5.93× speedup over SedonaDB on SpatialBench while reducing operational costs by 59.02%.

To my family, for your endless love, patience, and unwavering belief in me.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Xiaodong Zhang, for his continuous guidance, patience, and support throughout my Ph.D. journey. His profound technical insight, rigorous standards for research, and unwavering encouragement have fundamentally shaped how I approach computer systems research. I am equally indebted to my co-advisor, Dr. Rubao Lee, and former co-advisor, Dr. Hao Wang, whose hands-on mentorship, insightful technical discussions, and pragmatic problem-solving advice were instrumental in bringing this work to fruition. Working under their joint guidance has been an immense privilege, and I am especially grateful for the vibrant atmosphere of our group meetings, where conversations extended far beyond technical debates to eye-opening discussions on world events, emerging technology trends, and broader societal perspectives that broadened my horizons well beyond research.

I would like to thank Dr. Mo Sarwat, CEO, and Dr. Jia Yu, CTO of Wherobots, for giving me the opportunity to work with their engineering team to integrate our research prototypes, RayJoin and LibRTS, into their production spatial database, SedonaDB. I really enjoyed the collaboration with software engineers Dr. Dewey Dunnington, Dr. Feng Zhang, and others, and especially appreciated the opportunity to work together on addressing challenging real-world problems and translating research ideas into a production system.

I also want to express my appreciation to my labmates, Dayi Fan and Jiaxin Liu, for creating such an engaging and collaborative lab environment.

I would also like to thank the members of my dissertation committee, Dr. Suren Byna, Dr. Hanqi Guo, and Dr. Martin Kong, for providing constructive comments that substantially improved the quality of this dissertation.

This research was supported in part by the U.S. National Science Foundation under grants MRI-2018627, CCF-2005884, CCF-2210753, CCF-2312507, and OAC-2310510. I also gratefully acknowledge the financial support of the Presidential Graduate Fellowship from The Ohio State University.

Finally, this achievement would have been impossible without the unconditional love and support of my family. To my parents, thank you for your lifelong sacrifices, unwavering belief in my abilities, and the foundation you built for me. Most of all, to my wife, Jiamu Yu, thank you for your boundless love, endless patience, and constant companionship. You stood by me through every deadline, setback, and breakthrough, keeping me grounded and motivated every single day.

Vita

January 2022–Present	Ph.D. Student, Computer Science and Engineering, The Ohio State University, USA.
April 2026	Graduate Research Award, The Ohio State University, USA.
November 2025	Presidential Fellowship, The Ohio State University, USA.
June 2025–August 2025	Software Engineer Intern, Wherobots Inc., USA.
April 2024	Best Poster Award, CSE Graduate Student Research Exhibition, The Ohio State University, USA.
January 2022–December 2025	Graduate Research Associate, Computer Science and Engineering, The Ohio State University, USA.
June 2020–December 2021	Senior Development Engineer, Alibaba Group, China.
January 2019–May 2020	Software Engineer, Chehaoduo Group, China.
December 2017–September 2018	Visiting Research Associate, Computer Science and Engineering, The Ohio State University, USA.
September 2016–January 2019	M.E. Computer Science and Engineering, Northeastern University, China.
September 2012–June 2016	B.E. Software Engineering, Liaoning Technical University, China.

Publications

Research Publications

Liang Geng, Rubao Lee, Dewey Dunnington, Feng Zhang, Jia Yu, and Xiaodong Zhang. RayBooster: A Ray Tracing Engine to Accelerate SedonaDB. In *Proceedings of the VLDB Endowment (VLDB Industry Track)*, 2026.

Liang Geng, Zhehu Yuan, Rubao Lee, Fusheng Wang, and Xiaodong Zhang. X-HD: Fast Hausdorff Distance Computation with Ray Tracing. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2026.

Liang Geng, Rubao Lee, and Xiaodong Zhang. LibRTS: A Spatial Indexing Library by Ray Tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2025.

Liang Geng, Rubao Lee, and Xiaodong Zhang. RayJoin: Fast and Precise Spatial Join. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2024.

Liang Geng, Hao Wang, Jingsong Meng, Dayi Fan, Sami Ben-Romdhane, Hari Kadayam Pichumani, Vinay Phegade, and Xiaodong Zhang. RR-Compound: RDMA-Fused gRPC for Low Latency and High Throughput With an Easy Interface. In *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2024.

Shufeng Gong, Chao Tian, Qiang Yin, Zhengdong Wang, Song Yu, Yanfeng Zhang, Wenyuan Yu, **Liang Geng**, Chong Fu, Ge Yu, and Jingren Zhou. Ingress: An Automated Incremental Graph Processing System. In *The VLDB Journal (VLDBJ)*, 2024.

Ke Meng, **Liang Geng**, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *IEEE 39th International Conference on Data Engineering (ICDE)*, 2023.

Wenfei Fan, **Liang Geng**, Ruochun Jin, Ping Lu, Resul Tugay, and Wenyuan Yu. Linking Entities across Relations and Graphs. In *IEEE 38th International Conference on Data Engineering (ICDE)*, 2022.

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

Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, **Liang Geng**, Song Yu, Ge Yu, and Jingren Zhou. Automating Incremental Graph Processing with Flexible Memoization. In *Proceedings of the VLDB Endowment (VLDB)*, 2021.

Qiange Wang, Yanfeng Zhang, Hao Wang, **Liang Geng**, Rubao Lee, Xiaodong Zhang, and Ge Yu. Automating Incremental and Asynchronous Evaluation for Recursive Aggregate Data Processing. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020.

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

Simon Zhang, Mengbai Xiao, Chengxin Guo, **Liang Geng**, Hao Wang, and Xiaodong Zhang. Hypha: A Framework based on Separation of Parallelisms to Accelerate Persistent Homology Matrix Reduction. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.

Hao Wang, **Liang Geng**, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. Sep-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vii

List of Tables xiv

List of Figures xv

Chapter 1: Introduction 1

1.1 Fast and Precise Spatial Join on RT Cores 3
1.2 Designing a General-Purpose Spatial Indexing Library with Ray Tracing 4
1.3 RT for Spatial Functions (Case Study: Hausdorff Distance) 4
1.4 End-to-End RT Acceleration for Spatial Databases 5

Chapter 2: Fast and Precise Spatial Join on RT Cores 7

2.1 Introduction 7
2.2 Background 10
2.2.1 Spatial Join Queries and Spatial Indexes 10
2.2.2 Ray-Tracing and Its Programming Model 14
2.3 Spatial Join Operations on RT Cores 16
2.3.1 From Spatial Join to Ray-tracing 16
2.3.2 High Arithmetic Precision Beyond the Hardware Support 21
2.3.3 Taming High BVH Construction Overhead 25
2.4 RayJoin Deployment In Practice 27
2.4.1 Implementation Details 27

2.4.2 RT Cores Acceleration Enabled Applications . . . 28
2.5 Evaluation . . . 29
2.5.1 Evaluation Settings . . . 29
2.5.2 LSI Performance . . . 32
2.5.3 PIP Performance . . . 33
2.5.4 Evaluation of Precision . . . 34
2.5.5 Effectiveness of Adaptive Grouping and Parameter Tuning . . . 35
2.5.6 Scalability . . . 36
2.5.7 Polygon Overlay . . . 37
2.6 Related Work . . . 39
2.7 Conclusion . . . 41

Chapter 3: A Spatial Indexing Library by Ray Tracing . . . 42

3.1 Introduction . . . 42
3.2 Background . . . 45
3.2.1 Spatial Index . . . 45
3.2.2 Ray Tracing . . . 46
3.2.3 Acceleration Structure . . . 48
3.2.4 Programming Model for RT Cores . . . 49
3.3 From Spatial Query to Ray-tracing . . . 51
3.3.1 Point Query . . . 51
3.3.2 Range Query with *Contains* Predicate . . . 52
3.3.3 Range Query with *Intersects* Predicate . . . 53
3.3.4 Ray Multicast for Load Balancing . . . 58
3.4 Implementation and Interface for Users . . . 60
3.5 Update Spatial Index . . . 61
3.5.1 Insert . . . 61
3.5.2 Update and Delete . . . 62
3.6 Evaluation . . . 63
3.6.1 Evaluation Setup . . . 63
3.6.2 Point Query . . . 65
3.6.3 Range Query with Contains Predicate . . . 66
3.6.4 Range Query with Intersects Predicate . . . 67
3.6.5 Effectiveness of Load Balancing . . . 68
3.6.6 Update Performance . . . 70
3.6.7 Sensitivity to Updates . . . 71
3.6.8 Scalability . . . 71
3.6.9 Real-world Application: PIP . . . 72
3.7 Related Work . . . 73
3.8 Conclusion . . . 75

Chapter 4: A Fast Algorithm for Hausdorff Distance by Ray Tracing 76

4.1 Introduction 76
4.2 Background 79
4.2.1 Hausdorff Distance 79
4.2.2 HD Algorithms 80
4.2.3 RT-accelerated Nearest Neighbor Search 81
4.3 Design Principles 83
4.3.1 Reducing BVH Size 83
4.3.2 Reducing Redundant Computation 85
4.3.3 Avoiding Heavy Workload in Shaders 86
4.4 The X-HD Algorithm 87
4.4.1 Algorithm Overview 87
4.4.2 Iterative Nearest Neighbor Search 88
4.4.3 Culling Non-contributing Points 91
4.4.4 Offloading Heavy Cells 93
4.5 Implementation Details 95
4.6 Evaluation 98
4.6.1 Settings 98
4.6.2 Overall Performance 101
4.6.3 Effectiveness of Pruning Techniques 103
4.6.4 Effectiveness of Load-balance Techniques 105
4.6.5 Evaluation of Radius Growing Strategies 106
4.6.6 Evaluation of Adaptive Grid Sizing 107
4.6.7 Scalability 108
4.6.8 Sensitivity to Overlap 108
4.6.9 Memory Footprint 109
4.7 Related Work 111
4.8 Conclusion 112

Chapter 5: Integrating RT Acceleration into a Spatial Database 113

5.1 Introduction 113
5.2 Background 117
5.2.1 Apache SedonaDB 117
5.2.2 Ray-tracing and RT cores 120
5.2.3 Spatial Query by Ray Tracing 122
5.3 *RayBooster* Overview 123
5.4 GPU-friendly Storage Format 124
5.4.1 Storage Format 124
5.4.2 High-performance WKB Loader 126

5.5 Filtering with RT Cores . . . 127
5.5.1 Build the BVH . . . 128
5.5.2 Point Query . . . 129
5.5.3 Range Query . . . 129
5.6 Refinement with RT Cores . . . 130
5.6.1 Solving Combinatorial Complexity . . . 130
5.6.2 Calculating DE-9IM Matrix with RT cores . . . 131
5.6.3 Monolithic BVH by Z-Stacking . . . 133
5.6.4 CUDA-based Implementations as Fallback . . . 134
5.7 Memory Management . . . 135
5.7.1 Host Memory Management . . . 135
5.7.2 Device Memory Management . . . 135
5.8 Evaluation . . . 137
5.8.1 Settings . . . 137
5.8.2 SpatialBench . . . 138
5.8.3 Overall Performance . . . 140
5.8.4 Cost-effectiveness . . . 143
5.8.5 Running Time Analysis . . . 144
5.8.6 Effectiveness of Memory Management . . . 146
5.9 Related Work . . . 147
5.10 Conclusion . . . 149

Chapter 6: Conclusion . . . 150

Bibliography . . . 152

List of Tables

Table 1. Evaluated artifacts, queries, implementation notes, and measured operations. (H2D: Host to Device) 30

Table 2. Statistics of real-world datasets 31

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.) 32

Table 4. Polygon overlay execution time. The cell represents "Processing Time (Preprocessing Time) in seconds." ($*$: GPU-based) 37

Table 5. Summaries of the artifacts evaluated in the dissertation 64

Table 6. Real-world datasets collected from ArcGIS Hub and OpenStreetMap. 64

Table 7. Datasets and their statistics. The Gini index is used to describe the skewness of the data. 101

Table 8. Hardware specifications and pricing. 138

Table 9. Relational Spatial Joins from SedonaBench Query Workload. *: Refinement stage in Q9 fallbacks to CUDA. 140

Table 10. Average execution times of SpatialBench queries in seconds with standard deviations. “PGStrom Build+Exec.” represents the average SQL execution times plus the index build-up time on geometry columns used in the corresponding query. 141

List of Figures

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. 11

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. 11

Figure 3. Top: Switching between the RT Cores and SMs; Bottom: Execution flow of OptiX. 15

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 16

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). 19

Figure 6. Downcasting a 32-bit floating-point to 16-bit. 22

Figure 7. Low-precision AABB failed to enclose the line segment in high-precision, leading to geometric inconsistencies. 23

Figure 8. Conservative Representation: adding 1 to mantissa bits to compute a conservative right boundary 24

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. . . . 26

Figure 10. Summary of RayJoin's LSI speedups over baselines by considering total execution time. 33

Figure 11. Summary of RayJoin's PIP speedups over baselines by considering total execution time. 34

Figure 12. Execution time breakdown on $LKNA \bowtie PKNA$ datasets by varying merging threshold s 35

Figure 13. LSI scalability: R is fixed to 5M polygons, varying S from 1M to 5M 36

Figure 14. PIP scalability: R is fixed to 5M polygons, varying S from 1M to 5M 37

Figure 15. Summary of RayJoin's polygon overlay speedups over baselines by considering total execution time. 38

Figure 16. Two valid ray-AABB intersection cases: ray hits the AABB or ray origin is within the AABB 47

Figure 17. (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). 48

Figure 18. 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). 53

Figure 19. Three cases of (anti-)diagonal-rectangle intersections that imply the two rectangles intersect. 54

Figure 20. Load balancing with *Ray Multicast* technique 59

Figure 21. (a) Query time of 100K point queries; (b) Query time by varying the number of queries on OSMParks dataset 66

Figure 22. (a) Query time of 100K range queries with the *Contains* predicate; (b) Query time by varying the number of queries on OSMParks dataset 67

Figure 23. Query time of 10K *Range-Intersects* queries by varying selectivity. 67

Figure 24. 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 69

Figure 25. Index buildup costs, update rate by varying batch size, and query performance sensitivity to update ratios 70

Figure 26. 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. 72

Figure 27. The end-to-end execution time of 100K point in polygon queries 73

Figure 28. (a) Query Q searches its nearest neighbor within radius r; (b) Achieve the query with the inverse test that finds ray-AABB intersections followed by a point-in-sphere test . . 82

Figure 29. (a) using a uniform grid to organize spatially proximal points; (b) Create an MBR for points in a cell and iteratively expand the search radius. 84

Figure 30. Using HD estimators to calculate the lower and upper bounds of point-MBR distance. p_3^A is within the MBR, so the lower bound is zero 86

Figure 31. Execution process of *X-HD*. Stage 1: A grid is built over P^B to partition spatially proximal points into cells. Rays are cast from P^A search ray-AABB intersections with RT cores. `IsIntersection` shader sequentially processes each point in the cell to maintain the nearest neighbor. Stage 2: heavy cells are staged to an Intersection Queue, and they are processed with a dedicated CUDA kernel for better load balance. 87

Figure 32. Comparing Overall Performance of Hausdorff Distance Solutions. (a) Running time distribution of six method on BraTS datasets; (b-c) End-to-end execution time on geospatial and graphics datasets 102

Figure 33. (a) Running time analysis by enabling optimization options on *Dragon-Asian Dragon* datasets; (b) Breakdown workloads by counting the number of intersections and the number of points involved in distance computation 104

Figure 34. (a) Running time breakdown of *X-HD* and comparing the effectiveness of using load-balancing (LB) on geospatial datasets; (b) The same experimental settings but with graphics datasets. 105

Figure 35. Comparing three radius growing strategies. "Add Diagonal" increases the radius by the length of the cell diagonal; "Double Radius" doubles the search radius 106

Figure 36. Comparing the running time of a grid with the adaptive sizing technique to the optimal grid on all datasets. 107

Figure 37. (a) Evaluation of scalability by increasing points; (b) Sensitivity to overlap by separating two datasets 109

Figure 38. (a) Memory usage of geospatial datasets; (b) Memory usage of graphics datasets 110

Figure 39. Architecture of Apache SedonaDB 118

Figure 40. Performance comparison with SpatialBench. 120

Figure 41. (a) and (b) illustrate the definition of the ray and the cases of ray-AABB intersection test; (c) illustrates the point query with short ray; (d) range query with diagonal rays 121

Figure 42. Architecture of *RayBooster*. Both filtering and refinement are offloaded to RT cores. 123

Figure 43. (a-b) shows a `MultiPolygon` array and its in-memory storage format; (c) shows a unified `GeometryCollection` array that can contain any type of geometries 124

Figure 44. Filter stage of spatial join by ray-tracing 128

Figure 45. Demonstration of casting rays to find geometry element intersections and compute the DE-9IM model. 132

Figure 46. Execution workflow of RT-based refinement with Z-stacking technique. (a): Polygons on the indexed side; (b): Intersection Scene of the indexed polygons and queries; (c): Two

query points and a query polygon; (d): Refinement results by applying a mask matrix to the intersection matrices. 133

Figure 47. SQL execution breakdown (SF=1) of PGStrom Indexed. “Miscs” include sort/-group/aggregate operations. 142

Figure 48. Cost-effectiveness analysis on different platforms with SpatialBench (SF=10); (a) log scale, (b) linear scale 144

Figure 49. Comparing major time components of CPU and GPU joins (SF=1, Single Partition, g6e.2xlarge) 145

Figure 50. *RayBooster* execution time breakdown 146

Figure 51. Effectiveness of memory optimization strategies 147

Chapter 1: Introduction

According to a report from ESRI, eighty percent of big data is spatial [48]. Applications involving spatial data processing are ubiquitous, including location-based social networks (LBSNs), ride-sharing services, autonomous driving, and medical imaging. Spatial data has several defining properties. First, its volume is large due to data collected from numerous GPS-enabled devices and satellite imagery. Additionally, spatial data can change frequently, especially when sourced from mobile devices. Moreover, many spatial algorithms (e.g., overlay, nearest-neighbor search, spatial joins) are computationally intensive. At the same time, many spatial applications are time-sensitive, such as emergency management systems and interactive visualization tasks [244]. Therefore, timely processing is a fundamental requirement for spatial algorithms and systems.

A spatial index is essential for efficiently storing and retrieving spatial data by reducing the search space. In spatial data processing systems, range queries and k-nearest neighbor (kNN) searches are highly useful, enabling queries such as "retrieve all restaurants within 2 km of a user's current location" and "which are the three nearest gas stations to me?". Traversing a spatial index involves pointer chasing, which leads to numerous random memory accesses and frequent branch evaluations. As a result, searching a spatial index on GPUs, devices critical for high-performance computing, is generally inefficient due to their sensitivity to irregular memory access patterns and branch divergence.

Recent hardware advancements have introduced dedicated Ray Tracing (RT) cores into modern GPUs to accelerate ray-tracing rendering. RT cores are designed to accelerate ray-primitive intersection tests and the traversal of the Bounding Volume Hierarchy (BVH), which is essentially a spatial index. This new piece of hardware gives us the opportunity to accelerate spatial data processing by repurposing RT cores, considering the similarity of rendering workloads and spatial data processing tasks.

However, repurposing RT cores for spatial workloads is not trivial. First, RT cores only support finding ray-primitive intersections, while spatial data processing systems involve handling various geometries, such as points, line segments, rectangles, and polygons. Converting spatial processing tasks into ray-tracing problems is a fundamental challenge. In addition, the volume of spatial data is large. Mapping each geometry to a primitive, such as a triangle or an Axis-Aligned Bounding Box (AABB) to build the BVH can quickly exhaust the device memory, rendering it hard to handle large-scale spatial datasets. The precision of RT cores is also inherently limited, as they natively support only single-precision floating-point numbers (FP32), whereas geospatial data is often stored in double precision for high-precision (FP64) algorithms. Achieving high-precision computation with low-precision hardware is a challenge. Finally, spatial data scientists usually work with high-level programming languages to process the data. Therefore, it is challenging for them to write RT programs to accelerate their spatial data processing tasks. A general-purpose library with an easy interface is highly desirable to lower the barrier to RT programming, ideally accessible via Structured Query Language (SQL) within a database system. Together, these hurdles motivate a central question: how can we bridge the gap between low-level hardware primitives and production spatial databases while systematically resolving storage format mismatches, index construction overheads, and the combinatorial complexity of spatial functions?

Motivated by the demand for high-throughput spatial data processing and the massive performance potential of RT cores, we investigate repurposing RT cores for accelerating spatial processing, spanning from fundamental primitive operators to full-fledged spatial database engines. Specifically, this dissertation makes four primary contributions, detailed in the following subsections.

1.1 Fast and Precise Spatial Join on RT Cores

Spatial joins are fundamental operations in spatial databases and Geographic Information Systems (GIS), enabling users to combine datasets based on their spatial relationships. Among them, Line Segment Intersection (LSI) and Point-In-Polygon (PIP) tests are core primitive queries that serve complex operations such as polygon overlay and spatial filtering [131]. However, both operations are computationally demanding when executed on large-scale datasets with millions of line segments and vertices. While GPUs offer massive thread parallelism, traditional GPU implementations suffer from irregular memory access patterns and branch divergence during spatial tree traversals [29].

In Chapter 2, we present *RayJoin*, a real-time spatial join framework that repurposes RT cores to accelerate spatial joins. RayJoin reformulates LSI tests as ray-AABB intersections and transforms PIP queries into a closest-hit search problem. To tackle the architectural challenge where RT cores natively operate with FP32 numbers while geospatial applications require FP64 precision, RayJoin introduces conservative representation, a method that creates a low-precision AABB that is just large enough to enclose the high-precision geometries. This allows RayJoin to guarantee double-precision results with the limited precision provided by the hardware, outperforming state-of-the-art CPU and GPU spatial join frameworks by significant margins.

1.2 Designing a General-Purpose Spatial Indexing Library with Ray Tracing

While specialized operators like RayJoin demonstrate the computational potential of RT cores, general spatial applications require a spatial index to implement the classic filter-refinement scheme [165], where the index requires the ability to support diverse query primitives, dynamic index maintenance, and a flexible programming interface. Developing custom ray-tracing pipelines using low-level APIs (such as NVIDIA OptiX) imposes a steep learning curve on spatial application developers [171], who are typically accustomed to high-level spatial libraries.

In Chapter 3, we design and implement *LibRTS*, a general-purpose, high-performance spatial indexing library powered by RT cores. LibRTS reformulates primary spatial query types, such as point queries and range queries, into RT-friendly problems. To support dynamic updates, LibRTS employs a two-level acceleration structure that enables efficient incremental insertions, coordinate updates, and deletions through BVH refitting, bypassing the prohibitive cost of full index rebuilds. Furthermore, LibRTS introduces a "ray-multicast" strategy to mitigate load imbalance under the function-limited RT programming framework NVIDIA OptiX, delivering an easy interface that bridges the gap between hardware ray tracing and general spatial indexing.

1.3 RT for Spatial Functions (Case Study: Hausdorff Distance)

Beyond basic range queries and spatial joins, many advanced spatial analytics tasks rely on spatial metrics across large-scale point clouds. The Hausdorff Distance (HD) is a prominent example, widely used in CAD [31, 114, 128, 230, 246], image processing [15, 37, 82, 101, 147, 205], computational geometry [10, 81], and spatial databases, such as PostgreSQL [145, 175]. Computing HD according to its definition between two sets of sizes N and M has an intrinsic $O(N \cdot M)$ complexity, which quickly becomes prohibitively slow on massive datasets. Existing

GPU-based solutions either rely on memory-intensive tree structures or brute-force approaches with optimizations [204], suffering from high performance variability under skewed datasets.

In Chapter 4, we propose *X-HD*, an efficient and performance-robust algorithm that accelerates Hausdorff distance computation using RT cores. X-HD follows an iterative nearest-neighbor search accelerated by hardware BVH traversals [152]. To overcome the intersection overhead of constructing per-point primitives, X-HD adopts a uniform grid partitioning technique that clusters proximate points into Minimum Bounding Rectangles (MBRs), drastically curtailing the intersection overhead and GPU memory footprint. In addition, X-HD incorporates early-break pruning conditions based on distance estimators [155] and offloads compute-heavy cell evaluations to CUDA kernels to address the in-shader load-balancing issue.

1.4 End-to-End RT Acceleration for Spatial Databases

While individual operators and indexing libraries offer substantial speedups in isolation, integrating them into full-fledged, industrial spatial databases presents critical systemic challenges. First, real-world spatial engines (such as Apache SedonaDB) store heterogeneous, nested geometries in Well-known Binary (WKB) format, rather than continuous arrays, preventing random geometry accesses. Second, to execute queries over complex geometries more efficiently, we may need a micro-index for each geometry appearing in every row of a table. This requirement introduces severe micro-indexing overheads if distinct BVHs are constructed for every feature. Third, a spatial query engine must support diverse geometry types and predicates, requiring hundreds of CUDA kernels to handle them and incurring substantial maintenance overhead, which necessitates a unified method to evaluate diverse queries without implementing each individually.

In Chapter 5, we develop *RayBooster*, an end-to-end RT-core acceleration solution natively integrated into Apache SedonaDB. RayBooster resolves data format divergence via a columnar,

GPU-friendly storage format based on hierarchical offset arrays, coupled with a multi-threaded parallel WKB loader on the CPU. To eliminate micro-indexing bottlenecks, RayBooster introduces "Z-Stacking," a technique that repurposes the unused third dimension in 2D geospatial workloads to encode feature IDs across distinct Z-planes, allowing millions of geometries to be indexed within a single monolithic BVH. Furthermore, RayBooster introduces a *RelateEngine* that computes the Dimensionally Extended 9-Intersection Model (DE-9IM) directly using ray casting to provide a unified spatial relation descriptor.

The remainder of this dissertation is organized according to these contributions. Chapter 2 presents the design for fast and precise spatial joins on RT cores with *RayJoin*. Chapter 3 describes the generic spatial indexing library *LibRTS*. Chapter 4 illustrates the *X-HD* Hausdorff distance computation algorithm. Chapter 5 details *RayBooster*, an end-to-end RT-accelerated spatial database engine for Apache SedonaDB. Finally, Chapter 6 concludes the dissertation and outlines future research directions.

Chapter 2: Fast and Precise Spatial Join on RT Cores

2.1 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." [27, 120, 150]. 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 [3, 46, 131, 167]. This requirement is particularly acute in domains like interactive urban planning, emergency management, and risk assessment [192, 228, 244].

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 [56, 154, 216] but suffers from limited parallel processing capabilities [11, 12, 138]. Rasterization-based methods offer parallelism but require intensive preprocessing [66, 208, 244]. Learned spatial indexes, though efficient in query performance and storage, struggle with high training costs and accuracy concerns [74, 106, 168, 177, 194, 251]. Consequently, GIS researchers and practitioners often resort to conventional indexing techniques, such as grids and trees, which offer numerous advantages [3, 12, 46, 71, 198, 200, 206, 226, 231, 232, 235, 238, 239, 245].

However, the conventional spatial indexes mentioned above have performance limitations from achieving real-time processing [197]. 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 [141, 144, 220].

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 [58, 113, 241, 250]. In fact, our evaluation shows that none of the existing spatial join methods satisfy the real-time processing requirement (§2.5.7), 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 [156, 157]. 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 chapter, 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 (§2.3.1).

- 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 (§2.3.2).

- We have proposed an algorithm that greatly reduces the BVH construction time and memory cost while keeping high query performance (§2.3.3).

- We have comprehensively compared and analyzed RayJoin with many existing spatial join solutions by a consistent benchmark with synthetic and real-world datasets (§2.5), 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]

2.2 Background

2.2.1 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,

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

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).

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

[2] To understand dataset R, considering the provinces/states (the three simple polygons) share the common boundaries forms into a country (the whole polygon).

Performing spatial join queries on a large volume of complex geometries is challenging due to the high computational cost [1]. Exhaustive search compares all pairs of geometries, resulting in $O(|R| \cdot |S|)$ time complexity, which is impractical for large datasets [137]. Thus, spatial indexes are necessary to improve query performance by reducing search space.

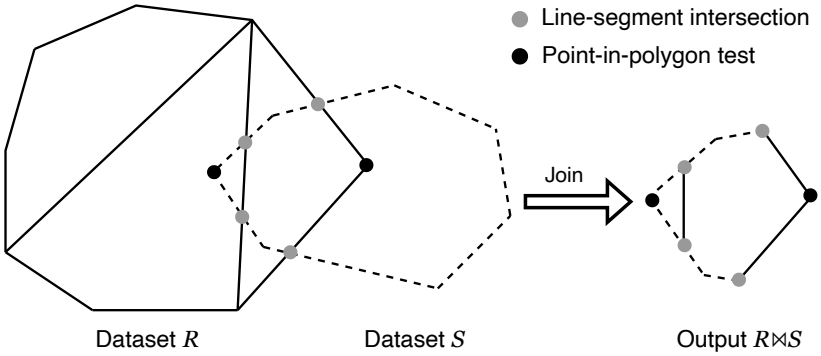


Figure 1. Example: polygon overlay analysis through spatial join. Dataset R contains the union of multiple simple polygons with common boundaries. Dataset S has a simple polygon. The output is the overlapped regions from two datasets.

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

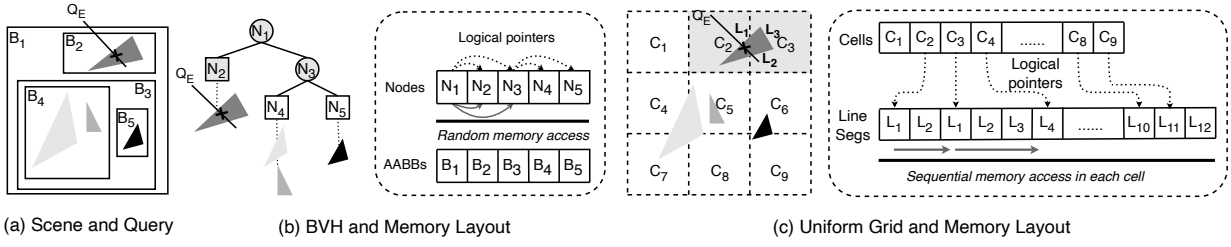


Figure 2. (a): given a group of geometric objects and a query Q_E, finding which geometry intersects with the query. (b) and (c): demonstrate the acceleration of the query with two spatial indexes and their memory layouts and access patterns.

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 [75].

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 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 [96].

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

[3]AABB is a synonym of MBR in this dissertation.

[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.

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.

2.2.2 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 §2.2.1. 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 [29]. Several ray-tracing interfaces are available for the ray-tracing hardware, including NVIDIA OptiX, Vulkan, and DirectX Raytracing. This chapter focuses on OptiX as it is interoperable with CUDA.

OptiX supports the "Multiple Instruction, Multiple Data" (MIMD) subset of CUDA [159]. 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 [258]. 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

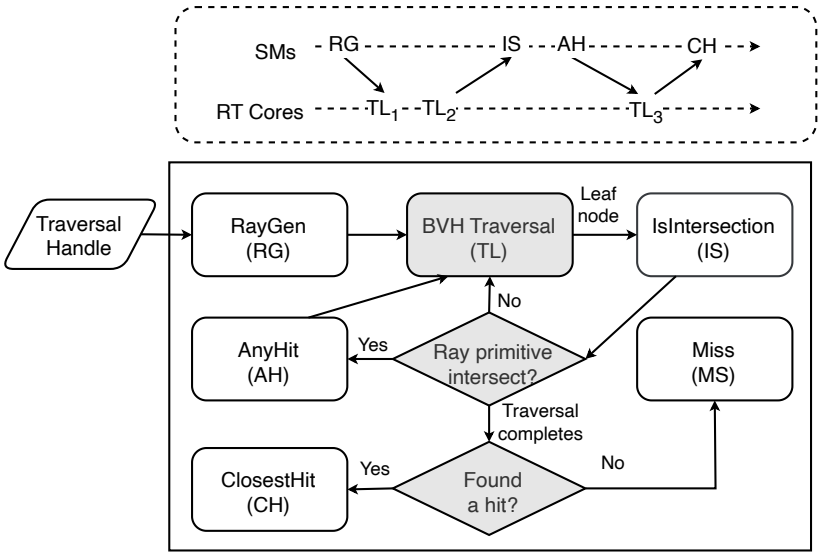


Figure 3. Top: Switching between the RT Cores and SMs; Bottom: Execution flow of OptiX.

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 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.

2.3 Spatial Join Operations on RT Cores

2.3.1 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):

$$\mathbb{R}(t) = O + t \cdot \vec{d}, \text{where } t \geq 0 \tag{2.1}$$

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 [131], which can then be formulated as two RT-friendly problems.

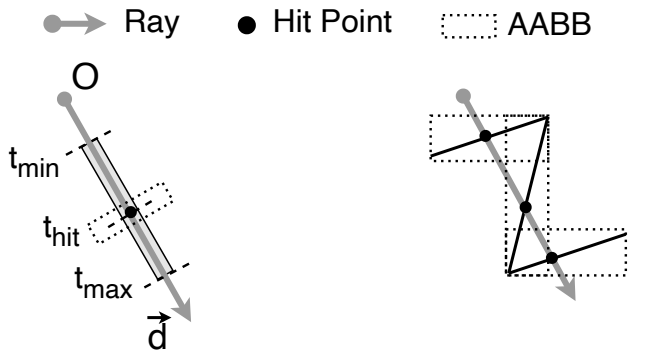


Figure 4. Formulating LSI as a ray tracing problem. By simulating the line segment with a ray, intersections are identified by ray-AABB intersection tests

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.

$$O = p_1, \vec{d} = p_2 - p_1, t_{min} = 0, t_{max} = 1 \tag{2.2}$$

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

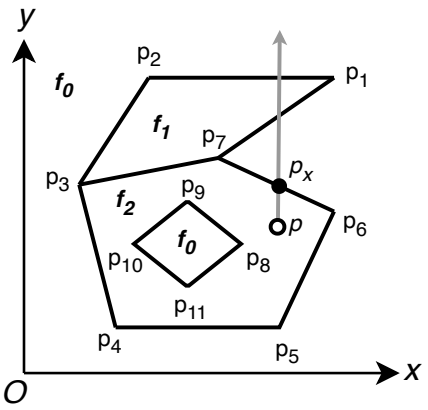
the intersection into result set Q. At line 27, we invoke the *optixIgnoreIntersection* function to ask OptiX to continue searching.

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         p_1, p_2 = GetEndpoints(s)                // Get two endpoints from line segment s
 5         O = p_1                                                          // Ray origin
 6         d⃗ = p_2 − p_1                                                  // Ray direction
 7         // Cast a ray carrying line segment ID of s
 8         optixTrace(h, O, d⃗, 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     d⃗ = 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) / d⃗.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
```

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 [196]. 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 [184]. This operation is the building block of complex spatial queries like polygon overlay [131].



(a) PIP query with ray-tracing

Query Point Ray Closest-Hit

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	$p_8, p_9, p_{10}, p_{11}, p_8$

(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).

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

[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.

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 [60, 131]. 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 §2.2. 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.

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`
  $t_{min}, t_{max} = \{0, \infty\}$
  for each p **in** S **do**
    $O = p$ `// Query point as origin`
    $\vec{d}.xyz = \{0, 1, 0\}$ `// Towards upward of y-axis`
    $optixTrace(h, O, \vec{d}, t_{min}, t_{max}, payloads(id_p))$ `// 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
  $p_x = R[id_l] \bowtie_{intersects} l$ `// Intersection point of the line segment from R and` l
  if $p_x \neq \phi$ **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

2.3.2 High Arithmetic Precision Beyond the Hardware Support

Ray-tracing frameworks like OptiX typically use FP32 for optimal performance [99]. 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 [95], 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 [86, 134]. 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.

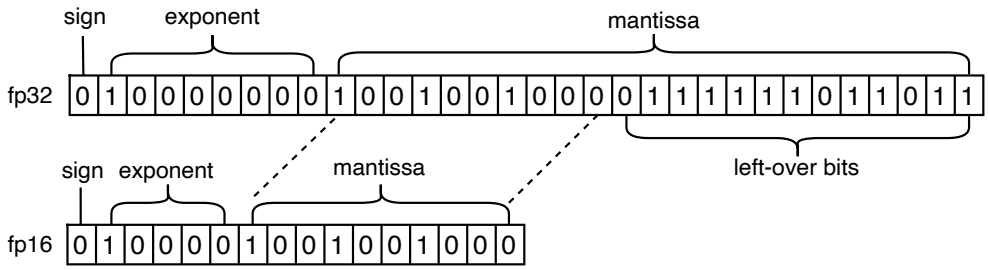


Figure 6. Downcasting a 32-bit floating-point to 16-bit.

To tackle the precision challenges, it is crucial to understand the source of imprecision. Figure 6 represents floating-point numbers in binary format[6], consisting of three components: sign, exponent, and mantissa [83]. 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" [83], 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

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

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.

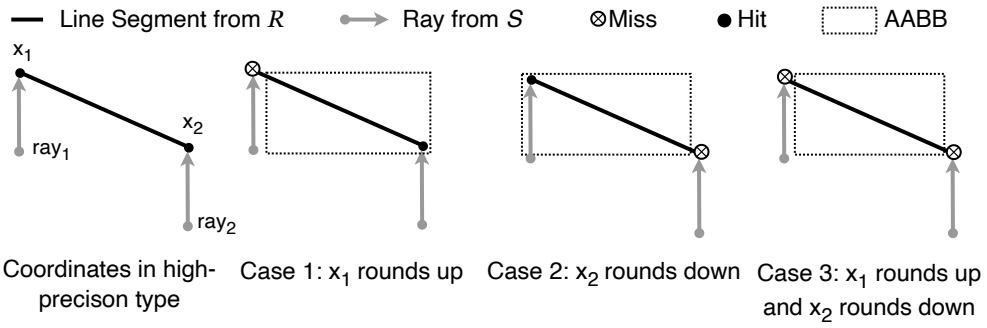


Figure 7. Low-precision AABB failed to enclose the line segment in high-precision, leading to geometric inconsistencies.

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.

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

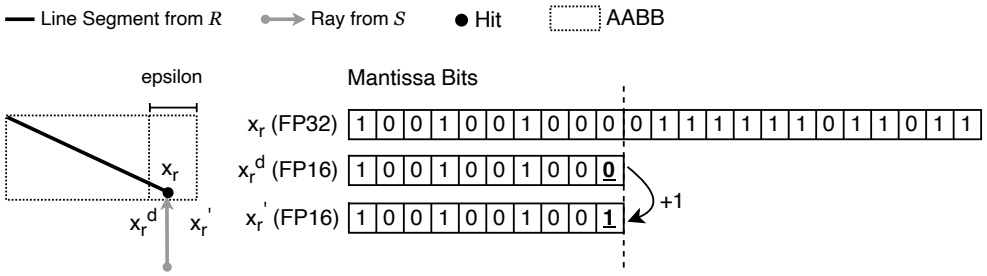


Figure 8. Conservative Representation: adding 1 to mantissa bits to compute a conservative right boundary

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).

2.3.3 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 (§2.3.1). 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 [258]. One may impulsively use the Ramer–Douglas–Peucker algorithm to simplify polylines and then create an AABB for each simplified polyline [180]. 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 $1 \leq i < j \leq n$). 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 $\frac{|B_i \cup B_j|}{max(|B_i|, |B_j|)} \leq s$, where s is the merging threshold. We will discuss how the parameter s impacts performance and how to find an optimal s in §2.5.5. This approach is inspired by R-Tree's heuristic insertion algorithm [71], which effectively curtails the occurrence of false positives by limiting AABB's dead space.

[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.

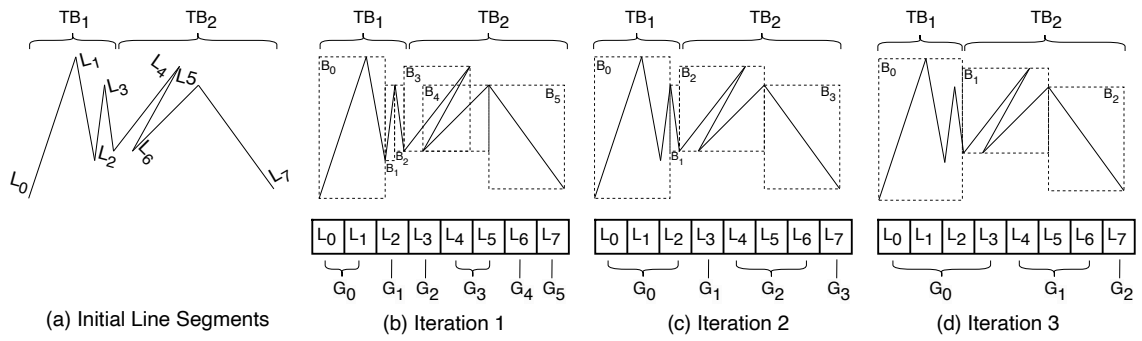


Figure 9. Execution process of Adaptive Grouping. Initially, a thread block (TB) processes a fixed number of line segments. Spatially close line segments are iteratively put together.

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. 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.

2.4 RayJoin Deployment In Practice

2.4.1 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) [45]. 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 [152].

2.4.2 RT Cores Acceleration Enabled Applications

We also discovered a subset of *ST* functions in geodatabases that may benefit from RT Cores [85]. 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 chapter (and is addressed in Chapter 5).

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 §2.5.7.

2.5 Evaluation

2.5.1 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 [96]. cuSpatial is a GPU-accelerated geospatial analytical platform from NVIDIA [158]. RasterJoin is a raster-based solution on the GPU for spatial aggregation [244]. PSSL is a state-of-the-art plane-sweep-based algorithm for LSI on GPUs [61]. To the best of our knowledge, GLIN is the only learned spatial index that supports indexing geometries with extends [219]. 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 [131]. Kinetica and PostGIS are geospatial databases [84]. RasterIntervals uses a raster-based method to accelerate polygon overlay query [66].

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 [47] and global maps, such as *Lakes* and *Parks* from OpenStreetMap (OSM) [46]. 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 [97]. The polygon size and segment length are set to 0.001 and 10, respectively. The Gaussian distribution is set to $\mu = 0.5, \sigma = 0.1$.

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

Artifact	Evaluated Queries	Implementation Notes	Preprocessing	Measured Processing
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

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 [131]. RayJoin only

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

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 §2.5.5.

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.

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)**

2.5.2 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 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.

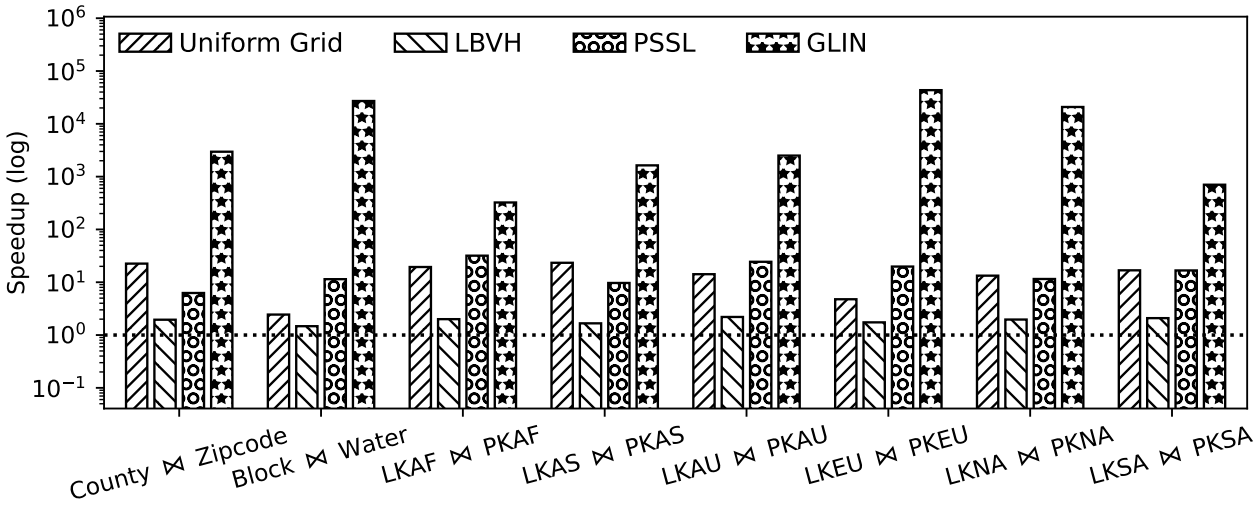


Figure 10. Summary of RayJoin's LSI speedups over baselines by considering total execution time.

2.5.3 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.

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

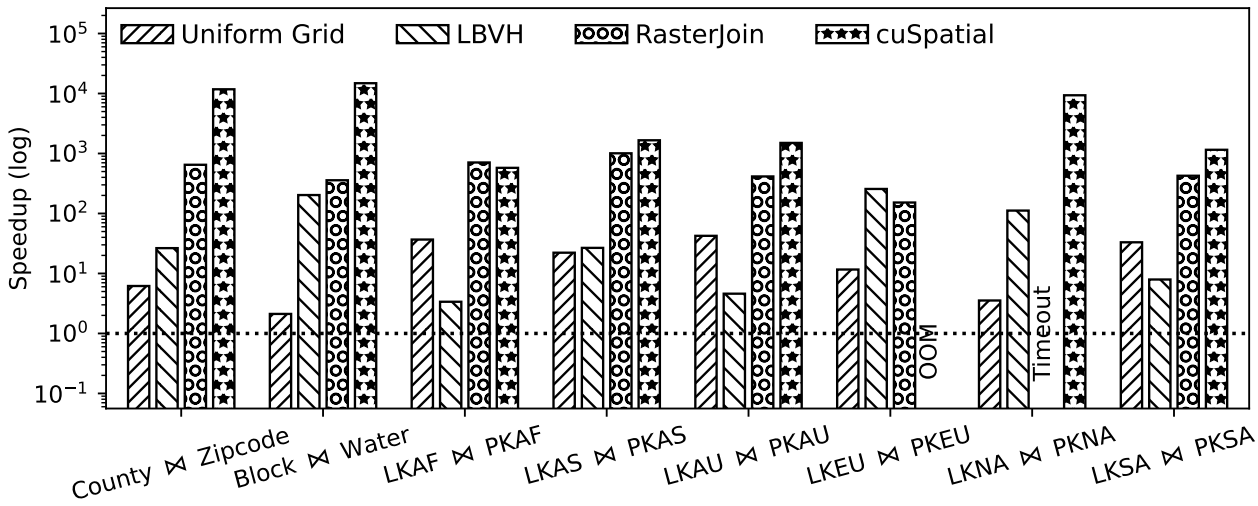


Figure 11. Summary of RayJoin's PIP speedups over baselines by considering total execution time.

(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.

2.5.4 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 completely the same answers compared to the FP64 program, and we have verified the correctness of all datasets. These experiments confirm the analysis in §2.3.2, and our solution eliminates all the false negative cases due to the precision loss.

2.5.5 Effectiveness of Adaptive Grouping and Parameter Tuning

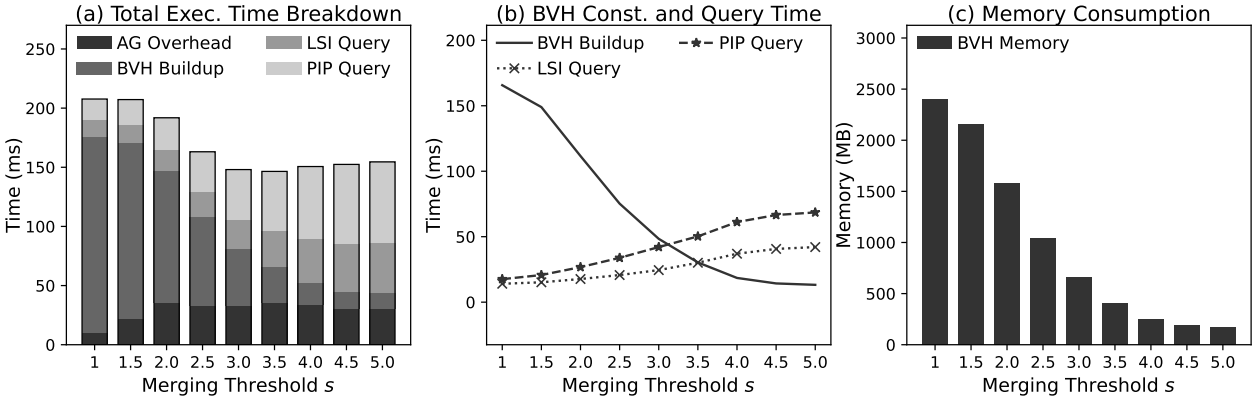


Figure 12. Execution time breakdown on $LKNA \bowtie PKNA$ datasets by varying merging threshold s

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.

2.5.6 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.

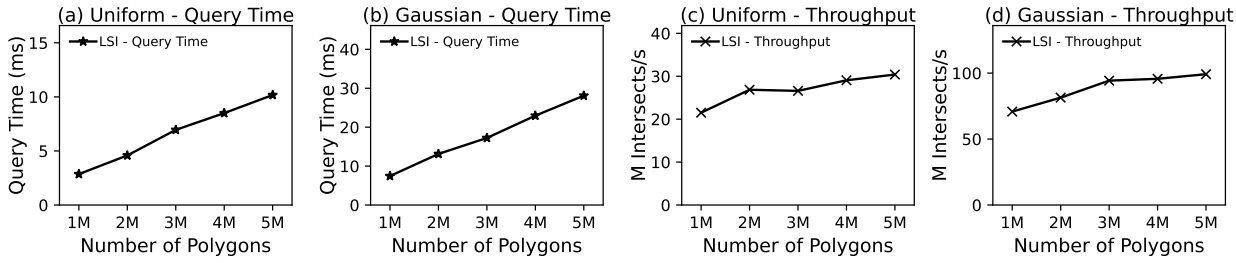


Figure 13. LSI scalability: R is fixed to 5M polygons, varying S from 1M to 5M

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.

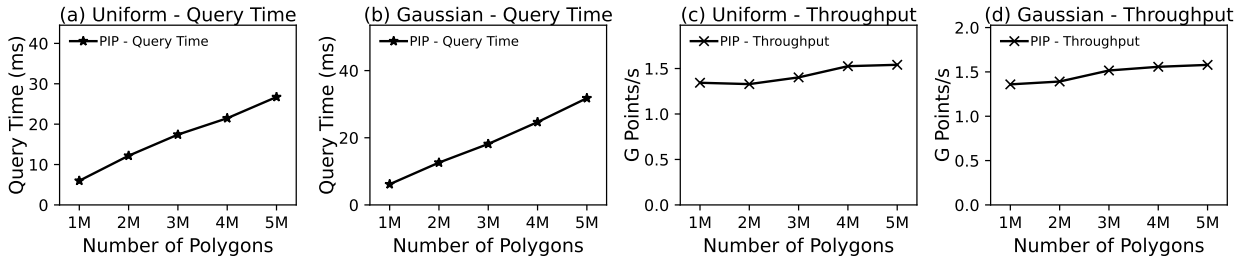


Figure 14. PIP scalability: R is fixed to 5M polygons, varying S from 1M to 5M

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 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.

2.5.7 Polygon Overlay

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)**

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 (§2.2.1). The overlay processing is extremely time-consuming due to the complexity of the overlay algorithm and the irregularity of the execution.

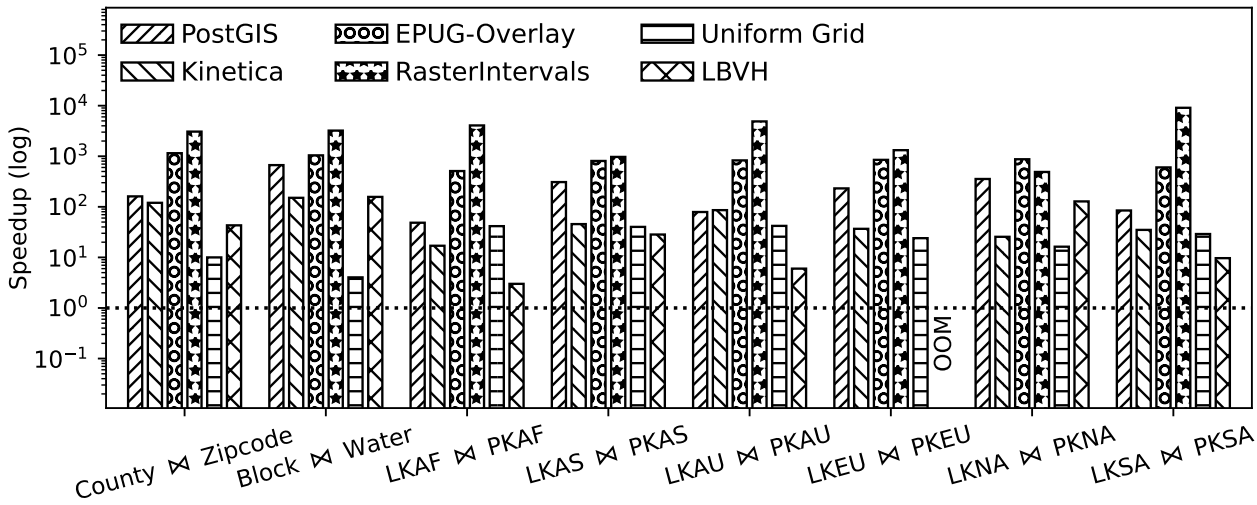


Figure 15. Summary of RayJoin's polygon overlay speedups over baselines by considering total execution time.

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* ⋈ *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.

2.6 Related Work

Spatial Indexing on Advanced Hardware. Geometric Performance Primitives (GPP) is a robust polygon overlay library on GPUs with a uniform grid [12]. Liu et al. proposed a filtering technique to accelerate spatial join with fine-grained tiles [125, 126]. Rayhan et al. designed an R-tree that leverages SIMD instructions [182]. SwiftSpatial is a spatial join accelerator on FPGAs [88]. OCTANE is a framework using octree on the GPU to accelerate molecular dynamics simulations [212]. 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 [244]. RasterIntervals is the state-of-the-art polygon overlay method with the approximation technique to reduce the refinement cost [66]. PixelBox employs a rasterization-based method to compute the Jaccard Similarity [221]. IDEAL is a vector-raster hybrid method for PIP and point-to-polygon distance calculation [208]. 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 [138] 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 [61]. 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 [190]. RTNN adopted spatial reordering and query partitioning optimizations for K nearest neighbor (KNN). TrueKNN supports arbitrary radius KNN-search on RT Cores [152]. JUNO is the first work supporting high-dimensional KNN search on RT Cores [127]. A method presented in [132] supports non-Euclidean distance measures for KNN. A method proposed in [108] 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 [218]. Literature [90] explores using RT Cores to serve as a B-Tree. RTScan leverages RT Cores to accelerate index scans [130]. Accelerating Range Minimum Queries with RT Cores is discussed in [140]. The most recent work related to this chapter, Pierce, investigates spatial joins over 3D meshes on RT cores [73]. None of these works address the spatial join problem that this chapter targets, which presents unique challenges such as problem formulation, precision preservation, and performance optimization.

2.7 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.

Chapter 3: A Spatial Indexing Library by Ray Tracing

3.1 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.[8] 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 [29].

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 [64, 78, 108, 127, 130, 133, 148, 152, 190, 218, 258].

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

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 [72]. 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 [78, 130, 140], incurring additional computational and storage overhead.

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 [78, 79].

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 (§3.3.1, §3.3.2, and §3.3.3).

2. We are the first to identify and address the load balancing challenge that arises when repurposing RT cores (§3.3.4).

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 (§3.5).

4. LibRTS is an open-source library offering a user-friendly interface that simplifies the integration of RT cores into spatial data processing applications (§3.4).

3.2 Background

3.2.1 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 [23, 25, 59, 255], Quadtrees [57, 249], and grids [2, 4] 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 [20, 71, 129, 176, 231, 232, 237] and BVHs [38, 70, 139, 186]. 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 [13, 14]. In contrast, grid-based indexes map geometries to cells with linear memory space, improving memory efficiency but struggling with skewed data [12, 197, 199]. 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 [42, 106, 121, 123,

177]. However, most learned spatial indexes are designed for disk-based predictions and merely identify the disk pages containing the queried results [117, 177]. Furthermore, training a learned index is time-consuming [124], and few such indexes support geometries with extents [5, 219].

3.2.2 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 [227]. 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 (3.1):

$$\mathbb{R}(t) = O + t \cdot \vec{d}, \text{where } t \geq 0 \tag{3.1}$$

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 $t_{min} \leq t_{hit} \leq t_{max}$.

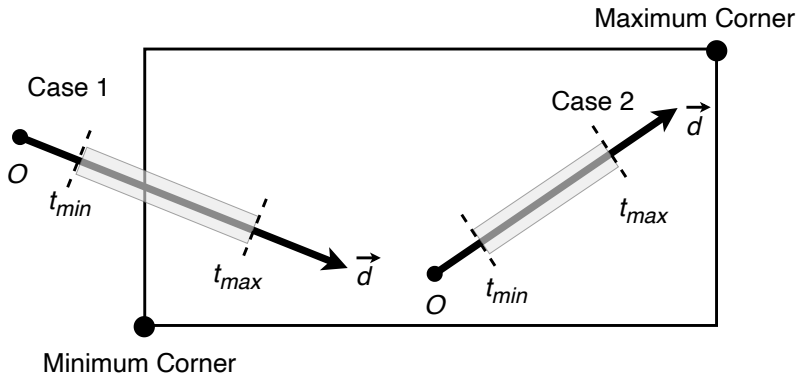
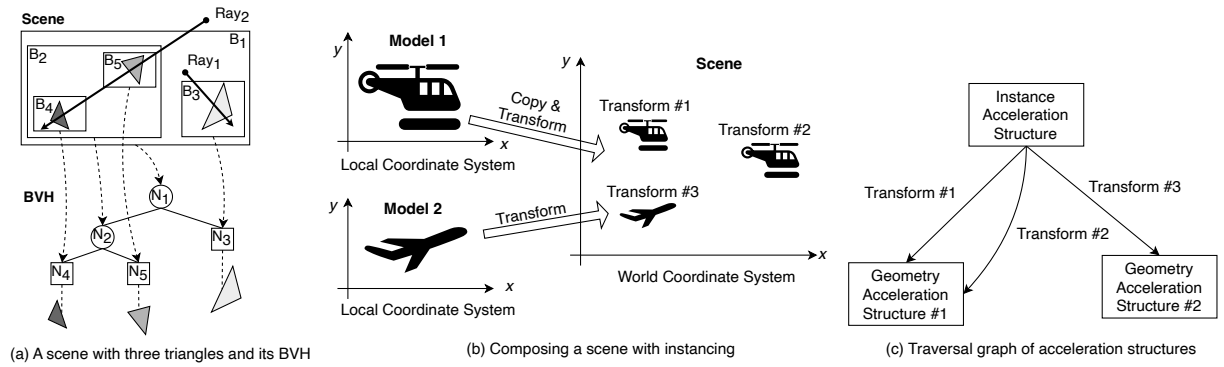


Figure 16. Two valid ray-AABB intersection cases: ray hits the AABB or ray origin is within the AABB

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 16 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 16, 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.



(a) A scene with three triangles and its BVH (b) Composing a scene with instancing (c) Traversal graph of acceleration structures

Figure 17. (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).

3.2.3 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 17(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 17(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 17(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.

3.2.4 Programming Model for RT Cores

RT cores, introduced with the Turing architecture [29], 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 [156].

Additionally, the ray-primitive performance has doubled with each new generation of GPUs, offering "free" performance gains for RT-enabled applications [156, 157].

NVIDIA OptiX [159] 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 [159]. 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 [78]. 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[9] 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.

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

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.

3.3 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.

3.3.1 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 18 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 §3.2.2, 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 (§3.2.2). 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 18, 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.

3.3.2 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).

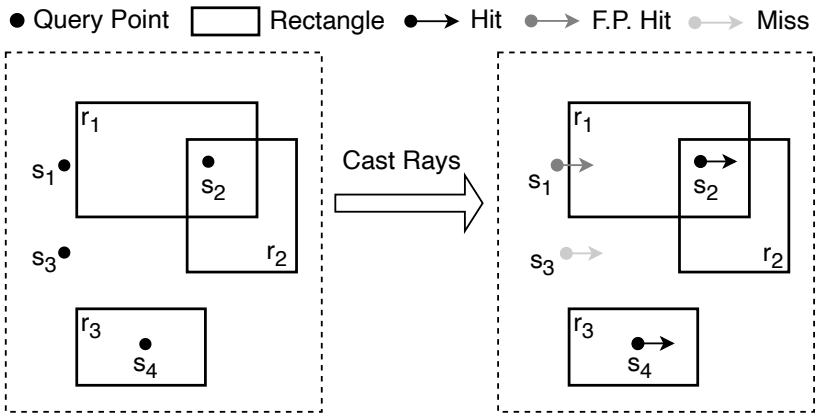


Figure 18. Casting rays from the query points results in three intersection cases: a hit (black); a False Positive (F.P.) hit (grey), and a miss (light grey).

Definition 2 (Rectangle-Rectangle Containment)**.** Given two rectangles r_1 and r_2, predicate $Contains(r_1, r_2)$ is true if $r_1.x_{min} \leq r_2.x_{min} < r_2.x_{max} \leq r_1.x_{max} \wedge r_1.y_{min} \leq r_2.y_{min} < r_2.y_{max} \leq r_1.y_{max}$, 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 (§3.3.1) 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.

3.3.3 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 $r_1.x_{min} \leq r_2.x_{max} \wedge r_1.x_{max} \geq r_2.x_{min} \wedge r_1.y_{min} \leq r_2.y_{max} \wedge r_1.y_{max} \geq r_2.y_{min}$, otherwise it is false.

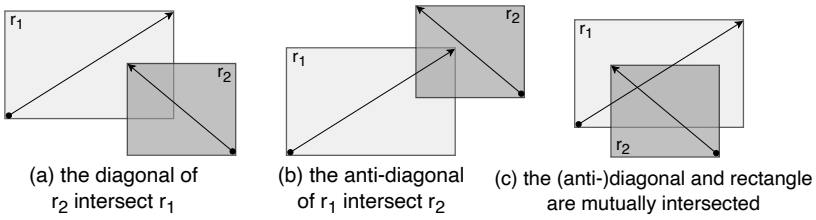


Figure 19. Three cases of (anti-)diagonal-rectangle intersections that imply the two rectangles intersect.

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 19(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. As previously noted, a (anti-)diagonal is essentially a line 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} (§3.2.2). For a diagonal defined by two endpoints p_1 and p_2, the ray parameters can be derived as follows [64].

$$\text{Ray origin } O = p_1$$

$$\text{Ray direction } \vec{d} = p_2 - p_1 \tag{3.2}$$

$$t_{min} = 0, \text{and } t_{max} = 1$$

We can verify the correctness of these parameters by substituting them into Equation (3.1). 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 (§3.2.2, 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 19(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 3 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" [98, 173]. 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 19(c)). *Backward Casting* is implemented in a similar way, but it omits the deduplication process.

Algorithm 3: 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         d⃗ = (s.x_min, s.y_max) − O
 9         optixTrace(h, O, d⃗, 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 ∪ (id_r, id_s)
21         end if
22     end if
23 end procedure
```

3.3.4 Ray Multicast for Load Balancing

Recall that OptiX adopts a single-ray programming model (§3.2.4), 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 [22, 142, 144, 220, 223]. Unfortunately, these techniques are not applicable to RT cores because OptiX does not support thread block or warp synchronization (§3.2.4). 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.

Figure 20 demonstrates how *Ray Multicast* works. Assuming we have three AABBs and $k = 3$. Figure 20(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 $x \in (2,3)$ [10]. In Figure 20(b), three rays are cast with x-coordinates offset by 0, 1, and 2. Each ray therefore intersects only one AABB.

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

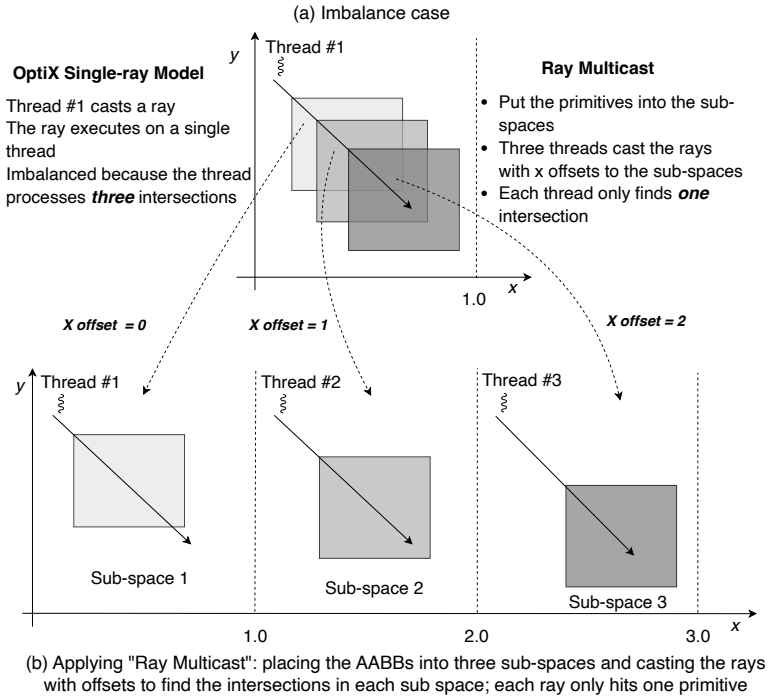


Figure 20. Load balancing with *Ray Multicast* technique

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 (3.3)). The goal is to find a k to minimize cost C.

$$C = (1-w) \cdot C_R + w \cdot C_I \tag{3.3}$$

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 (3.4).

$$C_R = |R| \cdot k \cdot log(|N|) \tag{3.4}$$

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 (3.5).

$$C_I = \frac{|N| \cdot |R| \cdot s}{k} \tag{3.5}$$

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 §3.6.5.

3.4 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 4 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

Algorithm 4: 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);
}
```

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.

3.5 Update Spatial Index

3.5.1 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 (§3.2.3). 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, the function `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.

3.5.2 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 [78]. 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.

3.6 Evaluation

3.6.1 Evaluation Setup

Baselines. We endeavor to include the most competitive baselines on both CPUs and GPUs, as listed in Table 5. Building on the work by Lawson et al. [111], who evaluated 20 open-source spatial libraries and concluded that no single solution is optimal for every query—while highlighting CGAL [33] and Boost [26] for their overall performance. Therefore, we include these libraries. We also evaluate ParGeo, a computational geometry library optimized for multicore environments [225]. 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 [219] 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 [96], to illustrate that LibRTS indeed benefits from hardware acceleration. We further assess a real-world application, the Point in Polygon (PIP) test. cuSpatial [158] 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 [64] (presented in Chapter 2), 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 cores to ensure

Table 5. Summaries of the artifacts evaluated in the dissertation

Artifact	Index Type	Query Type	Platform
Boost [26]	R-Tree	Point, Range	CPU
CGAL [33]	KD-Tree	Point	CPU
ParGeo [225]	KD-Tree	Point	CPU
GLIN [219]	Learned Index	Range	CPU
LBVH [96]	Linear BVH	Point, Range	GPU
cuSpatial [158]	Octree	Point	GPU
RayJoin [64]	BVH on RT cores	PIP	GPU
LibRTS	BVH on RT cores	Point, Range, PIP	GPU

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

Datasets. Table 6 shows the real-world geospatial datasets collected from ArcGIS Hub [47] and OpenStreetMap [46]. 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 [97].

Table 6. Real-world datasets collected from ArcGIS Hub and OpenStreetMap.

Dataset	Polygons	Description
USCounty	12.2 K	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 [111, 149, 219]. 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 §3.6.9. 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.

3.6.2 Point Query

From Figure 21(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 21(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

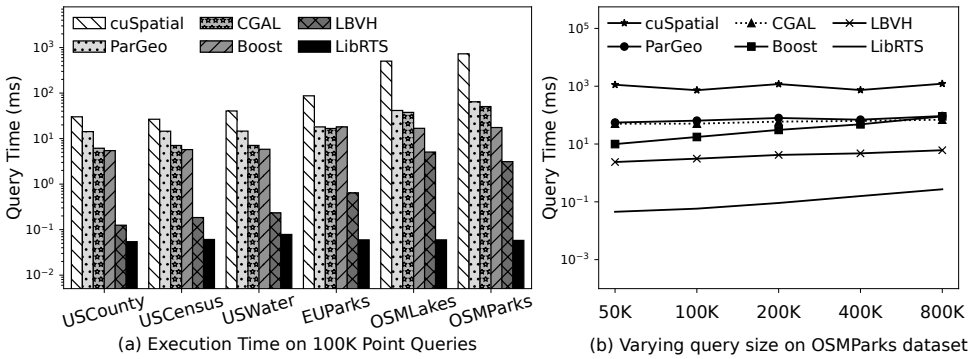


Figure 21. (a) Query time of 100K point queries; (b) Query time by varying the number of queries on OSMParks dataset

gap between these two indexing strategies narrows. Nonetheless, LibRTS consistently outperforms all baselines, even for a very large number of queries.

3.6.3 Range Query with Contains Predicate

Figure 22(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 22(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

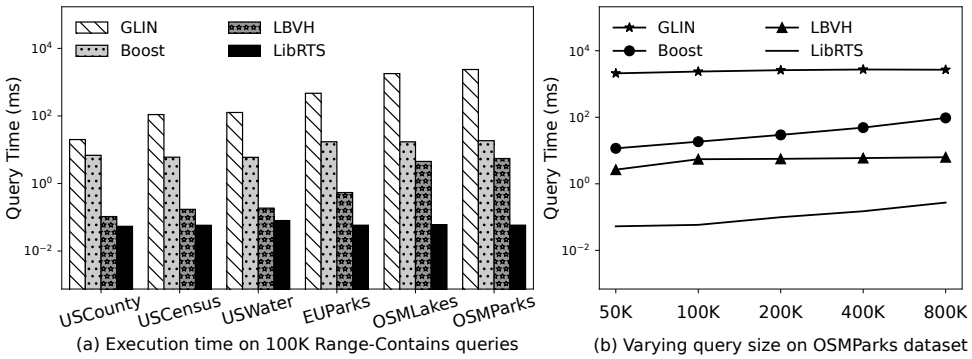


Figure 22. (a) Query time of 100K range queries with the *Contains* predicate; (b) Query time by varying the number of queries on OSMParks dataset

more queries, LibRTS consistently outperforms all the baselines, demonstrating its scalability in handling larger query volumes.

3.6.4 Range Query with Intersects Predicate

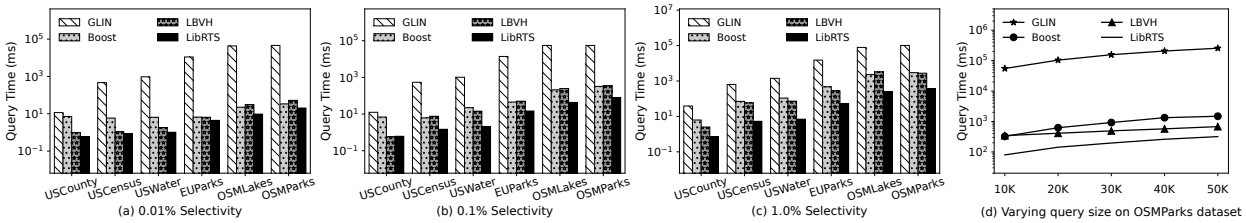


Figure 23. Query time of 10K *Range-Intersects* queries by varying selectivity.

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 23(a)-(c) shows the query times for different selectivity levels. As shown in Figure 23(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 [72], 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 23(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 23(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.

3.6.5 Effectiveness of Load Balancing

In §3.3.4, we introduce the *Ray Multicast* technique for improved load balancing, which requires the parameter k to find the lowest ray casting cost. Figure 24(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 24(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.

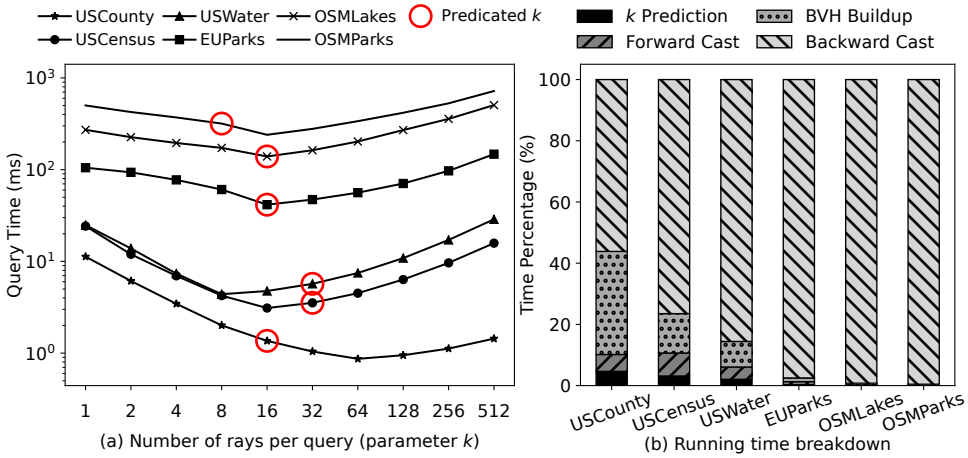


Figure 24. Demonstration of the effectiveness of *Ray Multicast* on *Range-Intersects* 50K queries and a selectivity 0.1%. (a) The changing of query times by varying the number of rays per query, with red circles indicating the predicted number of rays by our method; (b) Query time breakdown

Figure 24(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 §3.3.4, 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.

3.6.6 Update Performance

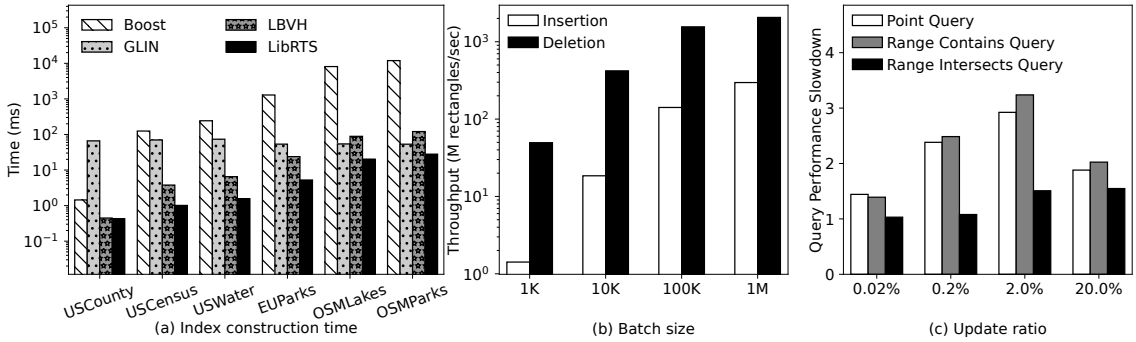


Figure 25. Index buildup costs, update rate by varying batch size, and query performance sensitivity to update ratios

Figure 25(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 25(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.

3.6.7 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 25(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.

3.6.8 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 [97]. Figure 26(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

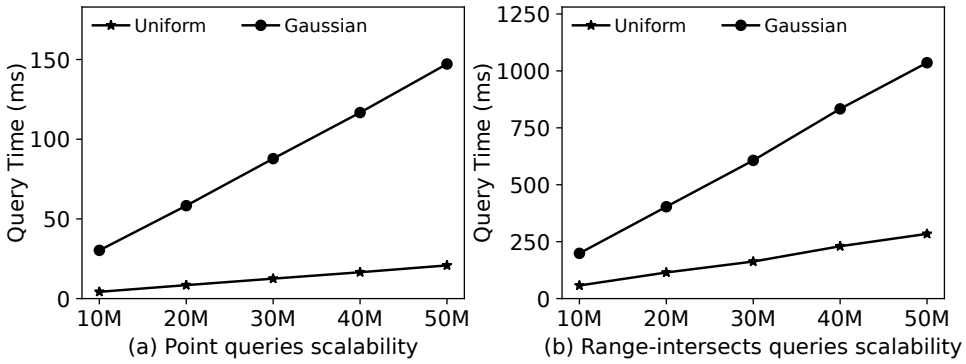


Figure 26. Scalability results. The number of queries is fixed to 10K for all the queries. The query times are reported by varying the number of rectangles indexed by LibRTS.

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 26(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.

3.6.9 Real-world Application: PIP

Figure 27 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

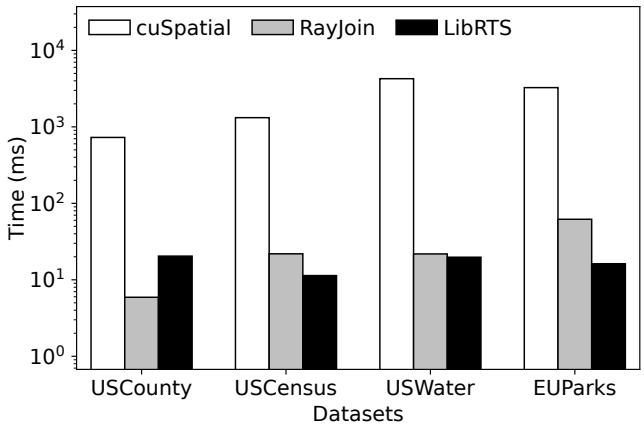


Figure 27. The end-to-end execution time of 100K point in polygon queries

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.

3.7 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 [53]. RTNN extended the radius search to k Nearest Neighbor (kNN) search with query scheduling optimizations [258]. TrueKNN further supports kNN search with an arbitrary radius [152]. JUNO offers an approximate nearest neighbor search in high-dimensional space [127]. Arkade supports kNN search in non-Euclidean space [133]. These works focus on neighbor search and do not support range queries.

Database Workloads RTIndeX turns RT cores into a B-tree [78]. RTScan leverages RT cores to accelerate index scans [130]. cgRX generalizes RTIndeX to achieve updateability and more performant queries [79]. RT-DBSCAN is an RT-based DBSCAN software [152]. Meneses et al. explore range minimum queries [140] 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 [218] and unstructured mesh point location [148]. Laass formulated the PIP query as a ray tracing problem by transforming polygons into a 3D representation [108]. RayJoin adapts a different formulation for PIP and supports the Line Segment Intersection query [64], 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. [190]. Maul et al. proposed utilizing RT cores to accelerate the generation of synthetic X-ray attenuation images [135]. Zhao et al. suggested leveraging RT cores for particle-based simulations [252]. 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. [72] 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. [19] 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.

3.8 Conclusion

In this chapter, 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.

Chapter 4: A Fast Algorithm for Hausdorff Distance by Ray Tracing

4.1 Introduction

The Hausdorff Distance (HD) is a spatial metric used to measure the dissimilarity between two sets of points. It quantifies the maximum distance from a point in one set to the nearest point in the other set, effectively capturing the greatest discrepancy between the two. HD is an important tool in various fields, including CAD [31, 114, 128, 230, 246], image processing [15, 37, 82, 101, 147, 205], computational geometry [10, 81], and spatial databases, such as PostgreSQL [145, 175].

The volume of spatial data has expanded rapidly in recent years, driven by the widespread use of high-resolution sensors, such as LiDAR, GPS-enabled devices, satellite imagery, and advanced medical imaging technologies. For example, datasets collected from LiDAR sensors to train self-driving cars can contain billions of points [21]. At the same time, spatial workloads are often time-sensitive [244], making it imperative to develop super fast HD algorithms supported by an effective hardware acceleration.

The HD computation is highly intensive and time-consuming. Existing approaches to solving HD typically follow its definition, where each point in one set finds its nearest point in the other set. Subsequently, the HD can be computed by taking the maximum of the distances between the points and their closest counterparts. The above operation is a special case of k-nearest neighbor (k-NN) search where $k = 1$. Using a tree-like data structure, e.g., KD-tree, to solve the problem on the GPU

can be very slow due to its inherent recursive nature, which causes excessive branch divergence and random memory access [92, 258].

Recent research has shown that using Ray-Tracing (RT) cores is beneficial for solving the k-NN problem [133, 152, 258]. RT cores are designed to accelerate photorealistic rendering, which are ubiquitous in consumer-grade GPUs, data centers, and supercomputing environments [6, 29]. To utilize these specialized units, the k-NN search must be reformulated as a ray-tracing problem, such as searching ray-Axis-Aligned Bounding-Box (AABB) intersections by traversing a Bounding Volume Hierarchy (BVH) tree, and the tree traversal is conducted on the RT cores.

However, naively invoking an RT-based k-NN library to solve HD yields poor performance. In existing works, such as *RTNN* [258] and *RT-KNNS Unbound* [152], each point is mapped to an individual AABB to exploit the RT-accelerated BVH search. This strategy leads to significant BVH traversal and intersection test overhead due to the large volume of points. In addition, some crucial domain-specific optimizations, such as Early Break (EB) [204] and HD estimators [155], cannot be applied in these libraries to reduce redundant computation, as they are viable only for the HD algorithm. Finally, the existing RT programming model, e.g., NVIDIA OptiX [161], lacks a load-balancing mechanism in the RT shader, leading to poor hardware resource utilization.

To address the above mentioned issues, we propose the following design principles for a high-performance HD algorithm on RT cores.

1. We aim to reduce the intensity of the BVH traversal and intersections. Our measurement shows that handling an intersection is about 5$\times$ more expensive than calculating the distance between two points.

2. The intensity of distance computation should be significantly reduced because the value of HD only comes from a single pair of points, implicating many non-contributing points can be culled.

3. Load-balancing should be carefully considered because RT cores adopt a single-ray programming model, and poor load-balancing will significantly underutilize the GPU [161].

In this chapter, we introduce *X-HD*, an RT-accelerated HD algorithm designed for high-performance processing of large-scale datasets. *X-HD* has three core techniques. First, we use a uniform grid to group spatially proximal points into a cell, which is inexpensive to build. Subsequently, each non-empty cell is extended by a search radius to create an AABB, which serves as the input for building the BVH. This step not only reduces the BVH size for faster construction but also results in fewer intersections and better spatial locality [44]. Reducing non-contributing points is our second core technique. Using the cell boundaries, we employ HD estimators to calculate the lower and upper bounds of HD, which helps us to prune non-contributing points within the cell. *X-HD* also incorporates the EB optimization to cull individual points that cannot increase the value of HD. Finally, while introducing a grid offers benefits, it also brings new issues, such as imbalanced workloads due to skewed data distribution. For better load balancing, *X-HD* selectively offloads the distance computation from the RT shader to a dedicated CUDA kernel to improve load balance.

Our experiments show that *X-HD* outperforms the state-of-the-art medical imaging framework *ITK* by 5.3× on average with MRI images. Compared to a GPU-optimized EB algorithm, *X-HD* achieves a speedup of up to 6.4× on large-scale geospatial datasets.

The contributions of this chapter are as follows:

1. We designed and implemented a highly optimized HD algorithm tailored for execution on RT cores.

2. We proposed using grid and HD estimators to reduce the intensity of BVH traversal and distance computation. The latter is beyond HD - it can also be incorporated into RT-based k-NN solutions [169].

3. We investigated the load balance issues inherent in RT shaders and proposed a general solution to improve load balance. This technique can be used by any RT-accelerated algorithms facing load-balancing challenges.

4. We have comprehensively compared and analyzed *X-HD* with five baselines using diverse real-world datasets.

5. Putting all the design and implementation efforts together, we created *X-HD* as an open-source software for the research and application communities [11].

4.2 Background

4.2.1 Hausdorff Distance

Given two sets of points P^A and P^B, where $P^A = \{p_1^a, p_2^a, \ldots, p_m^a\}$ and $P^B = \{p_1^b, p_2^b, \ldots, p_n^b\}$ in a metric space with distance function $\|\cdot\|$. The directed Hausdorff distance from P^A to P^B is:

$$\overrightarrow{H}(P^A, P^B) = \max_{p^a \in P^A} \{ \min_{p^b \in P^B} \{\|p^a, p^b\|\}\} \tag{4.1}$$

Note that the directed Hausdorff distance is not symmetric, e.g., $\overrightarrow{H}(P^A, P^B) \neq \overrightarrow{H}(P^B, P^A)$. The undirected Hausdorff distance can be trivially calculated by taking the maximum of the two directed Hausdorff distances, which yields a symmetric measure. While the directed HD finds its applications in graphics, such as finding the best partial distance between two images [82], the undirected HD can be applied to evaluate the quality of a medical image segmentation algorithm [93] and trajectory

[11] Source code: https://github.com/pwrliang/X-HD

analysis [170]. In this chapter, we focus on the directed Hausdorff distance for point sets in 2-D/3-D Euclidean space as it has wide applications [62, 82, 155, 201].

4.2.2 HD Algorithms

To compute HD, we can directly follow its definition in Equation 4.1 with a double-loop scheme. The outer loop iterates over every point $p^a \in P^A$, while the inner loop finds the minimum distance, *cmin*, from p^a to the points in P^B. After each inner loop, the global maximum distance, *cmax*, is updated with $cmax = max(cmax, cmin)$ to maintain the maximum of the minimum distances found. The time complexity[12] of this algorithm is $O(|P^A| \cdot |P^B|)$, which is not scalable and time-consuming for large datasets.

Early Break There is an opportunity to optimize the double loop by skipping points that cannot contribute to the Hausdorff distance, achieved by breaking the inner loop earlier [204]. As shown in Algorithm 5, the distance of a pair of points, *dist*, is checked against *cmax*. If *dist* is not greater than *cmax*, we can safely exit the inner loop. This pruning logic can be interpreted as follows: if the current minimum distance (*cmin*) found for point p^a is already less than *cmax*, then continuing the inner loop to yield a lower distance still can not be greater than *cmax* and thus cannot further increase the overall Hausdorff distance *cmax*.

This algorithm can be very efficiently implemented on the GPU. For the outer loop, we dispatch a thread block to take point p^A, while the inner loop can be unrolled with the thread block. In Line 8, we use a block-level reduction to aggregate the minimum distance from each thread and vote to break the loop with a thread block synchronization.

Nearest-Neighbor Search Since the inner loop in Algorithm 5 essentially computes the distance from p^a to its nearest neighbor point, we can avoid visiting all the points in P^B by formulating the inner loop as a k-NN problem, where $k = 1$.

[12] We consider the spatial dimension as a small constant factor, e.g., 2 or 3.

Algorithm 5: Hausdorff Distance with Early Break

```
Input  : Point Set P^A
Input  : Point Set P^B
Output : Hausdorff Distance
Shuffle(P^A) // Randomize points to avoid similar distances in successive
    iterations [204]
Shuffle(P^B)
cmax ← 0
for each p^a in P^A do
    cmin ← ∞                    // Min distance discovered from p^a to P^B
    for each p^b in P^B do
        dist ← ||p^a, p^b||
        cmin ← min(cmin, dist)
        if dist ≤ cmax then
            break // p^A cannot further increase cmax
        end if
    end for
    cmax ← max(cmax, cmin)
end for
return cmax
```

We first build a spatial index, e.g., a KD-tree, over the point set P^B. Then, we iterate over the point set P^A, performing a nearest neighbor search for each point $p^a \in P^A$ using the spatial index. Next, we calculate the distance between p^a and its nearest neighbor to update *cmax*. With the index, the average time complexity is $O(|P^A| \cdot \log|P^B|)$.

Although introducing a spatial index will reduce the work amount, resulting fewer distance computations, tree-like data structures are not efficient for the GPU due to the random memory accesses and branch divergence [258]. Our experiments show that using a KD-tree is almost always slower than the EB solution (§4.6.2). In this chapter, we will show how we turn this hardware inefficient algorithm into a GPU-friendly solution with RT cores.

4.2.3 RT-accelerated Nearest Neighbor Search

The basic idea of RT-based NN search originates from an idea known as the "inverse test" [53, 258]. For instance, a query point Q seeks to identify its nearest neighbor within a specified

radius r (Fig. 28(a)). This problem can be equivalently reformulated as an inverse test. As shown in Fig. 28(b), the problem is reversed to ask: "Which spheres of radius r, centered at data points, contain the query point Q?"

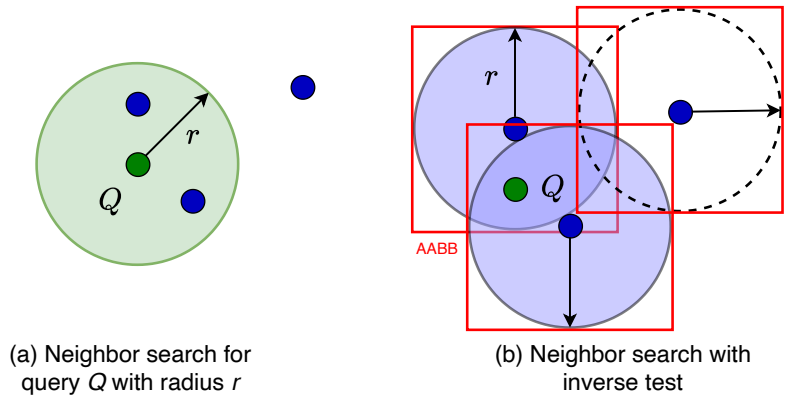


Figure 28. (a) Query Q searches its nearest neighbor within radius r; (b) Achieve the query with the inverse test that finds ray-AABB intersections followed by a point-in-sphere test

By utilizing this inverse test approach, we employ an AABB to enclose each sphere and cast rays from the query points to find potential intersections. The AABBs are defined by expanding the points by a radius r. A query point finds its nearest neighbor if its ray hits an AABB and the distance to the AABB's center is no more than r[13]. Otherwise, the search continues, and the point is reprocessed with an increased r (§4.5).

Searching for these intersections can be achieved by traversing the BVH built using the AABBs, and the traversal is automatically accelerated by RT cores. This design is therefore significantly faster than software-emulated tree search solutions [258]. The existing RT-accelerated NN search algorithms are based on NVIDIA's OptiX framework because of its support for BVH traversal acceleration. To write an RT program, developers must write RT shaders — essentially a piece

[13]In cases of multiple AABB intersections, the shortest distance is recorded.

of CUDA code, but executed from the perspective of a single thread. When the RT cores report a potential intersection, an RT shader called `IsIntersection` is invoked, allowing the developer to handle the intersection event. For the NN search case, the `IsIntersection` shader calculates the distance between the ray origin and the center of the AABB being hit and maintains the minimum distance discovered so far. Once the shader completes, the execution flow transfers from the CUDA cores back to the RT cores to continue the search process.

4.3 Design Principles

Directly using the RT-based NN algorithms to solve HD results in very poor performance due to the cost of building a large BVH, redundant computation, and load imbalance. We discuss these equally important issues and the ideas to solve them in this section.

4.3.1 Reducing BVH Size

The existing RT-based *k*-NN algorithms, such as *RTNN* [258] and *RT-KNNS Unbound* [152], adopt a one-to-one mapping between a point and an AABB (§4.2.3). This simple mapping scheme results in a high BVH construction cost and reduces the efficiency of filtering intersections. Existing research has shown that the building time is linearly correlated with the number of AABBs [258]. When the number of points is high, BVH construction becomes a performance bottleneck. A large BVH also results in more intersections. When casting a ray to search the BVH, whenever a leaf node is visited, `IsIntersection` will be invoked to notify the users to handle potential ray-AABB intersections. The more AABBs a BVH has, the more AABBs a ray will possibly hit.

To reduce the BVH size, we should allow an AABB to enclose multiple points. Only spatially proximal points should be grouped to reduce the dead space[14] of the AABB, leading to fewer false-positive intersections. If an AABB is hit, we can iterate over the points in the cell and narrow

[14]The dead space refers to the area in the AABB that has no geometries [52]

down the nearest point. There are many data structures that serve the above purposes. For example, a quad-tree or KD-tree can be used to partition the space, and spatially proximal points will be naturally placed in a tree node. However, we have tried them and found that these complex data structures are very expensive to build on the GPU due to their recursive nature.

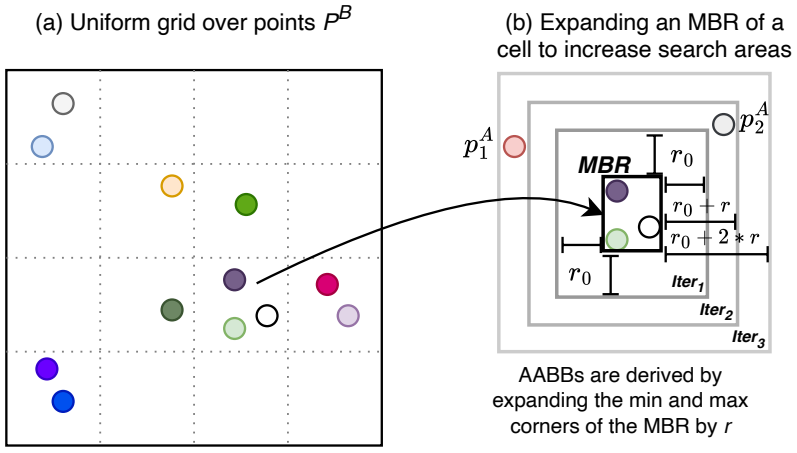


Figure 29. (a) using a uniform grid to organize spatially proximal points; (b) Create an MBR for points in a cell and iteratively expand the search radius.

A uniform grid [60] is a simple and widely used data structure [100, 224] that partitions the space into many identical, square-sized cells, as shown in Fig. 29(a). With a predefined grid, a point can be mapped into its cell with $O(1)$ time complexity, and the storage for the grid can be a simple array. Using the grid, proximal points are automatically placed into the same cell. Introducing a grid also causes other issues, such as load imbalance and determining an appropriate grid resolution.

4.3.2 Reducing Redundant Computation

4.3.2.1 EB and NN are not Mutually Exclusive

Recall that EB is essentially a pruning technique that skips a non-contributing point in P^A. While the NN method adopts a spatial index to prune the points in P^B that are very far from P^A. In short, these two methods prune non-contributing points from P^A and P^B, respectively.

These two methods can be used together. We could adopt the NN-based algorithm as the framework while incorporating the EB technique simultaneously. In the process of the NN search, if the current minimum distance (*cmin*) found for point p^A is lower than the global maximum distance (*cmax*), it is safe to exit from the search process early. Compared to the EB algorithm, which iterates through every point in P^B, using a search tree will reduce the accesses to points P^B. By including the idea of EB, we can early-stop the search process to reduce redundant computation. Since this technique is only viable for the HD algorithm, directly invoking a standard k-NN search library will miss opportunity of applying this optimization, but *X-HD* seizes the opportunity to reduce unnecessary computation.

4.3.2.2 Utilizing HD Estimators

Nutanong et al. proposed an early HD algorithm that uses the branch-and-bound search principle to avoid visiting some tree branches in the index [155]. Unfortunately, their method is inherently sequential and thus cannot be used in massive parallel environments. However, the utilization of Hausdorff distance estimators in their paper motivated us to further prune redundant computation.

The HD estimators provide lower and upper bounds on point-MBR distance [155], denoted by $MinDist(p^A, M^B)$ and $MaxDist(p^A, M^B)$, respectively. As illustrated in Fig. 30, *MinDist* is the distance from p^A to the closest face of M^B, while *MaxDist* is the distance to the farthest corner. The estimators allow us to eliminate redundant computations. We can cull all the points within M^B if

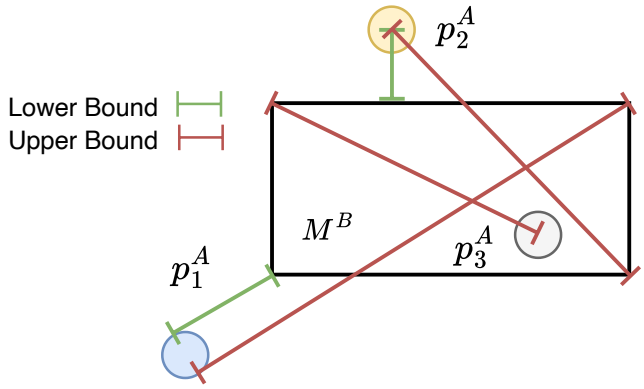


Figure 30. Using HD estimators to calculate the lower and upper bounds of point-MBR distance. p_3^A is within the MBR, so the lower bound is zero

they cannot contribute to the HD. For example, if the upper bound of the HD is not greater than *cmax*, we can safely skip all the points enclosed by M^B because they cannot increase *cmax*. We will elaborate on how to use the HD bounds to reduce computation workloads in §4.4.

4.3.3 Avoiding Heavy Workload in Shaders

OptiX adopts a single-ray programming model, where a single thread casts a ray and executes `IsIntersection` shader in serial. A thread will take a long execution time if it involves a large workload, therefore, underutilizing the GPU resources. Unfortunately, OptiX does not allow the use of warp/block synchronization intrinsics, which makes it impossible to use various load-balancing techniques, such as Cooperative Thread Array (CTA) scheduling [122, 144, 166]

Given a uniform grid, due to the inherent nature of real-world datasets, the points are often not evenly distributed across the grid cells. When handling the points within a cell, a single thread will serially visit them to calculate distances. While this may not be a problem for cells containing very few points, it can be disastrous for cells with tens of thousands of points, leading to a long running time. Due to the limitations of the RT programming model, the only way to address this issue is to

offload the distance computation in RT shaders to general CUDA kernels, which allows us to use various synchronization primitives to rebalance the workloads.

4.4 The X-HD Algorithm

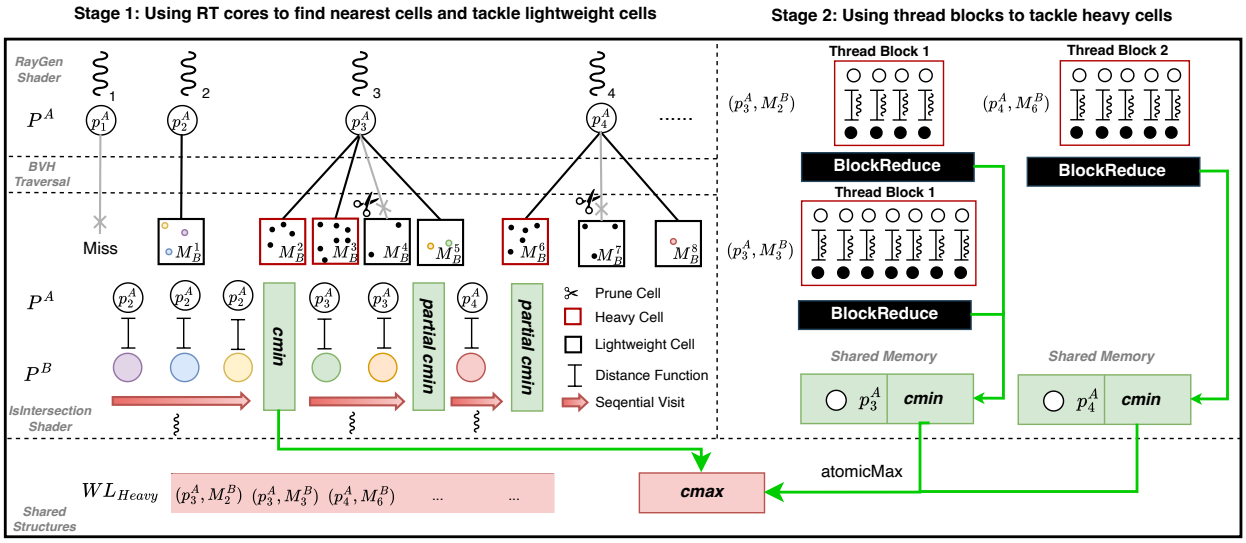


Figure 31. Execution process of *X-HD*. Stage 1: A grid is built over P^B to partition spatially proximal points into cells. Rays are cast from P^A search ray-AABB intersections with RT cores. `IsIntersection` shader sequentially processes each point in the cell to maintain the nearest neighbor. Stage 2: heavy cells are staged to an Intersection Queue, and they are processed with a dedicated CUDA kernel for better load balance.

4.4.1 Algorithm Overview

X-HD first uses a uniform grid as the first-level index to partition points P^B, and spatially proximal points will be grouped into their cells. The second-level index, a BVH tree, is built over non-empty cells to accelerate the search for cells potentially containing the nearest neighbor. Rays are cast from P^A to find the cells hit by these rays (Fig. 31, Stage 1). Each `IsIntersection` shader

invocation indicates the ray is close to a cell that potentially contains the nearest neighbor. Within the shader, we estimate the lower and upper bounds of the point-MBR distance, which helps prune non-contributing points, reducing redundant computation. Upon the execution of `IsIntersection`, we visit every point in the cell to pinpoint the nearest neighbor. Since some cells in real-world datasets may contain a large volume of points, sequentially visiting every point in such a cell is prohibitively slow. To mitigate the impact of imbalanced workloads, we do not immediately process these cells. Instead, we record the intersections in a worklist (WL_{Heavy}) and later process them in a dedicated CUDA kernel (Fig. 31, Stage 2), where thread blocks cooperatively process the cells for better load balance.

4.4.2 Iterative Nearest Neighbor Search

X-HD is an iterative algorithm that searches for the nearest neighbor by gradually growing the search radius r to reduce unnecessary intersections [152]. An iteration consists of building a BVH, casting rays, handling intersections, and adjusting the search radius, and Fig. 31 shows the first three steps. The iterations will repeat until every point $p^A \in P^A$ finds its nearest neighbor.

Build BVH. Initially, points P^B are inserted into a uniform grid G to place them into different cells, as shown in Fig. 29(a). We calculate the MBR for each non-empty cell, and these MBRs are then expanded by a search radius r, forming AABBs as inputs to build a BVH. Using the MBR instead of the cell boundary provides a tighter boundary, which helps to reduce false-positive intersections and more accurate HD bounds.

Cast Rays. Each point p^A casts a short ray to search for ray-AABB intersections, where the ray origin is p^A, and the direction can be arbitrary. Searching for intersections is implemented by traversing the BVH, which is accelerated by RT cores. This process is therefore very fast.

Handle Intersections. The `IsIntersection` shader is triggered when a ray hits an AABB. In the shader, we retrieve the AABB ID to locate its corresponding cell in the grid and calculate the distances between p^A and the points in the cell, where the minimum distance, *cmin*, is also maintained. After visiting all points in the cell, we use *cmin* to update the global maximum *cmax* with $atomicMax$. Some cells may not contribute to the HD, we prune them in this process.

Adjust Search Range. Some points in P^A may not hit any AABBs at all due to an insufficient search radius. Consequently, we increase r for a wider search range. Fig. 29(b) visualizes this process with a cell containing three points from P^B and two query points, p_1^A and p_2^A, from P^A. In *Iteration 1*, an AABB is created by extending the MBR by an initial search radius r_0, but it does not cover either p_1^A or p_2^A. The search radius is then increased to $r_0 + r_d$ to include more points from P^A, where r_d is how long the search radius grows in each iteration. In *Iteration 2*, p_2^A intersects the AABB, but p_1^A does not. The radius is further increased to $r_0 + 2 * r_d$. Finally, in *Iteration 3*, p_1^A intersects the AABB, and the search stops.

Algorithm 6 summarizes the above three steps. Line 1-5 is for building the BVH. Two worklists, WL_{In} and WL_{Miss}, are used to maintain the points casting rays and the points that cannot find intersections when the current iteration ends, respectively. In Line 11, rays are cast from the points in WL_{In} to find intersections, which is accelerated by RT cores. Subsequently, `IsIntersection` shaders will be invoked whenever an intersection is found, in which the distances between p^A and points in the cell will be calculated (Algorithm 7). If the ray from a point does not hit any AABB, the point will be moved to WL_{Miss}, which will be searched again with an expanded search radius in the next iteration. Cells with too many points will not be immediately processed in `IsIntersection` shader. They are instead addressed with a dedicated kernel for better load balance (Algorithm 9). Therefore, we cannot use *cmin* to update *cmax* because the computation is incomplete. We stage the incomplete *cmin* values to *partial_cmin*, allowing the CUDA kernel `TackleHeavyCells` to

resume the computation (§4.4.4). In Line 24, we increase the radius for a larger search area, which is explained in §4.5. In Line 26-27, we update the BVH with a new radius using the BVH refit function, which is much faster than rebuilding the BVH [78]. The above process repeats until WL_{In} is empty, and *cmax* will be the result of HD. Since the algorithm is a variant of *NN* search, its complexity remains $O(M \cdot N)$. However, the adoption of various pruning techniques makes *X-HD* performant in practice.

Algorithm 6: *X-HD* Outline

```
   Input  : Point Set A, Point Set B
   Output : Hausdorff Distance
 1 Insert points P^B into a uniform grid G
 2 MBRs ← GetNonEmptyCells(G)
 3 r_0 ← GetInitRadius()
 4 AABBs ← {M^B | (M^B_min − r_0, M^B_max + r_0), M^B ∈ MBRs}
 5 BVH ← BuildBVH(AABBs)
 6 WL_In = P^A, WL_Miss ← ∅, cmax ← 0
 7 while WL_In ≠ ∅ do
 8     partial_cmin ← ∅, WL_Heavy ← ∅
 9     for each p^A in WL_IN do
10         cmin ← ∞
11         optixTrace(p^A, cmin)      // Cast a ray at p^A; cmin is a reference to be updated in
             IsIntersection
12         Handle Intersections in IsIntersection shader     // automatically invoked by RT cores
             (see Algorithm 7)
13         if the ray does not hit any cell then
14             WL_Miss ← WL_Miss ∪ p^A
15         else if the ray does not hit any heavy cell then
16             if cmin ≠ ∞ then
17                 atomicMax(&cmax, cmin) // Update only if the ray is not terminated
18             end
19         else
20             partial_cmin ← partial_cmin ∪ cmin
21         end
22     end for
23     cmax ← TackleHeavyCells(partial_cmin, cmax, WL_Heavy)
24     r ← GetNextRadius(r, WL_In, WL_Miss)                          // New search radius
25     WL_In ← WL_Miss, WL_Miss ← ∅
26     AABBs ← {M^B | (M^B_min − r, M^B_max + r), M^B ∈ MBRs}
27     BVH ← RefitBVH(AABBs)                                  // Update the BVH with new AABBs
28 end
29 return cmax
```

Algorithm 7: IsIntersection RT Shader (called by RT cores)

Input : Point p^A, Heavy cells WL_{Heavy}, global maximum $cmax$
Output : Updated $cmin$, WL_{Heavy}

$cmin \leftarrow getPayload$ // Get previous $cmin$ from payload register
$i \leftarrow GetPrimitiveIndex()$ // index of the intersected AABB
$M^B \leftarrow MBRs[i]$
// WL_{Heavy} is updated by $CalculateMinDist$ in Algorithm 8
$min_dist = CalculateMinDist(p^A, M^B, cmax, cmin, WL_{Heavy})$
if $min_dist = \infty$ **then**
  TerminateRay() // Stop BVH traversal as p^A is culled
else
  $cmin = min(cmin, min_dist)$
  Write back $cmin$ to the payload
end

4.4.3 Culling Non-contributing Points

Not every point will contribute to the final Hausdorff distance. Given a point $p^A \in P^A$ and a subset of points from P^B that an MBR M^B tightly encloses, we prune either p^A or all the points in M^B if they cannot increase the value of $cmax$ by considering these cases:

- Case 1: $MaxDist(p^A, M^B) \leq cmax$. If the upper bound of the distance from point p^A to any point in M^B is not greater than $cmax$, then these points cannot increase $cmax$ and will be discarded.
- Case 2: $MinDist(p^A, M^B) \geq cmin$. If the lower bound of the distance from p^A to M^B is not less than the current minimum distance ($cmin$) of p^A, the points in the MBR cannot decrease the minimum distance further.
- Case 3: $MinDist(p^A, M^B) > r$. If the lower bound of the distance is larger than the current search radius (r), the points in the MBR are outside the search area and can be pruned.

For *Case 1*, we should cull p^A because the distance from p^A to its nearest neighbor will not be greater than $cmax$; continuing the search for the nearest neighbor will not increase $cmax$. For *Case*

2 and *Case 3*, we should cull all the points in M^B because they do not help to reduce *cmin*. If a point-MBR pair passes the above checks, we still have the opportunity to cull p^A again with EB.

Incorporating Early Break. The EB and the NN algorithms are not mutually exclusive. We further optimize *X-HD* by incorporating the EB strategy to prune points that will not affect the final HD value. For instance, if the distance from p^A to any point in M^B is no greater than *cmax*, p^A can be safely discarded.

Algorithm 8: CalculateMinDist

Input : Point $\mathbf{p^A}$, MBR $\mathbf{M_B}$, Current minimum distance **cmin**, Global maximum distance **cmax**, Search Radius **r**, Worklist $\mathbf{WL_{Heavy}}$
Output : Updated current min distance from p^A to points in M_B

```
1  if MaxDist(p^A, M^B) ≤ cmax then
2  |  return ∞                                              // Prune p^A by Case 1
3  else if MinDist(p^A, M^B) ≥ cmin then
4  |  return cmin                                          // Prune M_B by Case 2
5  else if MinDist(p^A, M^B) > r then
6  |  return cmin                                          // Prune M^B by Case 3
7  end
8  if |M^B| <offloading threshold then
9  |  for each p^B in M^B do
10 |  |  dist ← ||p^A, p^B||
11 |  |  if dist ≤ r then
12 |  |  |  cmin ← min(cmin, dist)
13 |  |  end
14 |  |  if dist ≤ cmax then
15 |  |  |  return ∞                                       // Early break
16 |  |  end
17 |  end for
18 else
19 |  WL_heavy ← WL_heavy ∪ (p^A, M^B)                     // Heavy cell
20 end
21 return cmin
```

We incorporate the three conditions into a distance computation function, as listed in Algorithm 8. In Line 1-7, the return of an infinity value means we should discard p^A, and the return of *cmin*

without computation indicates the points in the MBR should be discarded. After passing these checks, it is worthwhile to visit the points in the MBR to calculate distances. Line 8 checks whether the number of points in M^B is lower than a given threshold to determine whether we should visit the point in the shader. From Line 9 to Line 17, each point in the cell will be sequentially visited to calculate the minimum distance. Whenever we find the distance of a pair of points that is no greater than *cmax*, we can stop the computation and discard p^A according to the early break condition (Line 14-16). If a cell contains too many points, we postpone the computation and store the intersection (p_i^A, M_j^B) in the worklist WL_{Heavy} for later processing in a dedicated CUDA kernel.

4.4.4 Offloading Heavy Cells

Introducing a uniform grid also leads to an imbalanced workload issue. Considering that real-world datasets are not uniformly distributed, some cells may contain very few points (“Lightweight Cells” in Fig. 31), while some cells may contain a large volume of points ("Heavy Cells"). Due to the constraints of the RT cores’ single-ray programming model, we have to sequentially process the points in the cell with the shader, which results in poor performance.

To improve load balance, we only offload the work of heavy cells to a dedicated CUDA kernel. One might wonder why only offload heavy cells instead of processing all intersections entirely with the CUDA kernel. There is a trade-off between using the RT shader and the CUDA kernel to calculate distances. If we entirely offload the distance computation to the kernel, we have no chance to update the global maximum *cmax* during the BVH traversal. Therefore, we cannot timely prune points with the early break optimization, leading to more redundant computation. In practice, we set the threshold to the load-balancing kernel’s thread block size (calculated by `cudaOccupancyMaxPotentialBlockSize`) to strike a balance between a better load balance and

high occupancy. Because offloading cells with fewer points than the block size would leave threads idle, it is more efficient to process these lightweight cells serially directly in the RT shader.

Algorithm 9: TackleHeavyCells CUDA Kernel

```
   Input  : Incomplete minimum distance partial_cmin, Global maximum distance cmax, Worklist WL_Heavy
   Output : Updated cmax
1  for each thread block takes point p^A ∈ WL_Heavy do
2      __shared__ point_t p^A
3      __shared__ float cmin
4      cmin = partial_cmin[p^A] // Resume computation from the incomplete cmin computed by
           InIntersection shader
5      __syncthreads()
6      for each MBR M^B ∈ WL_Heavy that p^A intersects do
7          p^B ← M^B[threadIdx.x]                          // Each thread takes a point p^B
8          dist ← ||p^A, p^B||
9          min_dist = BlockReduce(dist)
10         if threadIdx.x = 0 then
11             if min_dist ≤ cmax then
12                 cmin ← ∞                                 // Flag down early break
13             else
14                 cmin ← min(cmin, min_dist)
15             end
16         end
17         __syncthreads()
18         if cmin = ∞ then
19             break                                        // Thread block exits the loop
20         end
21     end for
22     __syncthreads()
23     if threadIdx.x = 0 and cmin ≠ ∞ then
24         atomicMax(&cmax, cmin)
25     end
26 end for
```

Algorithm 9 shows the CUDA kernel for processing heavy cells. The outermost for loop iterates over each point in WL_{Heavy} with a thread block. In Line 4, we retrieve the incomplete minimum distance *cmin* computed in `IsIntersection` shader. Both p^A and *cmin* are stored in shared memory for fast processing. After that, the entire thread block will cooperatively visit each

intersected MBR M^B to maintain *cmin*. In Line 7, each thread in the block takes a point in M^B and calculates the distance. A block reduction primitive is used to aggregate the minimum distance collected from each thread. If any distance is no greater than *cmax*, we will notify the entire thread block to break the loop and give up the processing of all following intersections. In Line 22, we synchronize the block to ensure *cmin* is ready, and it is then applied to *cmax* with the $atomicMax$ instruction.

4.5 Implementation Details

Finding Initial Radius. We use the lower bound of $\overrightarrow{H}(P^A, P^B)$ as the starting search radius r_0 because this value represents the smallest possible value of search radius. The lower bound can be easily calculated using the MBRs of P^A and P^B [155].

Growing Search Radius. We grow the search radius by r_d after each iteration. If r_d is very small, many iterations may be needed to lift the radius to an appropriate value that allows p^A to hit an extended cell, and each iteration is not free. It has to cast rays and adjust the BVH for the new search radius. Conversely, if r_d is very large, we may only need one more iteration to complete the algorithm. However, we could face excessive intersections, which would be more costly than the overhead of adjusting the BVH.

We propose an adaptive radius growing strategy based on our observation. The size of the worklist WL_{In} decreases sharply at the beginning of the iterations. After a few iterations, the size of WL_{In} changes slowly. This can be explained by the presence of a few outliers that are very far from the dataset center. Based on this observation, we should adjust r according to the changing rate of WL_{In}. If the size of WL_{In} sharply decreases, we should also increase r quickly to reduce iterations. If WL_{In} decreases slowly, we should carefully increase r to avoid too many unnecessary intersections.

Algorithm 10 explains this idea. We set four predefined expansion factors: $\{8,4,2,1\}$. If the worklist size decreases slowly, e.g., by less than $\frac{1}{8}$, r is increased quickly by eight times of the grid cell diagonal. Note that the search radius r must not exceed the upper bound of HD, so we constrain its value in Line 9.

Algorithm 10: GetNextRadius

Input : Current search radius r, Current worklist WL_{In}, Next iteration worklist WL_{Miss}
Output : New search radius r

1 $wl_reduce_ratio \leftarrow \frac{|WL_{In}|-|WL_{Miss}|}{WL_{In}}$
2 $radius_expand_factors \leftarrow \{8,4,2,1\}$
3 **for each** $expand_factor$ ***in*** $radius_expand_factors$ **do**
4 **if** $wl_reduce_ratio < \frac{1}{expand_factor}$ **then**
5 $r \leftarrow r + expand_factor * cell_diagonal$
6 **break**
7 **end**
8 **end for**
9 $r \leftarrow min(r, MaxDist(M^A, M^B))$ `// Max value of the radius`
10 **return** r

Adaptive Grid Sizing. The grid resolution greatly impacts the performance of *X-HD*. If the grid is too sparse (low resolution), the number of cells is also low. Consequently, the number of intersections will be low, spending less time on BVH traversal. However, the number of points in cells will be much larger. In the extreme case of a grid having only a single cell, *X-HD* effectively acts as the EB solution. Conversely, if the grid resolution is very high, the number of intersections will also be very high, but each cell contains a minimum number of points, resulting in fewer nearest neighbor candidates. Therefore, the best grid resolution depends on the trade-off between the cost of the BVH search and the cost of the distance calculation for candidate points in the cell. To approximate the optimal grid size, we define a cost function $C(l)$ for query point p^A based on the cell side length l, as shown in Equation 4.2.

$$C(l) = c_{hit} * I + c_{dist} * P \tag{4.2}$$

c_{hit} is a constant cost associated with a ray hitting an MBR, and c_{dist} is a constant cost of calculating the distance between a pair of points. I is the number of intersections of the ray originated at p^A, and P is the total number of points covered in the search area. Since we extend the MBRs by r, the actual volume indexed is $(l+2r)^n$, where n is the number of dimensions, and the volume of a cell is l^n. A single query point will intersect multiple logical cells when $l \ll r$ because the extended MBRs will overlap. Therefore, I can be approximated with the ratio of their volumes (Equation 4.3).

$$I = \frac{(l+2r)^n}{l^n} = (1+\frac{2r}{l})^n \tag{4.3}$$

P can be approximated by the search volume multiplied by the point density of P^B, which is listed in Equation 4.4.

$$P = \rho(l+2r)^n \tag{4.4}$$

Summing the above components, we have Equation 4.5:

$$C(l) = c_{hit} \cdot (1+\frac{2r}{l})^n + c_{dist} \cdot \rho(1+2r)^n \tag{4.5}$$

To find the minimum cost $C(l_{opt})$, we take the derivative of $C(l)$ with respect to l and set it to zero $\frac{dC}{dl} = 0$, as $C(l)$ is strictly convex. This yields Equation 4.6:

$$l_{opt} = (\frac{2 \cdot r \cdot c_{hit}}{\rho \cdot c_{dist}})^{\frac{1}{n+1}} \tag{4.6}$$

Equation 4.6 intuitively describes that if the ratio (c_{hit}/c_{dist}) is high, we should increase l to reduce intersections. If the distance calculation is expensive (e.g., high-dimensional space), we should use a finer grid to narrow down candidate points. In practice, we use the initial radius r_0 as r, and c_{hit} and c_{dist} are determined using the clock() instruction with a profiling kernel when *X-HD* is launched, which makes it adaptive to different GPUs. We evaluate the effectiveness of this method in §4.6.6.

4.6 Evaluation

4.6.1 Settings

4.6.1.1 Baselines

We only consider exact HD solutions for a fair comparison, as *X-HD* is a general exact HD algorithm.

ITK. HD has a very important application in medical imaging to evaluate the performance of a segmenting algorithm [146, 204, 214]. We include the state-of-the-art industrial medical imaging segmentation tool, the Insight Toolkit (*ITK*), as a baseline. *ITK* supports parallel processing on the CPU. Note that the algorithm that *ITK* used is exclusively designed for processing voxels [136]. It cannot process floating numbers or two datasets with different sizes, so we only run MRI datasets on it.

NN-KD. *NN-KD* is a CUDA-based nearest neighbor search implementation using a KD-tree, and we use the *cukd* library as it is a high-quality implementation [217]. The introduction of this baseline is to show that a tree-based method without pruning techniques on the GPU is not very efficient.

NN-Clover. *Clover* is the state-of-the-art KNN library for GPU. *Clover* adopts a spatial-graph-based method, and it claims that it is generally faster than the RT-based method [92]. We evaluate

its performance for the HD algorithm. The pruning technique also cannot be used in *Clover*, as it is designed for general KNN search.

EB. *EB* represents a GPU-based early break implementation of the HD algorithm. *EB* is very GPU-friendly due to its simple logic flow. There is no widely recognized *EB* solution available, so we implement it by ourselves.

RT-HDIST. *RT-HDIST* is the latest and the only work that exploits RT cores to compute HD [102]. *RT-HDIST* differs from us in the following ways.

1. Different Grid Design. *RT-HDIST* adopts a quantized grid to group points. The quantized grid maps all floating-point coordinates to an integral range of $[0, 2^k]$, where k is the bit count that determines the grid size. Subsequently, a cluster of points within a cell is represented by the cell's top-left corner, which is called a "representative point." By using the grid, *RT-HDIST* creates AABBs centered only at the representative points instead of at every individual data point.

2. No pruning techniques. Since all coordinates are converted into integers, the original boundary information of the cells is lost. Therefore, their quantized grid design cannot benefit from various pruning techniques like in *X-HD*.

3. No load balance considerations.

4. No adaptive radius adjustment. *RT-HDIST* simply grows the radius by the length of the cell diagonal in each iteration.

5. No automatic grid-sizing. *RT-HDIST* relies on the users to manually tune the grid size.

4.6.1.2 Parameters

For *NN-KD*, it has a parameter to decide the leaf size; we use its default value of 8. *Clover* has a compile-time parameter H, which is the number of hubs. We found that $H = 256$ yields overall good performance. *RT-HDIST* has a parameter called bit count to determine the resolution of its quantized grid. For graphics datasets, we use the parameters presented in their paper [102]. For the MRI datasets, we use its default setting, which is 8. *X-HD* does not require any hand-tuned parameters.

4.6.1.3 Datasets

We collected a diverse set of datasets, including MRI images, geospatial datasets, and graphics models. The MRI datasets are from the Multimodal Brain Tumor Segmentation Challenge 2020 (BraTS) [16, 17, 143], comprising 494 images. Each image includes different types of tissue relaxation time. The locations of non-empty voxels are extracted as the coordinates. The names of datasets starting with US are US-based maps, such as the boundaries of counties [40], zip codes [50], census blocks [51], and water areas [49]. *OSMLakes* and *OSMParks* represent the boundaries of lakes and parks worldwide [46]. These large-scale spatial datasets are widely used for performance evaluation of spatial data processing systems [12, 46, 65, 219]. The graphics models come from The Stanford 3D Scanning Repository [41, 63, 107, 215]. We include them for an equivalent evaluation as they are also used in *RT-HDIST*. We also summarize the statistics of the datasets in Table 7, including the number of points, dimensionality, and data distribution characteristics. To reflect the skewness of the datasets, we use a uniform grid to partition the datasets and calculate the Gini index of the number of points in the cells, a commonly adopted method to describe the skewness of data distribution [142]. From the statistics, we can see that MRI datasets have a very low skewness (low Gini index), while the graphics and geospatial datasets can be very skewed.

4.6.1.4 Hardware

We use an NVIDIA RTX 3090 for the GPU-based artifacts. *ITK* is the only CPU-based software, and we evaluate it on a workstation equipped with an Intel i9-12900.

4.6.1.5 Measuring

For GPU-based methods, we start timing when the datasets are loaded onto the GPU and stop timing when the HD is produced. Note that no profilers are currently available to profile the utilization of RT cores [133, 151]. Therefore, we use the total running time and the number of intersections to analyze the execution patterns for RT-based methods.

4.6.2 Overall Performance

Fig. 32(a) shows the running time distribution for the BraTS datasets. Since BraTS has hundreds of images, we follow an evaluation method similar to Taha et al. [204] by randomly selecting 250

Table 7. Datasets and their statistics. The Gini index is used to describe the skewness of the data.

Dataset	Category	#Dims	#Points or (max, avg, median)	Gini Index or (max, avg, median)
BraTS	494 MRI images	3D	(1.9M, 1.5M, 1.5M)	(0.17, 0.14, 0.14)
USCounty	Geospatial	2D	9.4M	0.77
USZipcode	Geospatial	2D	43.9M	0.61
Lakes	Geospatial	2D	301.7M	0.76
Parks	Geospatial	2D	403.7M	0.76
USWater	Geospatial	2D	22.8M	0.61
USCensus	Geospatial	2D	52.3M	0.65
All Nodes	Geospatial	2D	2.7B (Partially used)	–
Dragon	Graphics	3D	0.4M	0.42
AsianDragon	Graphics	3D	3.6M	0.38
HappyBuddha	Graphics	3D	0.5M	0.46
ThaiStatuette	Graphics	3D	4.9M	0.47

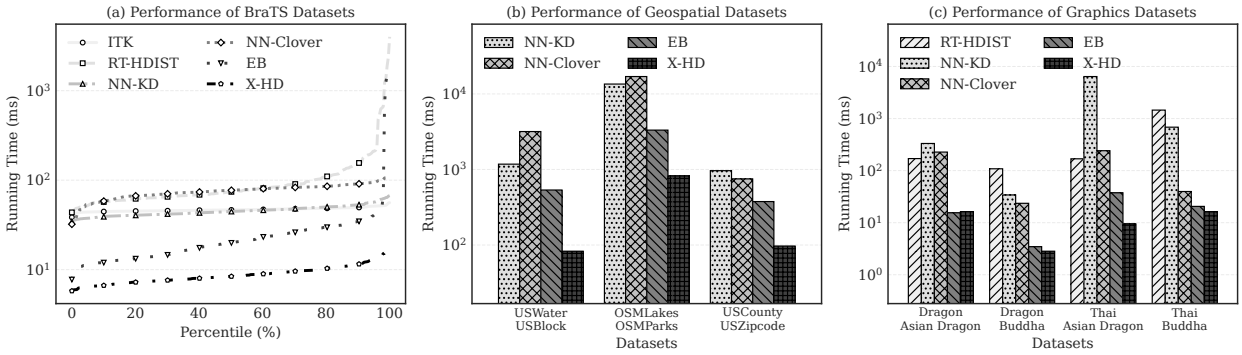


Figure 32. Comparing Overall Performance of Hausdorff Distance Solutions. (a) Running time distribution of six method on BraTS datasets; (b-c) End-to-end execution time on geospatial and graphics datasets

pairs of images and plotting the running time distributions. For example, a data point at $x = 20$ and $y = 5$ indicates that the running time for no more than 20% of the data points is no more than 5ms. We first examine the industrial baseline, *ITK*, which exhibits a very flat running time line, meaning it is less susceptible to data distributions, with an average and median running time of 46.8ms and 46.5ms, respectively. *NN-KD* produced performance numbers very similar to *ITK*, despite being a GPU-based method. *NN-Clover* and *RT-HDIST* are the least performant methods for these datasets. Since *EB* is a GPU-based brute-force solution with pruning techniques and is very efficient on the GPU, it is the best baseline in the majority of cases. However, the running time after the 95^{th} percentile is very long because, in these cases (similar distances between points), the EB condition is rarely triggered. Finally, *X-HD* consistently outperforms all baselines for all data points due to our efficient utilization of RT cores, pruning techniques, and better load balancing. The average running time is 8.9ms, and the median time is 8.4ms, showing that our method is less susceptible to data distributions, a similar trend to *ITK*. We will analyze why *X-HD* is so performant in §4.6.3 and §4.6.4.

Fig. 32(b) shows the performance numbers on the geospatial datasets, which have a large number of points and high skewness. We could not include *RT-HDIST* because it cannot finish the experiment within an hour. We first look at *NN-KD*, which took 1176ms on *USWater-USBlock* datasets, with 94.8% of the time spent on building the KD-tree. *Clover*'s NN search took about 3150ms, with 63ms to search for the closest hub and 2.3s to search for nearest points in the hub. The GPU-based *EB* took about 535ms, which is about 2.2× faster than *NN-KD*. Finally, *X-HD* only took 83ms, achieving 6.4× speedups over *EB* due to our comprehensive optimization techniques. Fig. 32(c) shows the performance numbers over medium-sized and skewed datasets. We can see that *X-HD* is the fastest except in one case, *Dragon-Asian Dragon*, where *EB* took about 15.4ms while *X-HD* took 16.2ms. For the *Dragon-Buddha*, *Thai-Asian Dragon*, and *Thai-Buddha* datasets, *X-HD* outperforms *EB* with speedups of 1.2×, and 3.9×, 1.3×, respectively. The performance of *RT-HDIST* is not competitive due to excessive ray-AABB intersections and distance computations, even with RT cores. As a result, *X-HD* outperforms *RT-HDIST* with speedups of 10.5×, 38.3×, 17.9×, and 90.2×, respectively, for these four pairs of datasets.

4.6.3 Effectiveness of Pruning Techniques

We measure the running time as we gradually enable the optimizations, as shown in Fig. 33(a). To better analyze the behaviors of the optimizations, we measure the number of MBRs being hit (*Intersections*) and the number of pairs of points used in the distance calculation functions (*Visited Point Pairs*), as listed in Fig. 33(b). **No-Opt** stands for a vanilla uniform grid without any optimizations, which took about 8 seconds on *Dragon-Asian Dragon* datasets, where the number of intersections is about 165 million and 1.8×10^{11} pairs of points were involved in the distance calculation. Note that *No-Opt* is not equivalent to *RT-HDIST* due to the different grid design. After enabling **EB**, the running time reduced to about 2 seconds, the number of intersections went down

to 21 million, and the number of point pairs was reduced to 2.3×10^{10}. We can see that by enabling EB, both the intersections and visited points are reduced because some points from P^A are discarded as they cannot increase *cmax*. After enabling the **Prune** optimization with the help of HD estimators, the number of intersections almost stayed the same. However, the number of points visited was reduced by more than two orders of magnitude, to about 589 million. With HD estimators, we can skip many points that cannot reduce *cmin*. Since *RT-HDIST* does not have any pruning techniques, it results in more intersections than our method, which is very costly to process. However, the number of points visited by *RT-HDIST* is lower than ours, which could be a finer grid they used. Searching for an intersection is more expensive than processing a pair of points. For example, we empirically measured the former step using the `clock()` API, which takes about 1300 cycles on an RTX 3090, while the latter takes only about 260 cycles. As a result, *X-HD* outperforms *RT-HDIST* by 3.1× (without enabling load balance).

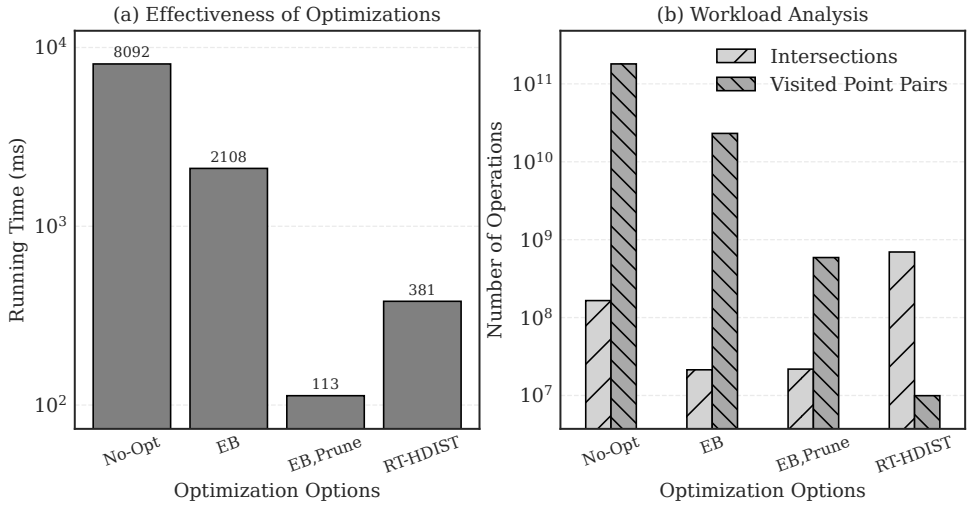


Figure 33. (a) Running time analysis by enabling optimization options on *Dragon-Asian Dragon* datasets; (b) Breakdown workloads by counting the number of intersections and the number of points involved in distance computation

4.6.4 Effectiveness of Load-balance Techniques

We analyze the effectiveness of the load balance techniques with geospatial and graphics datasets because they result in very skewed grids. Fig. 34(a) shows the breakdown of the total execution time with the option of load-balancing optimization. *OSMLakes-OSMParks* is the dataset most susceptible to the load-imbalance issue due to the skewness of the grid. Without enabling the offloading kernel to improve load balance, the RT shader took 3390ms to visit points in the grid cells and calculate point distances. After enabling the offloading kernel, the RT shader only took 223ms and the additional CUDA kernel took 254ms. The total time for the RT shader and the kernel is only 477ms, which is 7.1× faster than using the RT shader alone. Fig. 34(b) shows that offloading heavy cells is always beneficial, producing speedups of 7.6×, 2.9×, 4.4×, and 2.5×.

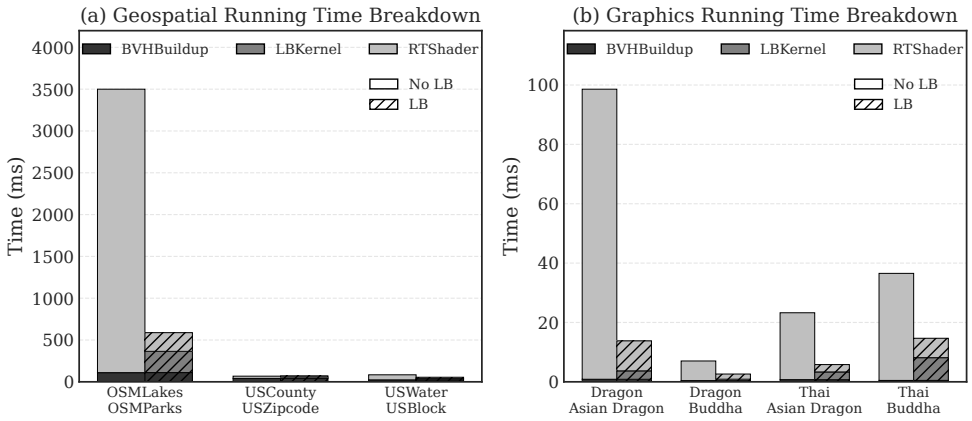


Figure 34. (a) Running time breakdown of *X-HD* and comparing the effectiveness of using load-balancing (LB) on geospatial datasets; (b) The same experimental settings but with graphics datasets.

4.6.5 Evaluation of Radius Growing Strategies

Fig. 35 compares three radius growing methods. “Add by Diagonal” method increases the search radius by the length of the cell diagonal for every iteration, which is used by *RT-HDIST* [102]. “Double Radius” method doubles the search radius, which is used by *RT-KNNS Unbound* [152]. We can see that “Double Radius” favors the geospatial datasets, while “Add Diagonal” shows consistent high-performance on graphics datasets. By looking into the running logs of the two methods, we found that “Add Diagonal” took 300 iterations on the *OSMLakes-OSMParks* datasets due to the conservative radius growing strategy, while “Double Radius" only took 7 iterations. Since the extent of geospatial datasets tends to be very high, a slowly growing radius will result in many iterations to converge. In turn, the aggressive growing strategy of “Double Radius” causes too many unnecessary intersections, resulting in consistently poor performance on the graphics datasets. Finally, our method adaptively grows the radius according to the changing ratio of the worklist, making it consistently good for all datasets.

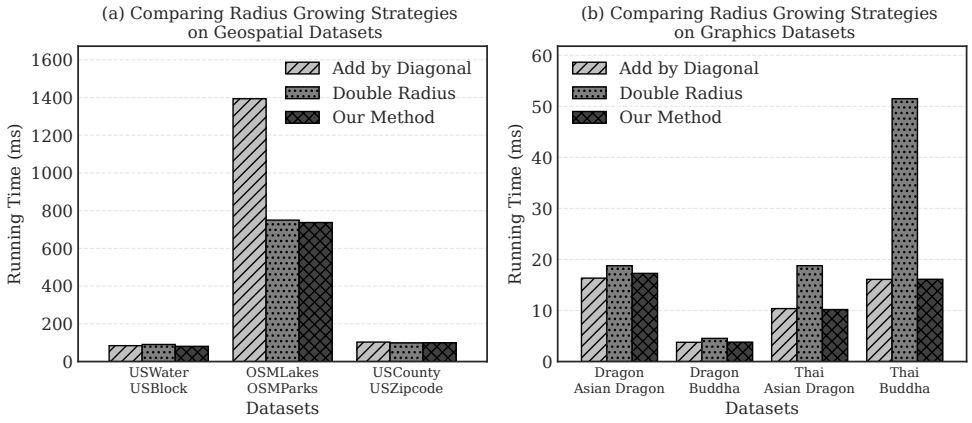


Figure 35. Comparing three radius growing strategies. “Add Diagonal” increases the radius by the length of the cell diagonal; “Double Radius” doubles the search radius

4.6.6 Evaluation of Adaptive Grid Sizing

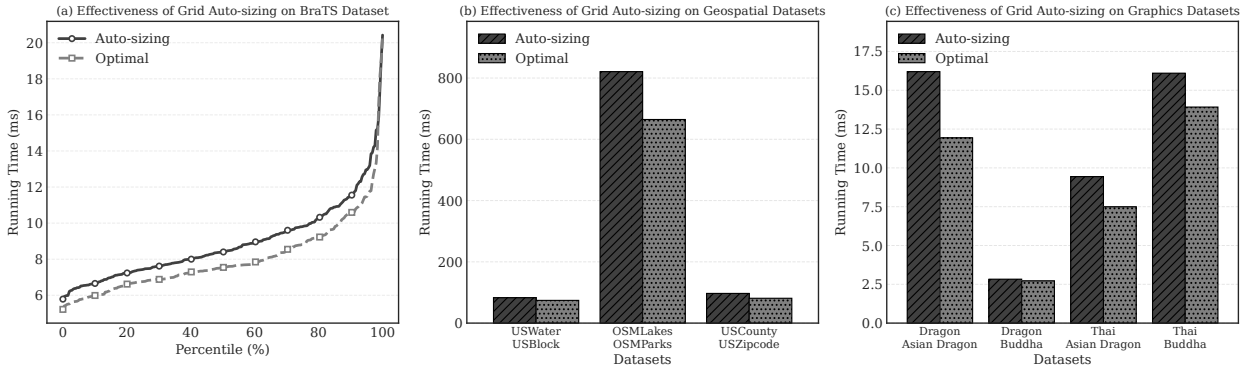


Figure 36. Comparing the running time of a grid with the adaptive sizing technique to the optimal grid on all datasets.

To evaluate the effectiveness of the grid-tuning technique in §4.5, we compare the running time to a grid with an optimal grid size for each dataset. The optimal grid size is determined with a brute force search, which is prohibitively expensive and cannot be accepted in practice. The two curves in Fig. 36(a) show the running times for BraTS datasets. We can see that the running time from an auto-sizing grid is only slightly slower than the optimal grid resolution, with an average and median slowdown of 12.9% and 5.1%, respectively. For the worst case, the auto-sizing grid is 123% slower than the optimal grid resolution. The very close performance numbers indicate that our cost-based grid-tuning method is highly accurate.

Fig. 36(b) and (c) compare the running times between the auto-sizing grid and the optimal grid on geospatial and graphics datasets. For *USWater-USBlock*, the optimal grid took 74ms while the grid with adaptive sizing took 83ms, which is 12.2% slower than the optimal grid. For the largest *OSMLakes-OSMParks* dataset, the optimal grid took 665ms, while using our method took 821ms,

experiencing a slowdown of 23.5%. For the graphics datasets, the slowdowns of the auto-sized grid are 35.7%, 4.1%, 25.9%, and 15.7%, respectively.

4.6.7 Scalability

We evaluate how these methods behave as we increase input size. We used a subset of the *All Nodes* dataset [46]. P^A and P^B are from the same dataset, where P^B is translated along the x-axis by 0.5% to make them not completely the same. Fig. 37(a) shows the runtime of *EB-GPU*, *NN-RT*, and *X-HD* as the number of points increases. When the number of points is 12.5M, *NN-KD* and *NN-Clover* took 7444ms and 450ms, respectively, and *EB* took 283ms. *X-HD* only took 68ms, which 4.2× faster than the best baseline. Subsequently, we doubled the input size, and the running time of *NN-KD* increased 4.3×. *EB* and *NN-Clover* methods exhibit more favorable scalability profiles, increasing by about 1.9×. While *X-HD* increased by a factor of 2.6×, it is still significantly faster than all other methods. When increasing the input size from 100M to 200M, the running time of *NN-KD* increased about 2.3×. *EB* increased 2.5×. While *NN-Clover* increased 2.1×. *X-HD* increased by about 2.9×, which falls within the range of its $O(n \log n)$ time complexity.

4.6.8 Sensitivity to Overlap

The performance of HD algorithms can be susceptible to data distribution. For example, the NN-based algorithms favor two highly overlapping datasets, while the EB-based algorithm favors two non-overlapping datasets [204, 246]. We aim to evaluate how *X-HD* behaves by changing the degree of overlap between two datasets. Fig. 37(b) reports the running times as we gradually change the degree of overlap. This change is achieved by translating P^B along the x-axis by a distance proportional to 0.01% up to 0.6% of the extent of P^B's MBR. When the two datasets are highly overlapping (0.01%), *EB* performs the worst because the EB condition is rarely satisfied,

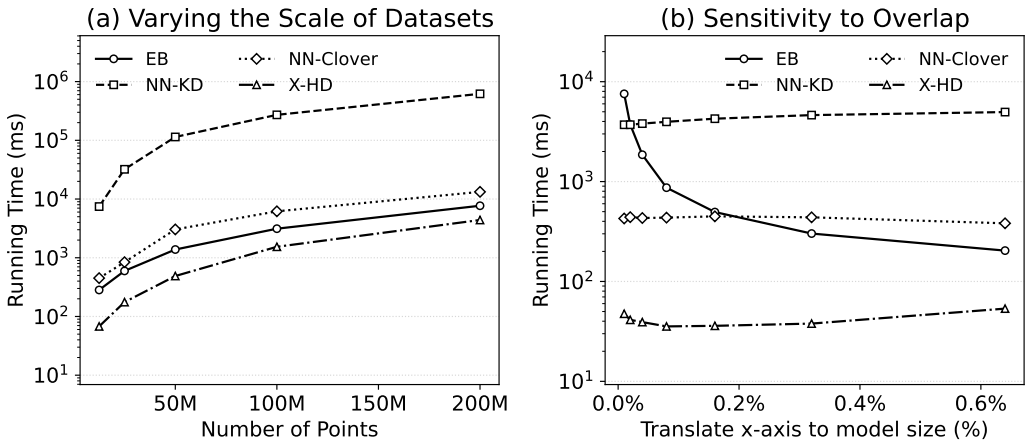


Figure 37. (a) Evaluation of scalability by increasing points; (b) Sensitivity to overlap by separating two datasets

deteriorating into a brute-force solution with a running time of 7561ms. In such a case, the NN-based method is favorable due to the less intensive index search. As the degree of overlap between the two datasets decreases, the running time of *EB* gradually decreases. Conversely, the running time of *NN-KD* increases because more branches in the KD-tree must be searched. *NN-Clover* is less susceptible to the degree of overlap, as its running time remains almost stable. *X-HD*, being an NN-based method incorporated into EB, is also less susceptible to the degree of overlap as it incorporates the advantages of these two algorithms.

4.6.9 Memory Footprint

Maintaining a low memory footprint is crucial for processing large datasets within the limited device memory. The EB algorithm requires no temporary memory, as it relies on the double-loop execution pattern. Conversely, NN-KD, NN-Clover, and *X-HD* require auxiliary memory to construct spatial indices. Among these, NN-KD generally consumes the least memory because data points can be naturally structured into a compact KD-tree.

Fig. 38 illustrates the temporary memory footprints across the geospatial and graphics datasets. As shown in Fig. 38(a), *X-HD* requires slightly more memory than NN-KD on the larger geospatial datasets. For example, on the OSMLakes-OSMParks dataset, NN-KD occupies 3080 MB of memory, while *X-HD* uses 3929 MB. For *X-HD*, this allocation breaks down into 9.5 MB for the BVH, 1542.3 MB for the grid, and 2301.8 MB for the worklist used to track heavy cells. Averaging the memory usage across the geospatial datasets, NN-KD consumes 1271 MB, NN-Clover requires 3614 MB, and *X-HD* uses 1549 MB—which is only marginally higher than the KD-tree.

On the graphics datasets, *X-HD* exhibits a more dynamic memory footprint. Because the BVH and grid require significantly less space for these inputs, most of the memory is allocated to the worklists, which may vary substantially. On average, NN-KD consumes 23.8 MB, NN-Clover takes 60.5 MB, and *X-HD* uses 46.0 MB. Given the substantial performance improvements offered by *X-HD*, this slight memory overhead is well justified.

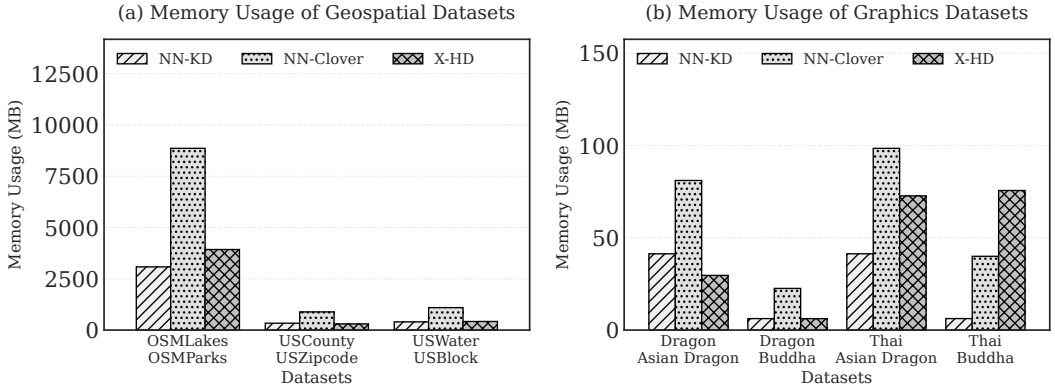


Figure 38. (a) Memory usage of geospatial datasets; (b) Memory usage of graphics datasets

4.7 Related Work

Hausdorff Distance. Nutanong et al. first applied branch-and-bound to accelerate HD computation [155], introducing a best-first search with a hierarchical priority queue. Taha et al. proposed the EB technique, achieving near-linear average complexity on medical images [204]. Zhang et al. [246] classified datasets into three categories and designed a hybrid framework to select algorithms accordingly. Chen et al. addressed EB inefficiency with a local-start search [35], while Ryu et al. improved pruning with points-ruling-out and random sampling [189]. These methods primarily target CPUs and rely on structures unsuitable for GPUs. Distorch [185] provides a GPU library, but it estimates the 95% HD, mainly for medical applications. MedVoxelHD is a GPU-accelerated morphological Hausdorff, which only supports voxel processing and does not support Euclidean distance [146].

Distance Computation on the GPU gDist is a GPU-based parallel algorithm that efficiently computes the minimum/maximum Euclidean distance between general triangle meshes through improved traversal and culling techniques [55]. Their problem is related but distinct from ours: HD calculates the maximum of minimum distances for each point, while gDist calculates the minimum of minimum distances and the maximum of maximum distances.

Spatial Data Processing with RT Cores. Kim et al. compute point-to-face HD with RT cores [103], whereas we address point-set HD, a more general and computationally intensive problem. Evangelou et al. reformulated radius search as ray tracing [53], extended by RTNN for kNN with optimized scheduling [258], TrueKNN for arbitrary-radius kNN [152], and Arkade for non-Euclidean kNN [133]. RayJoin focuses on point-in-polygon and line segment queries [64]. LibRTS offers a general-purpose RT–accelerated spatial index with dynamic geometry updates [65].

Applications of Hausdorff Distance. HD has been widely applied in trajectories [80, 110, 112, 118, 174, 234, 236, 247, 253, 256], images [15, 37, 54, 82, 101, 147, 205, 257], geometry [10, 81, 105],

and spatio-temporal data [116]. However, HD is usually treated as just one distance function, without dedicated optimization, leaving its high computational cost a bottleneck. Our approach makes HD practically viable for large-scale analysis by reducing this overhead.

Hausdorff Distance in Databases. PostGIS [145, 175] provides `ST_HausdorffDistance`, based on GEOS, but it is single-threaded and brute-force, lacking optimizations such as EB or NN and hardware acceleration. This results in poor performance on large or high-resolution datasets. Our algorithm addresses these limitations through algorithmic and hardware-based optimizations.

4.8 Conclusion

A central challenge in performance optimization for increasingly complex data analysis applications in the era of domain-specific architectures is how to effectively select, implement, orchestrate, and optimize algorithms to meet computational demands across heterogeneous hardware platforms. This chapter introduces *X-HD*, the fastest Hausdorff Distance computation algorithm, which fully leverages both general-purpose and ray tracing cores in modern GPUs. The strength of *X-HD* lies in using hardware-acceleration and applying domain-specific optimizations at the same time, making it a unified solution capable of diverse workloads while maintaining consistently high performance. We believe that *X-HD* not only delivers a fast Hausdorff Distance computation tool for spatial database systems and other applications, but also serves as a compelling example of effective algorithm engineering for emerging heterogeneous hardware resources.

Chapter 5: Integrating RT Acceleration into a Spatial Database

5.1 Introduction

The volume of geospatial data has grown exponentially with the widespread adoption of GPS-enabled devices, satellites, and automated driving systems, making large-scale spatial data analysis an increasingly urgent requirement [36, 243, 254]. To address this big-data challenge, researchers have developed a new spatial analytics ecosystem characterized by large-scale computing and storage capacity, high scalability, compatibility with low-cost commodity clusters. A representative early example of this effort is HadoopGIS [3]. Consequently, most modern OLAP databases adopt this "scale-out" design principle, processing data in batches to exploit massive parallelism across clusters. Following this paradigm with efforts from both academia and industry, the most widely used spatial database, Apache Sedona [239, 240], has been developed.

While distributed spatial systems effectively address rapidly growing data volumes, issues remain in processing complex geometries and meeting real-time interactive query requirements. These system-heavy databases are often ill-suited for agile analytics, where users iteratively modify SQL queries to explore data. In addition, they incur high end-to-end latency. For example, provisioning a distributed Apache Sedona cluster involves substantial deployment effort, machine costs, and inter-node communication overhead. Therefore, these systems are most cost-effective for massive datasets without strict low-latency requirements.

For spatial data processing with fast-response requirements, an alternative ecosystem has emerged based on single-node analytical engines running on powerful servers and following a "scale-up" paradigm. Research and development on these single-node engines has grown rapidly, enabling a wide range of applications [104, 163]. Representative systems include DataFusion [109], DuckDB [104, 179], GeoPandas [209], and Apache SedonaDB [8]. These systems have democratized access to OLAP capabilities by enabling interactive data exploration and visualization on local laptops and workstations.

However, this promise of "local interactivity" often breaks down due to a single machine's limited computational power. For example, spatial joins remain slow on CPUs [3, 88], causing queries over medium-sized datasets to stall for minutes or hours [7]. Such latency disrupts the interactive feedback loop and forces data scientists into costly waiting cycles. Spatial joins typically use a two-phase strategy: a filtering phase leveraging spatial indexes (e.g., R-tree [24, 32, 71]) to prune non-overlapping geometries, followed by a refinement phase evaluating geometric predicates [39] on the intersecting geometry pairs. Due to the limited capability of a single node, both stages may become performance bottlenecks.

In fact, a highly effective hardware accelerator for single-node spatial engines already sits on users' desks: Ray Tracing (RT) cores, widely available in modern consumer laptops and workstations [29]. During SQL execution, these cores remain idle because existing software stacks do not use them. Prior research demonstrates RT cores' potential to accelerate spatial workloads [64, 65, 152, 258]. However, no industrial-grade product yet exploits this capability. This gap motivates us to integrate RT-core acceleration directly into the production-quality SedonaDB engine, aiming to eliminate the "waiting game" in local spatial analytics.

While the algorithmic potential of RT cores for accelerating individual execution phases has been demonstrated in prototype systems and micro-benchmarks [64, 65, 195, 233], incorporating

these techniques into a robust production engine introduces a distinct set of systematic challenges for us to address:

- **Data Format Mismatch.** SedonaDB relies on the Well-Known Binary (WKB) format to maintain compatibility with data sources in GeoParquet [164] and computational geometry libraries like GEOS [69]. While WKB is ideal for compact serialization, its stream-oriented structure prevents the random geometry access required by massive GPU parallelism. Efficient refinement requires a GPU-native format that supports random access to arbitrary geometry.

- **Indexing Scalability.** Handling complex geometries during refinement typically requires fine-grained indexing (e.g., building an index for every polygon). Constructing and storing millions of micro-indexes on the GPU is prohibitive due to memory allocation overhead and the high latency of host-device synchronization.

- **Combinatorial Complexity.** The WKB format supports diverse geometry types (`Point`, `LineString`, `Polygon`, and Multi-variants). Considering both sides of the join, coupled with various predicates (e.g., `ST_Contains`). Hardcoding GPU kernels for hundreds of geometry-predicate combinations is unmaintainable and operationally infeasible.

- **Memory Management.** The physical execution plan in SedonaDB processes many partitions in parallel. Without strict resource awareness, this aggressive scheduling often disregards GPU memory quotas, leading to frequent Out-Of-Memory (OOM) failures when scaling to industrial datasets. In addition, spatial joins involve many temporary device memory allocations, which can make "small joins" even slower than CPU-based joins due to the allocation overhead.

To bridge the data format gap, we introduce a GPU-optimized storage format based on a Structure of Arrays (SoA) design. By segregating offsets, vertices, and geometry types, this format compactly

represents an array of geometries while enabling high-throughput random access. To resolve the indexing scalability barrier, we design a monolithic BVH construction schema. Instead of managing millions of small indexes, we arrange geometries on a 2D plane and encode geometry IDs into the unused Z-axis of the hardware-accelerated Bounding Volume Hierarchy (BVH). This novel mapping allows us to build a single, global index that saturates GPU resources without frequent host interaction.

To tame the combinatorial complexity, we introduce a unified RT-based *RelateEngine*. Rather than implementing ad-hoc kernels for every combination, this engine utilizes RT cores to compute the Dimensionally Extended Nine-Intersection Matrix (DE-9IM). This matrix serves as a universal topological descriptor, allowing a single optimized engine to resolve arbitrary geometry types and predicates. Finally, we adopt asynchronous memory allocation to reduce the allocation overhead and implement a memory-aware execution strategy that dynamically monitors available device memory and orchestrates chunk executions to prevent OOM errors.

We have developed an RT accelerated spatial join engine, called *RayBooster*, making the following contributions.

1. **System Integration:** We provide the first integration of RT-core acceleration into a production database system, accelerating both filter and refinement phases.

2. **RT-based RelateEngine:** We introduce a unified evaluation engine that computes DE-9IM using RT cores, supporting any combination of geometry types and spatial predicates with a single code path.

3. **Robust Execution:** *RayBooster* reuses SedonaDB's memory-aware scheduling that ensures the completion of queries under a given memory budget while aiming for high performance. An asynchronous memory allocation strategy is also used to reduce synchronization overhead.

4. **Performance & Cost:** We demonstrate that our approach achieves up to 5.93× speedup over SedonaDB on SpatialBench (§5.8.2) while reducing 59.02% operational cost, offering a cost-effective solution for industrial spatial analytics.

5.2 Background

5.2.1 Apache SedonaDB

SedonaDB is an OLAP engine designed for the high-performance analysis of spatial datasets that fit within a single node. Additionally, its architecture is not limited to a single machine. At Wherobots, it serves as the foundation of our next-generation WherobotDB production runtime. To handle distributed workloads, an internal project called SedonaGlue orchestrates multiple SedonaDB instances across a cluster. The specific mechanics of this orchestration fall outside the scope of this chapter.

SedonaDB is implemented in Rust and built on Apache DataFusion [109], which provides an extensible query planner and a vectorized execution engine to treat geospatial operations as first-class citizens. Figure 39 shows the architecture of SedonaDB.

Columnar In-Memory Datasets. SedonaDB adopts Apache Arrow as its in-memory storage format. While conventional spatial databases (e.g., PostGIS, DuckDB) store geometries as opaque BLOBs, incurring significant memory copy, serialization, and deserialization (SerDe) overhead. SedonaDB achieves zero SerDe cost by storing geometries as WKBs in Arrow binary arrays. This enables zero-copy data access and allows Arrow's computing kernels to exploit SIMD capabilities on modern CPUs to accelerate data manipulation. The storage layer also maintains spatial statistics (e.g., global bounding box, feature/point counts) to guide query optimization. SedonaDB also tracks the Coordinate Reference System (CRS) throughout the query lifecycle, ensuring data integrity by preventing operations on mismatched units or projections.

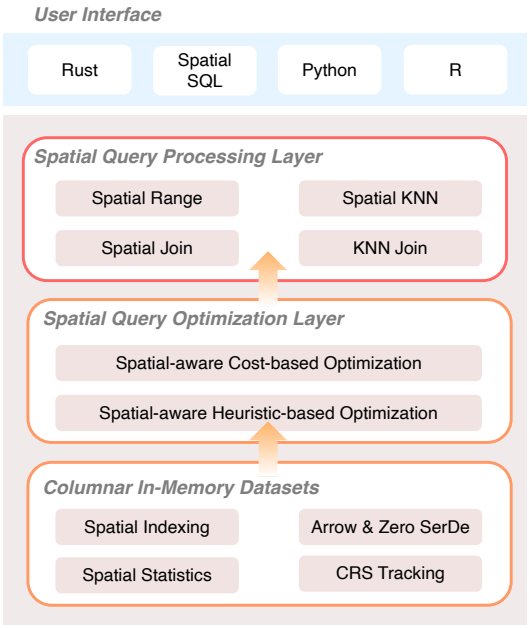


Figure 39. Architecture of Apache SedonaDB

Query Planning and Spatial Join. SedonaDB implements a logical optimization rule that pushes spatial predicates into unconstrained join nodes, transforming Cartesian products into spatial theta-joins. This structural rewrite ensures the physical planner identifies the operation as an index-supported execution. The spatial join follows the build-probe paradigm. The build phase materializes the smaller relation to construct an in-memory spatial index, such as an R-tree. The probe phase then streams the larger relation, querying this index via Minimum Bounding Rectangles (MBRs) to identify intersecting candidate pairs. This lookup constitutes the filter stage, a major performance bottleneck due to the massive volume of random memory accesses during tree traversal, as our subsequent experiments show that this stage can account for more than 30% of the spatial join execution time (§5.8.5).

Execution then proceeds to the refinement stage, where geometric computations apply a spatial predicate to them. In SedonaDB, this task is delegated to high-performance computational geometry

backends [18, 69]. With the WKB format, the engine eliminates the serialization overhead incurred when crossing library boundaries. This modularity ensures interoperability and allows the selection of the most efficient backend per data distribution. Based on storage-layer geospatial statistics, SedonaDB's cost-based optimizer may construct a feature-level index on either side to accelerate refinement for complex geometries. Finally, the refined candidate pair IDs are passed downstream to produce the query results. Determining the exact spatial relationship [34] often requires exhaustive comparisons of line segments, imposing a heavy computational load on processing complex geometries. In §5.8.5, we show that all queries take more than 60% time on the refinement stage.

Interfaces and Query Processing SedonaDB integrates with DataFusion by exploiting DataFusion's extensible optimizer API. In SedonaDB, geospatial functions (e.g., `ST_Intersects`) are registered as User-defined Functions (UDFs) within the DataFusion context. This allows the SQL parser to recognize spatial functions while maintaining compatibility with the standard SQL. SedonaDB supports both spatial join with relation predicates and KNN join. In addition, SedonaDB also provides interfaces for other programming languages for interoperability, including Rust, Python, and R.

Performance We use SpatialBench (§5.8.2) to compare SedonaDB against PostGIS, DuckDB, and GeoPandas on an AWS m7i.2xlarge instance. We focus on spatial relational joins: low-latency queries (Q2–Q6) and heavy joins (Q9–Q11). While Figure 40(a) shows PostGIS has competitive execution times, its significant index-building overhead negates this advantage for low-latency queries. DuckDB and SedonaDB achieve similar low-latency performance, whereas GeoPandas struggles at larger scales due to a lack of query optimization and multi-core underutilization. By contrast, DuckDB and SedonaDB leverage columnar layouts, vectorized execution, multi-core parallelism, and query optimization to achieve strong performance. SedonaDB excels on the intersection-over-union metric in Q9. For heavy joins (Q10–Q11), SedonaDB consistently delivers

strong results, aided by an adaptive strategy that selects the best partition-level algorithm based on spatial statistics. Meanwhile, DuckDB encounters scaling issues on Q10 that hinder interactive processing, and it fails to complete Q11 within an hour. Figure 40(b) shows that on large-scale datasets, SedonaDB is the only system capable of completing all queries.

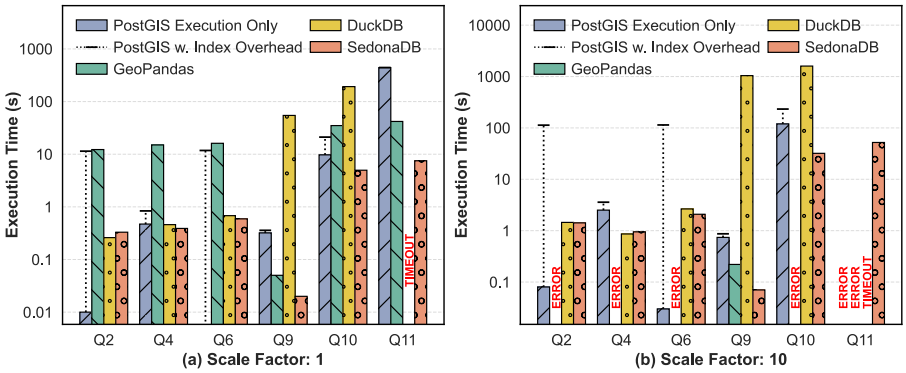


Figure 40. Performance comparison with SpatialBench.

5.2.2 Ray-tracing and RT cores

Ray tracing simulates light transport by tracing rays for photorealistic rendering. Historically computationally prohibitive for real-time applications due to massive ray-geometry intersection calculations, this changed when NVIDIA introduced RT cores [29]. These fixed-function hardware units accelerate the two most expensive pipeline stages: (1) BVH Traversal, locating potential intersecting primitives, and (2) Ray-Geometry Intersection, calculating the exact hit point. By offloading these tasks from general-purpose Streaming Multiprocessors (SMs), RT cores achieve orders-of-magnitude speedups over software implementations [29].

While originally designed for photorealistic rendering, the fundamental operation performed by RT cores, i.e., finding the intersection between a ray and a geometric primitive, is similar to many

common spatial search problems. Recent research has demonstrated the efficacy of repurposing this hardware for non-rendering workloads, such as serving as an index for relational databases [78], database scan [130, 195], nearest neighbor search [133, 152, 258], matrix-multiplication [248], and spatial queries [64, 65].

Most research on RT core repurposing relies on Ray-Axis Aligned Bounding Box (AABB) intersection tests. As shown in Figure 41(a), a ray is a half-line originating at point O with direction vector $\vec{d}$, parameterized by t. There are two intersection cases. *Case 1:* O is outside the AABB, but the ray hits its boundary. *Case 2:* O is within the AABB, regardless of boundary hits. Mapping search problems to ray tracing leverages the optimized BVH traversal of RT cores, achieving significant speedups over conventional methods. To exploit RT cores, developers can use NVIDIA OptiX [171], a specialized framework built on the CUDA ecosystem. Within this pipeline, users implement programmable shaders: the AnyHit shader handles each potential ray-geometry intersection, while the ClosestHit shader is triggered by the intersection nearest to the ray origin.

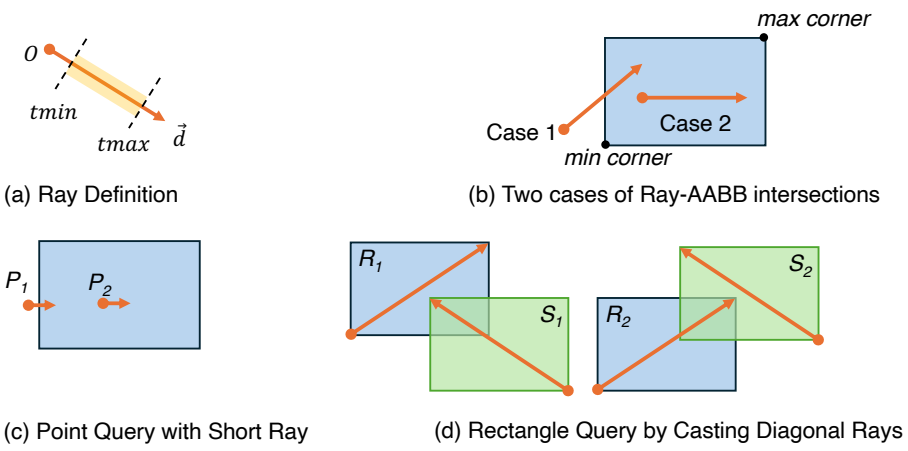


Figure 41. (a) and (b) illustrate the definition of the ray and the cases of ray-AABB intersection test; (c) illustrates the point query with short ray; (d) range query with diagonal rays

5.2.3 Spatial Query by Ray Tracing

As mentioned in §5.2.1, the execution of the spatial join consists of the filter and refinement stages. They are both amenable to the RT acceleration, as demonstrated in previous research works, *LibRTS* [65] and *RayJoin* [64], respectively.

LibRTS introduces a novel approach to accelerate point and range queries in the filter stage by repurposing RT cores. Specifically, it maps a point query to a ray-box intersection test with *Case 2*, where the query point serves as the ray origin. As shown in Figure 41(c), we cast short rays from points P_1 and P_2 to identify their intersections. RT cores will traverse the BVH guided by the rays and report intersections to SMs. As a result, P_1 hits the boundary (*Case 1*) but is not within the box, so it is filtered out. P_2 is within the box and is included in the query results. For the range query, LibRTS employs a dual-pass strategy based on box-diagonal intersection tests (*Case 1*). This involves a "forward pass" that casts rays along query box diagonals against the BVH, followed by a "backward pass" that casts anti-diagonal rays to capture symmetric overlapping conditions. As shown in Figure 41(d), if boxes R_1 and S_1 intersect, one box's diagonal or anti-diagonal must intersect the other box. Simulating these (anti-)diagonals with rays allows the fixed-function RT cores to handle the computationally intensive tree traversal, significantly outperforming software-based indexing on CUDA cores.

While LibRTS optimizes the filter stage, RayJoin leverages RT cores to accelerate the computationally demanding refinement stage, specifically targeting Line Segment Intersection (LSI) and Point-in-Polygon (PIP) queries. RayJoin repurposes the RT rendering pipeline to map spatial joins to segment-segment intersections. For LSI, the framework utilizes the `AnyHit` shader to identify all intersecting segments. For the PIP query, RayJoin decomposes polygons into separated line segments with neighboring information, effectively casting it as a Point Location problem [60, 131]. RayJoin formulates this by casting a ray from the query point and using the `ClosestHit` shader to

identify the closest line segment to the ray origin. By determining the orientation or relationship of this closest segment, RayJoin efficiently leverages the hardware's native intersection searching capabilities to accelerate the query.

5.3 *RayBooster* Overview

RayBooster is essentially an alternative physical plan implementation for SedonaDB, which employs a three-layer architecture: the *Storage Layer*, the *Indexing Layer*, and the *Refinement Layer*, as illustrated in Figure 42. The workflow begins with a CPU-based `ParallelWKBLoader` that parses WKBs in Arrow arrays from a spatial join. The Storage Layer transforms these into a GPU-optimized representation (§5.4.1) while dynamically classifying queries as either point-only or range queries based on their geometric extents. The Indexing Layer then dispatches these queries to specialized RT shaders, which are executed by OptiX. These shaders cast geometry-specific rays that traverse the BVH tree on RT cores to identify spatially intersecting candidate pairs (§5.5).

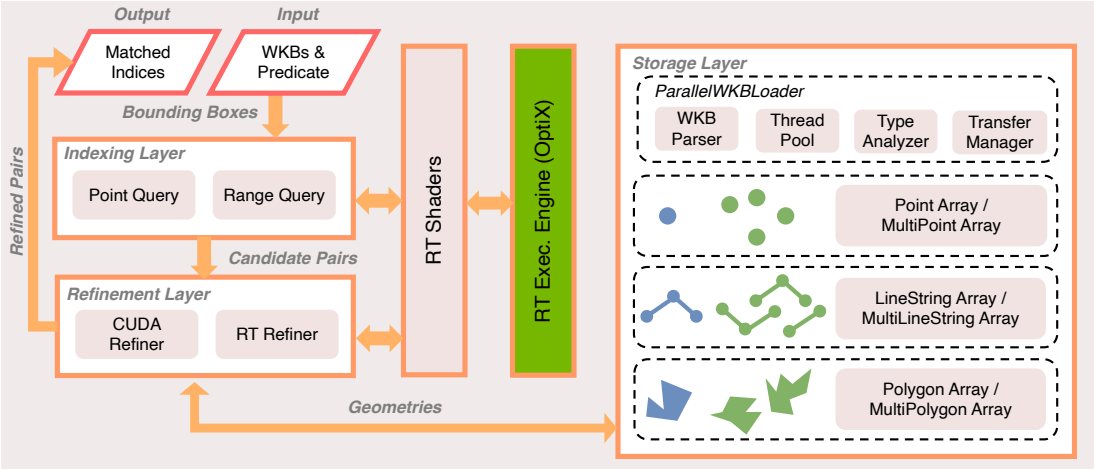


Figure 42. Architecture of *RayBooster*. Both filtering and refinement are offloaded to RT cores.

Because MBR intersections do not guarantee satisfaction of the spatial predicate, the refinement layer is necessary to eliminate false positives. *RayBooster* adopts a dual-path refinement strategy to ensure comprehensive operator coverage. An RT-based refiner processes computationally heavy, ubiquitous operators (e.g., PIP), while a standard CUDA-based refiner acts as a fallback for simpler or less common predicates. This design provides immediate system completeness for SedonaDB while facilitating the incremental transition of all operators to the RT pipeline. The final output—the IDs of verified, intersecting geometries—is then passed to the upper execution engines. We detail the refinement stage in §5.6.

5.4 GPU-friendly Storage Format

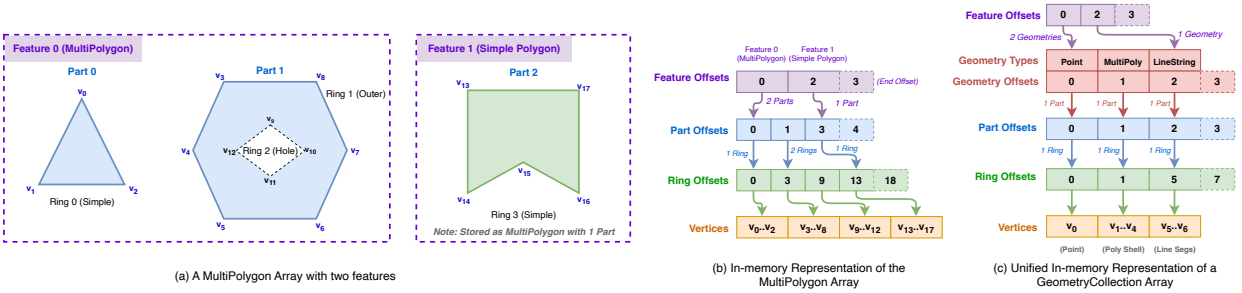


Figure 43. (a-b) shows a `MultiPolygon` array and its in-memory storage format; (c) shows a unified `GeometryCollection` array that can contain any type of geometries

5.4.1 Storage Format

Geometry data is inherently hierarchical. As shown in Figure 43(a), a `MultiPolygon` array contains multiple features[15], each comprising various `Polygon` parts with an exterior ring and optional interior rings (holes). To store these in contiguous memory space, we use hierarchical

[15] A feature refers to a geometry object in a database record, such as a `Point` or `MultiPolygon`.

offset arrays to navigate from features to parts, rings, and vertices. Vertices are stored in an ordered `double2` array to exploit vectorized LD/ST instructions. Figure 43(b) illustrates this using Feature, Part, and Ring Offset arrays.

Algorithm 11: Retrieve Feature i from `MultiPolygon` Array

Input : Feature index i; Offset Arrays O_F, O_P, O_R; Vertex Array V
Output : Set of rings (vertices) belonging to Feature i

1 /* Step 1: Identify range of Parts for Feature i */
2 $p_{start} \leftarrow O_F[i]$, $p_{end} \leftarrow O_F[i+1]$;
3 /* Step 2: Iterate over all Parts */
4 **for** $j \leftarrow p_{start}$ **to** $p_{end} - 1$ **do**
5 /* Identify range of Rings for Part j */
6 $r_{start} \leftarrow O_P[j]$, $r_{end} \leftarrow O_P[j+1]$;
7 /* Step 3: Iterate over all Rings */
8 **for** $k \leftarrow r_{start}$ **to** $r_{end} - 1$ **do**
9 /* Identify range of Vertices for Ring k */
10 $v_{start} \leftarrow O_R[k]$, $v_{end} \leftarrow O_R[k+1]$;
11 /* Step 4: Access contiguous vertex data */
12 $Ring \leftarrow V[v_{start} \dots v_{end} - 1]$;
13 **Process**($Ring$);
14 **end**
15 **end**

Given a feature ID, we can directly access its parts, rings, and vertices using the offset arrays. Algorithm 11 outlines this hierarchical retrieval process, with Figure 43(b) providing concrete values. For instance, to retrieve `Feature 0`, we first consult the Feature Offset array (O_F). The range $[O_F[0], O_F[1])$ yields the start ($p_{start} = 0$) and end ($p_{end} = 2$) indices in the Part Offset array (O_P), indicating that `Feature 0` comprises two parts. We iterate through these parts (Line 3). For `Part 0`, the range $[O_P[0], O_P[1])$ in the Ring Offset array (O_R) reveals a single ring (`Ring 0`) spanning vertex indices $[0,3)$. For `Part 1`, the range $[O_P[1], O_P[2])$ identifies two rings: `Ring 1`

bounded by $[1,2)$ in O_R (yielding vertex indices $[3,9)$), and `Ring 2` bounded by $[2,3)$ in O_R with vertex indices $[9,13)$.

This design natively supports an array of `MultiPolygon` geometries, as well as mixed collections of `Polygon` and `MultiPolygon` objects (treating `Polygon` as a single-part `MultiPolygon`). Similarly, we also provided `LineString` and `MultiLineString` arrays, which require fewer offset layers. `GeometryCollection` serves as a heterogeneous container for any geometry type. Despite its complexity, we represent it using hierarchical offset arrays. As depicted in Figure 43(c), this format introduces an additional layer tracking the specific geometry type and offsets for each part.

5.4.2 High-performance WKB Loader

Converting WKB arrays into our GPU-friendly format poses a significant bottleneck due to the massive volume and irregular layout of real-world spatial data. A single WKB array might mix simple geometries with complex anomalies containing millions of vertices. Given this severe irregularity, *RayBooster* delegates this conversion to the CPU via the `ParallelWKBLoader`, leveraging the CPU's efficiency in handling complex, branch-heavy parsing. During loading, we employ a two-pass scanning strategy to collect metadata, further improving overall efficiency and resource utilization.

In the first pass, a `TypeAnalyzer` scans the data to identify unique geometry types and infers the least generic type capable of representing the entire array (analogous to type upcasting), thereby minimizing representation overhead. The capacities of offset containers are also approximated using the number of bytes of WKBs. With this information, we can pre-allocate offset arrays without suffering from resizing overheads. Since we only read metadata of WKBs in this pass, it is very lightweight and fast.

The second pass is where the data conversion and loading happen. The WKB array is partitioned into chunks and distributed across a `ThreadPool` to eliminate thread-creation overhead during repeated spatial joins. Each thread will parse the chunked WKBs and count the number of parts, rings, and vertices. Then, a GPU-based `inclusive_scan` is used to create the offset arrays.

To mitigate straggler effects from irregular data, a load-aware `TransferManager` handles chunk sizing. Rather than splitting by element count, it calculates a byte-size prefix sum array and uses binary search to find optimal split points, ensuring threads process equal byte volumes. To prevent OOM errors during data loading, it dynamically sizes chunks based on available memory space.

5.5 Filtering with RT Cores

The goal of the filter stage in spatial joins is to identify candidate pairs whose MBRs overlap. While conventional filtering relies on software-based spatial indices such as R-trees, *RayBooster* achieves high throughput by leveraging a hardware-accelerated BVH. In a typical two-way spatial join, the smaller table is designated the *build side*, while the larger one is designated the *probe side* (§5.2.1). Depending on whether the geometry has a spatial extent (e.g., `Polygon` vs `Point`), *RayBooster* employs different techniques for index construction and querying.

Figure 44 shows the entire workflow of the filtering pipeline. The process begins by constructing a BVH over the build side's MBRs (Step 1). Subsequently, the probe side issues parallel queries against the BVH (Step 2). These queries are dynamically classified as either point queries (§5.5.2) or range queries (§5.5.3) based on their geometric properties. Probe geometries without spatial extent trigger point queries, whereas those with extent trigger range queries (§5.2.3). The resulting intersecting pairs are immediately flushed to an `IntersectionBuffer` (Step 3), implemented as a concurrent, append-only queue. We elaborate on the details below.

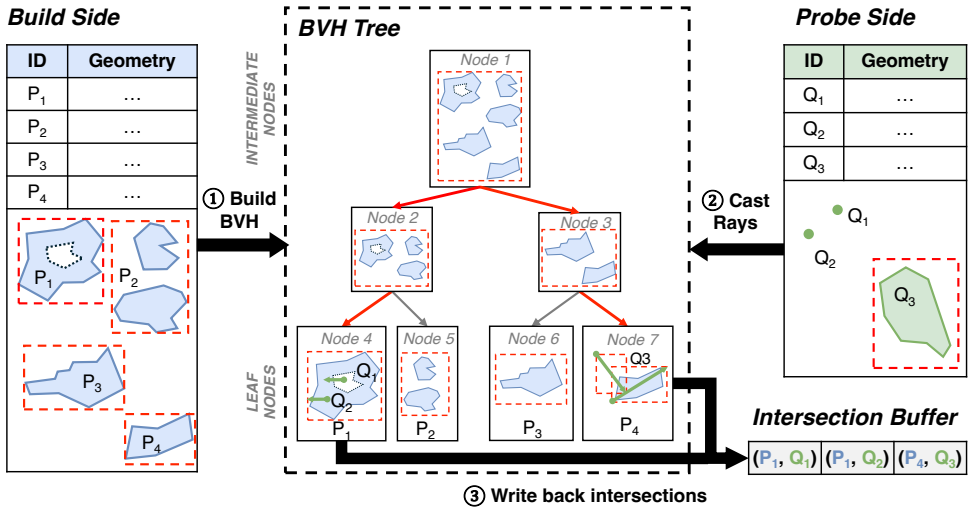


Figure 44. Filter stage of spatial join by ray-tracing

5.5.1 Build the BVH

Indexing MBRs. For geometries with a spatial extent, we compute a tightly enclosing 2-D MBR to serve as a computationally inexpensive approximation for the intersection tests. Each MBR is mapped to a 3-D OptiX AABB by setting the z-axis to zero. This array of AABBs is then fed to the NVIDIA OptiX engine to construct the hardware-accelerated BVH[16]. OptiX exposes several build configurations, allowing developers to trade construction cost for traversal performance or memory compaction. By default, *RayBooster* is tuned for maximum query performance and minimal memory footprint, incurring only a minor compaction overhead. However, users retain the flexibility to override these defaults via a `SET` statement prior to executing their SQL queries.

Indexing Points. Because the OptiX does not natively support indexing points, we must circumvent this limitation to index zero-extent points. We use an AABB to enclose multiple spatial proximal points while curtailing dead space to reduce false positives. *RayBooster* first spatially clusters the points using a space-filling curve. An AABB is then generated to tightly enclose a

[16]The BVH construction is executed on CUDA cores, whereas the subsequent traversal is accelerated by RT cores.

cluster of spatially proximal points. This strategic grouping drastically reduces the overall index size and improves query performance.

Addressing Precision Limitations. A critical systems challenge arises from the precision limitation of ray-tracing frameworks. Operations in OptiX are strictly limited to FP32 precision [89], whereas robust geospatial databases mandate FP64 precision. Relying solely on FP32 can lead to false negatives and missed intersections. To address this issue, *RayBooster* adopts the "Conservative Representation" technique introduced in *RayJoin* [64]. This technique creates a just-big-enough AABB in FP32 that encloses the MBR in FP64, guaranteeing no loss of valid candidate pairs.

5.5.2 Point Query

RT-based point querying relies on the ray-box intersection test, where each query point is modeled as a short ray (§5.2.3). Upon constructing the BVH, we cast short rays from the probe points. Each intersected AABB will be returned to SMs by RT cores. We subsequently retrieve their original MBRs with the AABB IDs, and perform an FP64 intersection test against the query point. Figure 44 demonstrates the above process, where Q_1 and Q_2 represent query points. We cast rays from the points as origins against the build-side BVH. The hardware traversal begins at the root node (Node 1), navigates through the internal node (Node 2), and terminates at the leaf node (Node 4), successfully yielding the candidate intersections (P_1, Q_1) and (P_1, Q_2). The BVH traversal is conducted on the RT cores and the intersections can be collected with the `AnyHit` shader.

5.5.3 Range Query

Range querying requires a more complex, two-pass diagonal-MBR intersection test (§5.2.3). For MBRs on the probe side, we first cast rays along their diagonals against the build-side BVH to identify intersecting AABBs. Then, a symmetric second pass is executed: rays are cast along the anti-diagonals of the build-side AABBs against a BVH for the probe-side. As demonstrated in [65],

this dual-pass ray casting guarantees the identification of all overlapping MBRs without duplication or omission. Q_3 in Figure 44 shows the execution of a range query. This query casts rays along the MBR diagonal of polygon P_4. A ray along the anti-diagonal of P_4 is also cast, but it does not intersect $Q3$. Finally, the identified intersection (P_4, Q_3) is appended to the intersection buffer.

5.6 Refinement with RT Cores

5.6.1 Solving Combinatorial Complexity

In a spatial join, the left and right sides can consist of any of the seven standard geometry types. When combined with over a dozen spatial predicates (e.g., `Contains`, `Touches`, `Crosses`), the number of unique refinement implementations approaches 500. Hardcoding each combination is tedious, error-prone, and hostile to system maintainability. To ensure `RelateEngine` remains a robust, maintainable component, we systematically reduce this complexity across both predicate types and geometry types.

Predicate Reduction. We map the evaluation of spatial predicates to one computation: computing the DE-9IM model. This matrix-based approach provides a systematic way to resolve the combinatorial explosion of topological relationships, especially when dealing with complex or multi-part geometries [39, 191]. This model is described by a 3×3 matrix that represents the intersection of the interior (I), boundary (B), and exterior (E) of two geometries. The matrix indicates the dimension of the intersection: F (no intersection), 0 (point), 1 (line), or 2 (area). Figure 45 shows the matrices for two Point-Polygon queries and one `Polygon-Polygon` query. After computing the DE-9IM model, a spatial predicate is evaluated via a logical AND operation between the predicate's mask matrix and the computed matrix. Figure 45(c) shows this for the `ST_Intersect` predicate. With the mask, the predicate evaluates to true if the 2×2 top-left elements are not all F.

Geometry Type Reduction. To further reduce combinatorial complexity, we represent the singular WKB geometry types using their `Multi` variants; e.g., a simple `Polygon` is internally treated as a `MultiPolygon` with exactly one part. This aligns with formal spatial data models that generalize topological operations over multi-component spatial objects rather than treating simple geometries as isolated special cases [191]. Furthermore, we exploit the symmetric properties of the DE-9IM model. For example, the refinement logic for a `MultiPolygon-MultiPoint` join can be equivalently derived from the `MultiPoint-MultiPolygon` implementation by swapping the join sides and transposing the matrix.

5.6.2 Calculating DE-9IM Matrix with RT cores

Calculating the DE-9IM matrix is amenable to RT core acceleration. By simulating geometric vertices and line segments as strategically cast rays, RT hardware can evaluate all necessary interior, boundary, and exterior intersections.

Build BVH. For a given pair of candidate geometries, we construct an index at line-segment level for the more complex geometry (e.g., having more vertices), designating it the *indexed side*, while the other acts as the *query side*[17]. We employ an AABB to enclose adjacent line segments [18] on the indexed side. Rays are then cast based on the specific geometry type of the query side. We demonstrate this below for Point-Polygon and Polygon-Polygon relations.

Example 1: Point-Polygon Evaluation. Figure 45(a) illustrates the evaluation of a `Point-Polygon` relationship. For query points, we cast an infinite horizontal ray. When the RT core detects a ray-AABB hit, we visit the enclosed line segments to perform an exact intersection test. *RayBooster* utilizes the ray-crossing (even-odd) algorithm to determine if a point lies within a

[17]Note: These designations are localized to the refinement stage and are distinct from the term of *build* and *probe* sides used in the earlier filter stage.

[18]By default, a group of 32 adjacent line segments is enclosed by an AABB to fully exploit warp-parallelism for computing the AABB corners.

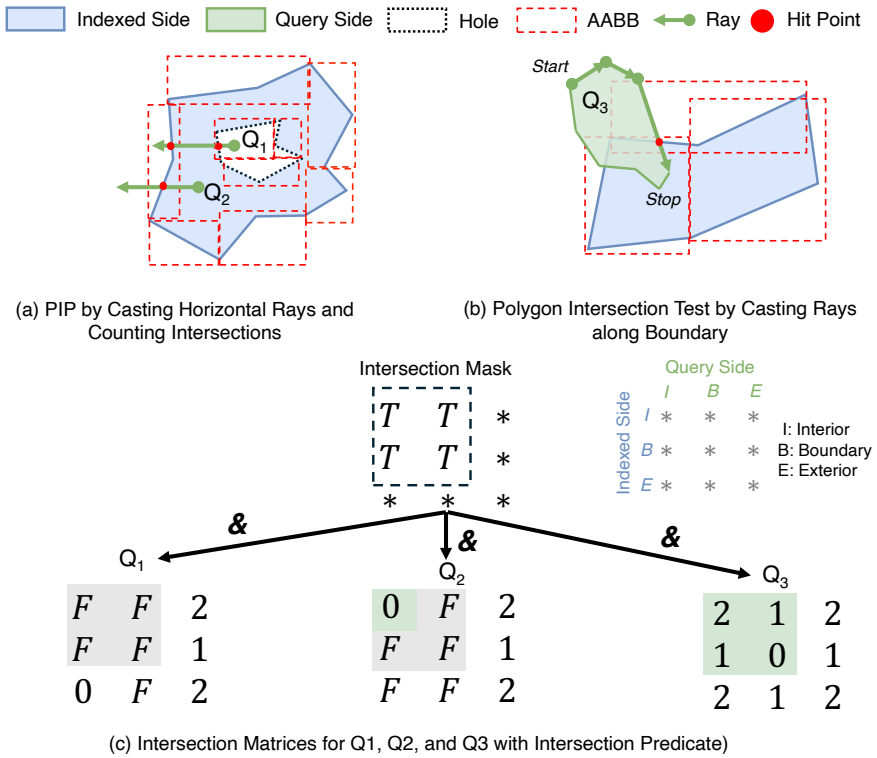


Figure 45. Demonstration of casting rays to find geometry element intersections and compute the DE-9IM model.

ring [203]. In the figure, rays from both Q_1 and Q_2 intersect the exterior ring exactly once (an odd number), indicating they are inside the outer boundary. However, robust refinement requires verifying holes. We need to recast the ray for Q_1 to determine whether the ray intersects the polygon's interior. Because Q_1 is found to reside within a hole, it is correctly classified as being outside the polygon's interior. As shown in Figure 45(c), the resulting interior-interior cell for Q_1 is labeled F. Conversely, Q_2 does not intersect any holes. Its interior-interior intersection is a point, so the value is 0. In `RelateEngine`, the degenerated cases, like a point-on-boundary, are also checked and mapped to the corresponding DE-9IM model.

Example 2: Polygon-Polygon Evaluation. Polygon intersection testing is decomposed into boundary intersection tests, and each line segment can be simulated as a ray. As shown in Figure

45(b), assuming the top-left line segment is the first segment of the query polygon's exterior ring, we cast rays along that segment's trajectory. If no intersection occurs, we proceed to the next segment until a hit is found. In this example, the interior-interior intersection forms an area (matrix value 2), the interior-boundary intersections form line strings (matrix value 1), and the boundary-boundary intersection results in points (matrix value 0).

In practice, we do not need to calculate every DE-9IM matrix element, `RelateEngine` evaluates core topological relationships (e.g., interior-interior) and derives the rest using dimensional invariants. For instance, given a point and a polygon, the $E \cap E$ is invariably 2D, and $E \cap B$ is 1D. The engine only resolves the point's location: if inside, $I \cap I = 0D$; if on the edge, $I \cap B = 0D$. By populating the matrix with these deduced states, our unified engine achieves the same time complexity as dedicated, hardcoded predicates (e.g., `ST_Intersects`). Thus, the overhead of completing the full DE-9IM matrix via constant-time logical derivations is negligible.

5.6.3 Monolithic BVH by Z-Stacking

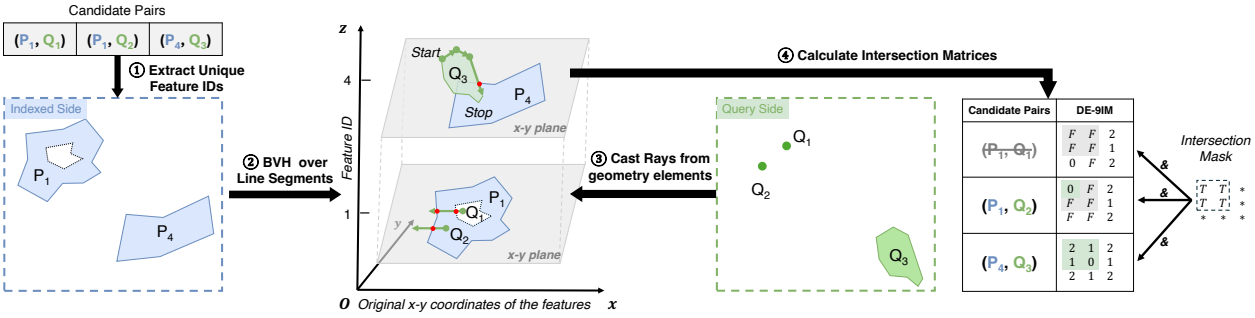


Figure 46. Execution workflow of RT-based refinement with Z-stacking technique. (a): Polygons on the indexed side; (b): Intersection Scene of the indexed polygons and queries; (c): Two query points and a query polygon; (d): Refinement results by applying a mask matrix to the intersection matrices.

Calculating the intersection matrix via RT cores requires indexing each individual geometry, meaning a spatial join with millions of features would require building millions of distinct BVHs. This per-geometry indexing approach is fatally bottlenecked by GPU synchronizations and massive BVH construction overheads. To solve this, we introduce a novel *monolithic BVH* design achieved through "Z-Stacking." Because hardware ray-tracing frameworks (like OptiX) are natively designed for 3-D space, and geospatial queries are confined to 2-D plane space, the Z-axis is entirely unused. We can repurpose this spare Z-axis to encode feature IDs [19] and build a monolithic BVH for all the features. We place the 2-D coordinates of each geometry feature onto a distinct X-Y plane, where the Z-coordinate corresponds to the feature's unique ID. This allows us to construct a single BVH for the entire batch of geometries.

Figure 46 illustrates this Z-Stacking technique in action. First, we extract the unique feature IDs on the indexed side from the candidate pairs. For example, polygon P_1 is mapped to the X-Y plane at $Z = 1$, while polygon P_4 is mapped to the plane at $Z = 4$ (Figure 46(b)). Subsequently, the query-side geometries cast rays exclusively along their corresponding X-Y planes. As shown in Figure 46(c), Q_1 and Q_2 are constrained to $Z = 1$, while Q_3 is placed on the plane with $Z = 4$. Q_1 and Q_2 cast horizontal rays to count intersections with P_1. As evaluated previously, Q_1 falls within a hole (yielding no interior intersection), while Q_2 falls within the valid polygon interior (yielding a 0-dimensional point intersection). The resulting matrices are shown in Figure 46(d).

5.6.4 CUDA-based Implementations as Fallback

While mapping refinement operators to ray-tracing yields immense performance benefits, the engineering effort to translate every query remains substantial. To function immediately as a drop-in GPU engine, *RayBooster*'s `RelateEngine` provides standard CUDA fallbacks for all operators,

[19] To avoid FP32 precision limitations on the Z-axis, the original feature IDs are renumbered into a compact, zero-based contiguous sequence during BVH construction. This prevents integer overflow and guarantees exact ID retrieval without precision loss.

while offering optimized, RT-accelerated paths for demanding Point-Polygon (and `Multi`) PIP queries. This dual-path architecture ensures immediate usability while lower-priority operators are incrementally ported.

5.7 Memory Management

5.7.1 Host Memory Management

Spatial joins require space for intermediate data structures, such as a spatial index and the geometries produced by the evaluated SQL expression. *RayBooster* inherits the robust execution framework of SedonaDB, which ensures the completion of execution under a given memory constraint while trying its best to maintain high performance. SedonaDB may partition the build and probe side using a KDB-Tree [183] to create balanced partitions, which are then spilled to disk as Arrow IPC files with low I/O overhead. SedonaDB uses DataFusion's `MemoryPool` mechanism to request memory allocation, and *RayBooster* inherits this mechanism to operate. When the spatial join's memory footprint reaches the pool's configured threshold, further allocations are denied, and SedonaDB will trigger the disk-spilling mechanism. *RayBooster* benefits this spilling workflow by replacing the default CPU-based spatial index with an RT-accelerated spatial index. When a spilled partition is loaded on demand, the "probe side" data streams through that partition, and it queries this RT-accelerated index to find matching geometries.

5.7.2 Device Memory Management

Reducing Allocation Overhead. Spatial joins require device memory allocations for geometries, BVHs, and worklists. Standard allocation calls (e.g., `cudaMalloc` and `cudaFree`) are expensive and cause host-device synchronization stalls [207]. *RayBooster* uses the RAPIDS Memory Manager (RMM) to solve this issue [181]. We use RMM's coalescing, best-fit pool sub-allocator.

All on-device data structures are wrapped in RMM's `device_uvector` and `device_buffer` containers to use RMM's memory allocator. This approach bypasses `cudaMalloc`'s synchronization overhead and leverages the CUDA memory pool APIs for fast memory allocation.

Handling Unpredictable Buffer Sizes. Memory consumption during filtering is highly unpredictable because the exact number of candidate pairs is unknown. Gauging a safe buffer size over-allocates memory, therefore wasting space. To address this issue, *RayBooster* adopts a two-pass execution model, including an initial counting pass followed by a materializing pass. In the counting pass, we cast rays and count the number of intersections, which is used to allocate the exact required memory space. In the second pass, we cast the rays again and store the results in the allocated buffer. Since the RT-accelerated index is fast, this two-pass approach incurs negligible overhead and avoids memory allocation waste, supporting large queries with the very limited device memory.

Adaptive Chunking. Building the BVH in a single batch may exhaust device memory. We use adaptive chunking to determine a safe chunk size based on free memory and the BVH size. Since *RayBooster* operates on a pre-allocated memory pool by RMM, `cudaMemGetInfo` cannot accurately measure free memory, so we use RMM's `tracking_resource_adaptor` to compute the unused space in the memory pool, along with unclaimed device memory.

Under Unified Memory Architecture. The newly introduced NVIDIA DGX Spark [162] may simplify memory management. This paradigm renders device allocators (RMM) unnecessary, as RT cores natively access regular pointers without explicit synchronizations. Hardware-managed page-faulting and large shared memory pools supersede complex software workarounds for unpredictable buffer sizing. Furthermore, out-of-core disk spilling transitions from a device constraint to a system-wide limit. This architectural shift abstracts low-level memory orchestration, allowing *RayBooster* to efficiently process irregular spatial workloads while simplifying software design. Optimizing for DGX Spark remains future work.

5.8 Evaluation

5.8.1 Settings

Baselines. As this chapter focuses on single-node OLAP systems, we exclude distributed frameworks like Apache Sedona. Since SedonaDB's advantages over DuckDB and GeoPandas are established in §5.2.1, we omit them. Our primary comparison evaluates SedonaDB with and without RT-acceleration by *RayBooster*. To the best of our knowledge, PG-Strom [91] is the only open-source spatial database that offers GPU acceleration by offloading spatial indexing and refinement to the GPU, but it relies on CUDA rather than RT cores.

Methodology. Historically, the field lacked a dedicated benchmark for spatial databases until the introduction of SpatialBench (§5.8.2), a sub-project under Apache Sedona. It provides a standardized, scalable, and open-source framework tailored for geospatial analytics. Its unbiased methodology ensures reproducible performance comparisons across diverse runtimes and environments.

Metrics. We mainly use running time as the metric for performance evaluation. We conduct our experiments with one warmup iteration followed by five additional executions, reporting the average end-to-end SQL execution time excluding the warmup round. A roofline model could be helpful for analyzing GPU utilization [30]. Unfortunately, the current profilers do not expose any performance counters of RT cores due to their proprietary nature [151].

Hardware. Detailed in Table 8, evaluations are conducted across a variety of GPU architectures in a workstation, cloud, and the Ohio Supercomputing Center (OSC). Notably, while the NVIDIA A100 and H100 do not have dedicated RT cores, *RayBooster* remains functional on these GPUs via software-emulated BVH traversal provided by OptiX [161]. With no additional maintenance efforts, we can assess performance in both hardware-accelerated and emulated environments. Without explicitly specifying, we always evaluate SedonaDB on an AWS m7i.2xlarge instance.

Environment and Configuration. We install PGStrom v6.1 extension on PostgreSQL 15. To have a fair comparison [178], we set the PGStrom configurations to the values recommended on its official website and enable indexes on geometry columns. *RayBooster* is a library written in CUDA/C/C++, and it provides a C interface for interacting with SedonaDB via the Foreign Function Interface (FFI). We use Rust 1.90.0, CUDA 12, OptiX 8, and RAPIDS 25.12.00. The only configuration we tune for *RayBooster* is the DataFusion batch size, which is set to 100K for Q2-Q9 and 2M for Q10 and Q11.

Table 8. Hardware specifications and pricing.

GPU	RT Cores	FP64 Perf.	VRAM (Bandwidth)	CPU (RAM)	Platform / Unit Price
RTX 3090	2nd Gen (82)	556.0 GFLOPS	24GB (936 GB/s)	i9-12900 (64GB)	Workstation –
A10	2nd Gen (72)	976.3 GFLOPS	24GB (600 GB/s)	8 vCPUs (32GB)	AWS g5.2xl $1.21/hr
L4	3rd Gen (60)	473.3 GFLOPS	24GB (300 GB/s)	8 vCPUs (32GB)	AWS g6.2xl $0.98/hr
L40S	3rd Gen (142)	1.43 TFLOPS	48GB (864 GB/s)	8 vCPUs (64GB)	AWS g6e.2xl $2.24/hr
A100	SW Emul.	9.75 TFLOPS	40GB (1.56 TB/s)	EPYC 7H12 (472GB)	OSC –
H100	SW Emul.	33.45 TFLOPS	94GB (3.36 TB/s)	Xeon 8470 (1TB)	OSC –

5.8.2 SpatialBench

To evaluate *RayBooster* under realistic industrial conditions, we utilize SpatialBench [9], a modern open-source benchmark suite designed for geospatial SQL. While standard OLAP benchmarks like TPC-H [213] are industry norms, they lack native geometry types and fail to adequately evaluate

spatial bottlenecks from complex predicates and spatial joins. SpatialBench bridges this gap by combining a systematic benchmarking methodology with large-scale urban mobility datasets, such as the NYC Taxi and Limousine Commission data [153]. The benchmark queries are derived from real-world WherobotsDB optimization challenges, providing practical, representative coverage of spatial operations.

Data Model and Generation The benchmark employs an extended star schema consisting of a Trip fact table (representing rideshare trajectories with spatial pickup and dropoff coordinates) and multiple dimension tables, such as Zone (hierarchical administrative boundaries) and Building (polygon footprints). To reflect the natural skew and density variations [97] of real-world geographic data, the synthetic data generator restricts geometries to continent-bounded polygons and distributes coordinates using a Hierarchical Thomas process [210]. This multi-level clustering algorithm realistically mimics multi-scale urban settlement patterns, producing heavy-tailed distributions of dense city centers surrounded by sparse rural outliers. This pattern places heavy demands on spatial indexing and partitioning strategies. We evaluate the systems on Scale Factors (SF) 1 (1.8GB) and 10 (7.0GB), omitting SF100 (>56GB) as it exceeds typical single-node capacities.

Query Workload The SpatialBench workload comprises 12 analytical queries (Q1–Q12) that evaluate distinct facets of a spatial query engine. Since this chapter focuses on spatial relational joins only (§5.1), we use a subset of SpatialBench, as listed in Table 9.

5.8.2.1 Query Characteristics

Queries Q2, Q4, Q6, and Q9 represent small, latency-sensitive spatial joins, as one of the input relations contains very few rows after filtering. In these scenarios, the computational workload is insufficient to saturate the GPU. Consequently, the primary objective is to ensure that offloading to the GPU does not introduce performance regressions (e.g., due to data transfer overhead over

Table 9. Relational Spatial Joins from SedonaBench Query Workload. *: Refinement stage in Q9 fallbacks to CUDA.

Query	Description	Spatial Characteristics	Scenario
Q2	Count trips starting within Coconino County (Arizona) zone	• Point-in-polygon spatial filtering (ST_Intersects) • Subquery with spatial geometry selection • Simple aggregation on spatially filtered data	Count all trips originating within a specific administrative boundary (county) for regional transportation statistics.
Q4	Zone distribution of top 1000 trips by tip amount	• Subquery with ordering and limiting; Aggregation on spatially joined results • Point-in-polygon spatial join (ST_Within) • Multi-step query; spatial filtering, and grouping	Analyze the geographic distribution of high-value trips (by tip amount) to understand premium service areas.
Q6	Zone statistics for trips within a 50km radius of Sedona city center	• Polygon containment check (ST_Contains) • Point-in-polygon spatial join (ST_Within)	Analyze trip patterns in zones within a metropolitan area around a city center.
Q9*	Building Conflation (duplicate/overlap detection via Intersection over Union (IoU))	• Self-join with spatial intersection (ST_Intersects); Area calculations (ST_Area) • Geometric intersection operations; Complex geometric ratio calculations	Detect duplicate or overlapping building footprints in GIS datasets to identify data quality issues.
Q10	Zone statistics for trips starting within each zone	• Point-in-polygon spatial join (ST_Within) • Aggregation with multiple metrics • LEFT JOIN on zones with no trips	Analyze trip patterns and performance metrics for each administrative zone.
Q11	Count trips that cross between different zones	• Multiple point-in-polygon spatial joins • Filtering based on spatial relationship results	Identify inter-district/city trips to analyze cross-boundary travels.

PCIe) [119, 122, 241]. Queries Q10 and Q11 are heavy joins that trigger a large volume of point-in-polygon operations.

5.8.3 Overall Performance

Table 10 summarizes the query results, including $SF = 1$ and $SF = 10$. We evaluated PGStrom on a g5.2xlarge instance, a standard configuration for GPU-based tasks (e.g., inference) at Wherobots. Since PGStrom is an extension for the PostGIS OLTP database, an exact apples-to-apples comparison is difficult. Therefore, we followed the SpatialBench methodology by explicitly listing

Table 10. Average execution times of SpatialBench queries in seconds with standard deviations. "PGStrom Build+Exec." represents the average SQL execution times plus the index build-up time on geometry columns used in the corresponding query.

Query	SF	PGStrom Indexed	PGStrom Build + Exec.	PGStrom No Index	SedonaDB	*RayBooster* on Various GPUs					
						RTX3090	L4	L40S	A10	A100	H100
Q2	1	0.17 ± 0.00	16.69	0.74 ± 0.09	0.33 ± 0.02	0.29 ± 0.01	0.35 ± 0.01	0.35 ± 0.02	0.42 ± 0.01	0.42 ± 0.04	**0.27 ± 0.01**
	10	0.44 ± 0.01	182.63	6.42 ± 0.05	1.50 ± 0.04	1.07 ± 0.02	1.56 ± 0.02	1.59 ± 0.03	1.91 ± 0.03	1.02 ± 0.05	**0.78 ± 0.00**
Q4	1	0.73 ± 0.00	1.51	35.19 ± 0.11	0.39 ± 0.01	0.30 ± 0.01	0.39 ± 0.02	0.39 ± 0.01	0.56 ± 0.20	0.32 ± 0.01	**0.22 ± 0.00**
	10	6.04 ± 0.01	7.97	105.20 ± 0.34	0.98 ± 0.06	0.74 ± 0.04	1.00 ± 0.01	0.97 ± 0.02	1.33 ± 0.14	0.58 ± 0.08	**0.32 ± 0.02**
Q6	1	0.01 ± 0.00	17.31	5.67 ± 0.03	0.59 ± 0.01	0.44 ± 0.01	0.62 ± 0.02	0.65 ± 0.03	0.89 ± 0.19	0.59 ± 0.02	**0.39 ± 0.01**
	10	0.04 ± 0.00	184.16	113.65 ± 0.32	2.21 ± 0.02	1.54 ± 0.11	2.35 ± 0.07	2.26 ± 0.06	2.89 ± 0.11	1.55 ± 0.18	**0.94 ± 0.02**
Q9	1	0.43 ± 0.00	0.48	47.10 ± 0.13	**0.02 ± 0.00**	0.04 ± 0.02	0.03 ± 0.00	0.03 ± 0.00	0.03 ± 0.00	0.04 ± 0.00	0.03 ± 0.00
	10	1.13 ± 0.00	1.32	610.38 ± 1.19	**0.07 ± 0.00**	0.08 ± 0.01	0.10 ± 0.00	0.10 ± 0.00	0.12 ± 0.00	0.13 ± 0.00	0.10 ± 0.00
Q10	1	23.25 ± 0.53	39.77	Timeout	4.96 ± 0.09	**1.26 ± 0.08**	1.53 ± 0.04	1.58 ± 0.14	2.40 ± 0.70	2.37 ± 0.23	1.77 ± 0.14
	10	112.41 ± 6.46	294.60	Timeout	33.21 ± 0.27	4.03 ± 0.08	5.60 ± 0.17	4.99 ± 0.17	6.95 ± 0.37	5.55 ± 0.41	**3.43 ± 0.19**
Q11	1	246.24 ± 2.00	247.02	Timeout	7.51 ± 0.12	**1.61 ± 0.08**	2.01 ± 0.04	1.93 ± 0.03	2.57 ± 0.10	2.07 ± 0.05	1.66 ± 0.04
	10	Timeout	-	Timeout	53.34 ± 0.39	6.91 ± 0.35	10.85 ± 1.20	8.55 ± 0.05	13.52 ± 0.82	7.91 ± 0.18	**5.65 ± 0.11**

both queries without indexing, indexed queries, and index-building times, providing readers with a transparent view of overall performance.

5.8.3.1 Queries with Latency Sensitivity Joins

As listed in Table 9, queries Q2–Q9 represent latency-sensitive joins. Considering execution time only, PGStrom demonstrates low latencies due to prebuilt indexes (e.g., Q6 is under 50ms). However, factoring in index construction time, which can be an order of magnitude longer, erases this performance edge. Figure 47 shows the execution time breakdown of PGStrom[20]. We can see that for Q2 and Q4, most of the time is spent on scanning operations, and the spatial-related functions are not a bottleneck, while Q9 is bottlenecked by the filter stage.

SedonaDB overperforms PGStrom due to its in-memory design, despite building indexes at runtime. When enabling *RayBooster*, execution times are comparable to or slightly faster than SedonaDB on the CPU across various GPU models. Interestingly, profiling reveals *RayBooster* also reduces table scanning times, a counterintuitive phenomenon explored in §5.8.5. These results align

[20]We cannot succeed on Q11 with "EXPLAIN ANALYZE" as PGStrom raises errors.

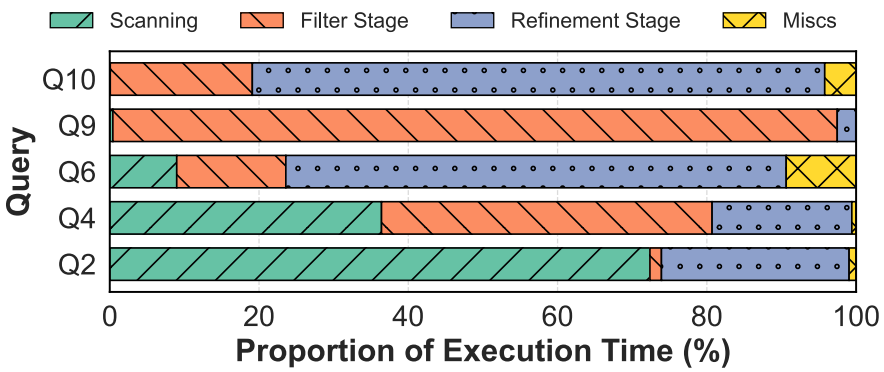


Figure 47. SQL execution breakdown (SF=1) of PGStrom Indexed. "Miscs" include sort/group/aggregate operations.

with our expectations. Since SedonaDB is not bottlenecked by these lightweight queries, enabling RayBooster successfully avoids adding latency overhead.

5.8.3.2 Queries with Heavy Joins

Queries Q10 and Q11 involve computationally heavy joins, which are the cases where *RayBooster* excels. As shown in Listing 5.1, Q11 performs a two-way spatial join that generates intensive PIP operations, which is the exact workload that *RayBooster* offers to optimize.

Listing 5.1. SQL query of Q11: calculating cross-zone trips.

```
SELECT COUNT(*) AS cross_zone_trip_cnt
FROM trip_geom t
JOIN zone_geom pickup_zone
  ON ST_Within(t.pickup_geom, pickup_zone.geom)
JOIN zone_geom dropoff_zone
  ON ST_Within(t.dropoff_geom, dropoff_zone.geom)
WHERE pickup_zone.z_zonekey != dropoff_zone.z_zonekey
```

When SF=1, SedonaDB's CPU implementation required 7.51s to execute Q11 on an m7i.2xlarge instance. Enabling *RayBooster* reduced this to approximately 2s across all evaluated GPU models. The fastest execution time for Q11 was 1.61s on an RTX3090, yielding a 4.66× speedup over the CPU baseline. Although the H100 lacks dedicated RT cores, its ultra-high memory bandwidth and

FP64 throughput allow it to marginally outperform most models equipped with RT cores, especially for large-scale datasets. However, this advantage is not significant. For instance, the consumer-grade RTX 3090 completed the query in 6.91 seconds for Q11 when $SF = 10$, while H100 took 5.65s, merely 22.3% slower than the enterprise-grade H100. For Q10, PGStrom took more than 23s for execution alone, where the majority of time was spent on the refinement stage (as shown in Figure 47), while SedonaDB's adaptive join makes it only take about 5s, which is 4.69× faster. After enabling *RayBooster*, it delivers speedups on Q10 ranging from 2.09× to 3.94× across the tested GPU models when $SF = 1$. The advantage of *RayBooster* is more obvious for $SF = 10$, achieving speedups from 4.93× to 9.68× across all the GPU models. These results confirm that *RayBooster* is an effective booster for computationally intensive queries.

5.8.4 Cost-effectiveness

Evaluating raw performance metrics in isolation is insufficient for enterprise use cases, where financial cost is a critical factor. We calculate the operational cost of each query by multiplying its execution time by the hourly instance rate. AWS is the major cloud platform at Wherobots, so we evaluate *RayBooster* with various EC2 instances. The CPU baseline (SedonaDB) runs on an m7i.2xlarge instance, priced at $0.4032 USD per hour, which is the major instance type in our company. Table 8 details the GPU instance types and their corresponding hourly rates.

Figure 48(a) illustrates the per-query cost across the CPU and GPU instances. For latency-sensitive joins (Q2-Q9), the cheapest GPU instance g6.2xlarge consistently incurs higher costs than the CPU baseline. However, for computationally heavy joins, GPU acceleration directly translates into cost savings. For example, executing Q10 (SF=10) on the CPU costs approximately $0.003720 USD, whereas the L4 GPU reduces this to $0.001524 USD, yielding a 59.02% cost reduction due to a steep 5.93× speedup. Figure 48(b) aggregates the total execution cost across all

queries. The results demonstrate that the A10 and L4 GPU models deliver overall cost savings. In contrast, the L40S instance incurs a higher cost than the CPU baseline due to its steep hourly rate. When factoring in both the extra costs for small queries and the substantial savings on heavy joins, *RayBooster* reduces the total workload cost by 38.11% when utilizing the L4 GPU.

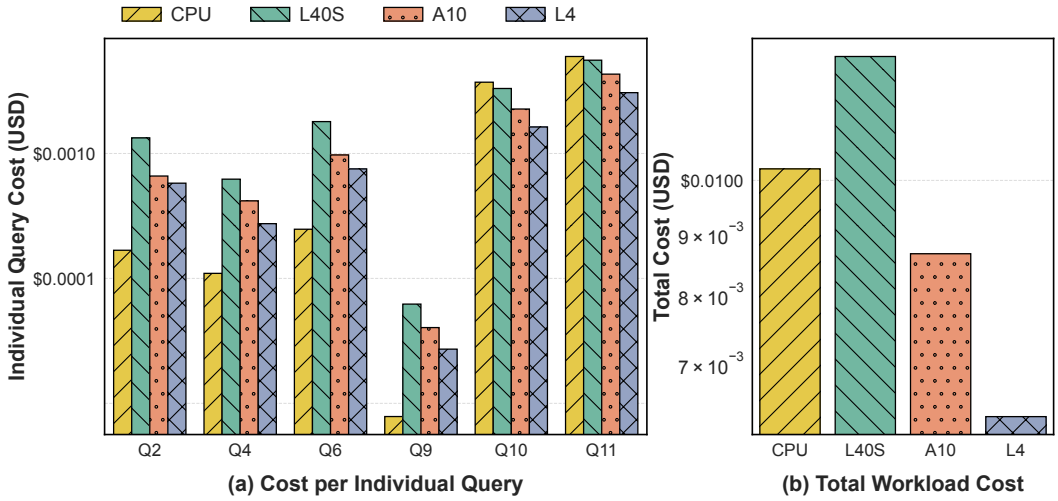


Figure 48. Cost-effectiveness analysis on different platforms with SpatialBench (SF=10); (a) log scale, (b) linear scale

5.8.5 Running Time Analysis

To understand the performance bottlenecks and analyze how *RayBooster* reduces execution time, we profiled the query execution. Figure 49 presents a breakdown of the major execution time components for a lightweight query (Q2) and a computationally heavy query (Q11) on a g6e.2xlarge instance. To observe the execution behavior and capture true wall-clock time, we set the engine to use a single partition. Without this setting, the profiler reports the aggregated CPU time across

all partitions, which exceeds the actual wall-clock time. Thus, the absolute execution times in this targeted analysis differ from the end-to-end results presented in Table 10.

The *Zone* table contains highly irregular polygon boundaries representing administrative zones. Parsing and processing these complex geometries consumes significant CPU resources in SedonaDB, extending scan times. Interestingly, although *RayBooster* targets spatial joins, upstream scan operations also execute substantially faster. This could be attributed to the offloading of computationally intensive spatial joins to RT cores, which reduces CPU contention, allowing the CPU to process scans more efficiently. Consequently, execution times for both operations are reduced. Isolating the spatial join itself, *RayBooster* achieves 4.62× and 4.07× speedups for Q2 and Q11, respectively.

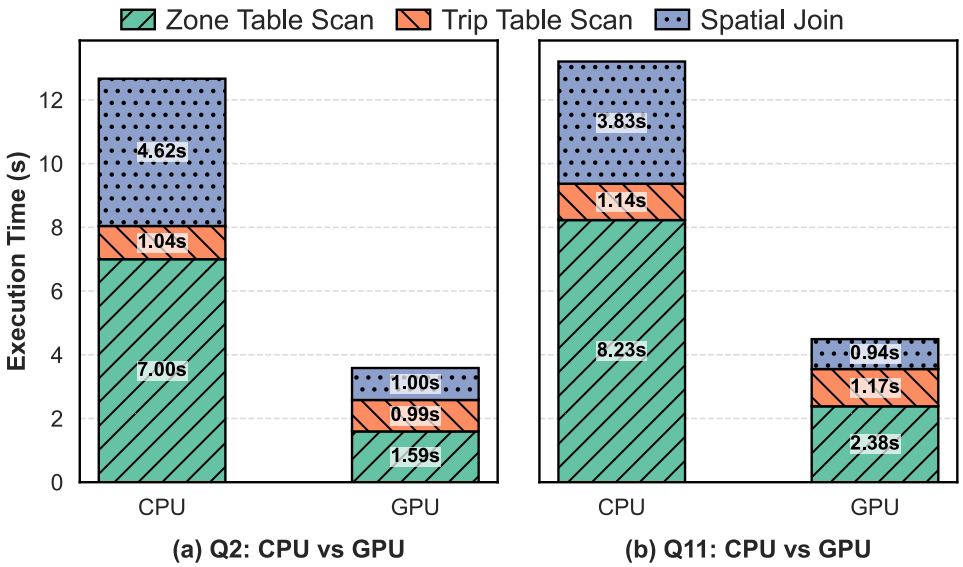


Figure 49. Comparing major time components of CPU and GPU joins (SF=1, Single Partition, g6e.2xlarge)

Figure 50 details the execution time breakdown of the spatial join operation. As shown, the refinement phase consistently dominates execution, consuming over 64% of the time across all

queries, followed by filtering and WKB conversion. While the conversion overhead is negligible for lightweight queries such as Q2, Q4, and Q6, it becomes a significant bottleneck for Q9 due to the large number of polygons processed on both sides of the join. Furthermore, index construction for the filtering phase is highly efficient, accounting for up to 4.2% of the total time across all queries.

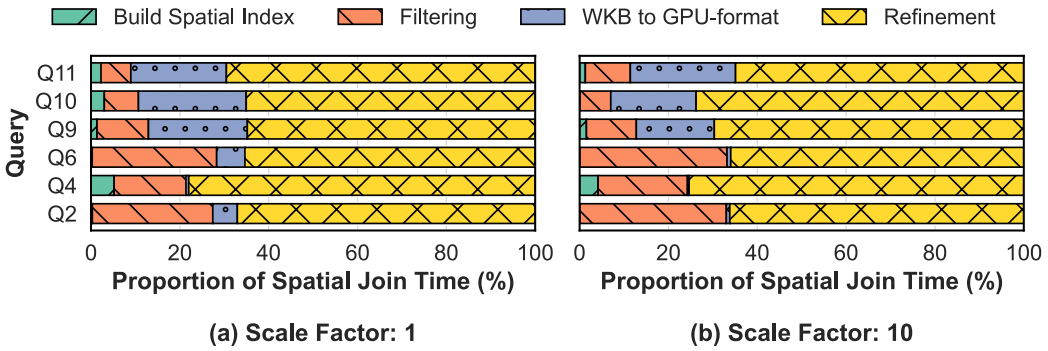


Figure 50. *RayBooster* execution time breakdown

5.8.6 Effectiveness of Memory Management

Figure 51(a) shows the performance impact of memory allocation strategies in *RayBooster*, comparing the default cudaMalloc against an asynchronous memory allocator used for dynamic structures like the BVH, IntersectionQueue, and temporary storage. The results indicate that asynchronous memory allocation is consistently beneficial, as it avoids unnecessary host-device synchronizations, yielding a 5% to 36% speedup on the SF=10 dataset. The performance gains are substantial, particularly for latency-sensitive joins, where we observe execution time improvements of 36% for Q2 and 27% for Q6. Furthermore, even for compute-bound queries such as Q10 and Q11, the asynchronous allocator provides notable speedups of 5% and 6%, respectively. Figure 51(b) illustrates *RayBooster*'s spilling behavior under memory constraints ranging from 5GB to

15GB, with Q11 memory-hungry three-table join. At a 5GB limit, heavy spilling slows execution to 88.93s, over 8× slower than pure in-memory processing (Table 10). However, performance improves almost linearly as the budget increases up to 9GB, with spilling relieved. Once the budget exceeds 11GB, it fully supports Q11, yielding near-in-memory performance. These results confirm *RayBooster*'s robustness in heavily memory-constrained environments.

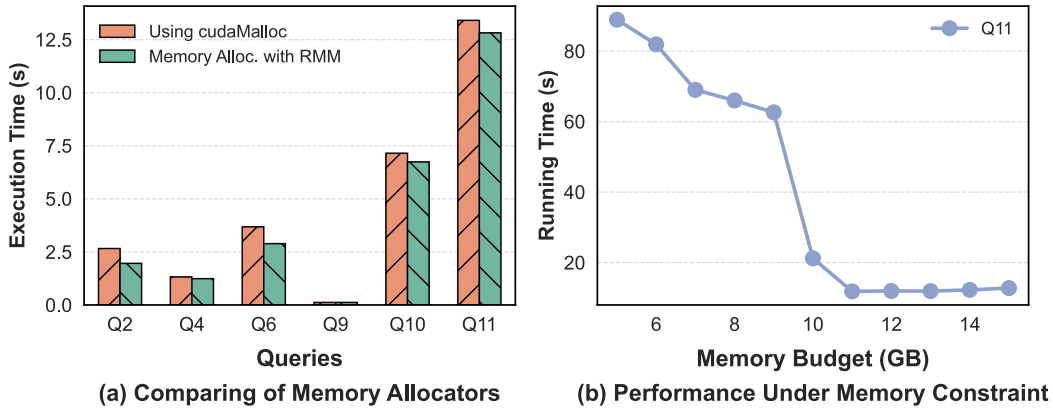


Figure 51. Effectiveness of memory optimization strategies

5.9 Related Work

Repurposing RT Cores Several recent studies have leveraged RT cores to optimize KNN searches and spatial queries. For instance, RTNN improves KNN performance through spatial reordering and query partitioning [258]. TrueKNN supports KNN searches without a predefined radius [152]. JUNO achieves high-dimensional neighbor searches on RT cores [127]. Arkade [133] adapted KNN to support non-Euclidean space. Closer to this work in the dissertation, RayJoin (Chapter 2) utilizes RT cores specifically for spatial joins, and LibRTS (Chapter 3) repurposes them as a general-purpose spatial index.

GPU Databases Over the past decade, substantial research into GPU-accelerated databases has yielded numerous prototypes and optimization techniques [28, 58, 76, 94, 115, 172, 187, 188, 193, 202, 211, 222, 229, 241, 242, 250]. Although this foundational work has led to the development of several production-grade GPU databases, native support for GPU-accelerated spatial queries remains scarce. For example, while the OLAP database HEAVY.AI [77] supports spatial operations for massive datasets, it does not offload these tasks to the GPU. RateupDB [113], a full-featured CPU/GPU HTAP system, lacks spatial querying capabilities. Kinetica [84], a distributed memory-first OLAP database, offers GPU acceleration for spatial, but does not use RT technology. Furthermore, empirical comparisons against it are precluded by its terms of service. Finally, PG-Strom [91] extends PostgreSQL with CUDA acceleration, serving as a primary baseline in our evaluation. *RayBooster* is the first production-level product that uses RT cores to accelerate spatial queries.

GPU Spatial Data Processing Despite the proliferation of GPU-accelerated spatial processing research, few solutions are production-ready. cuSpatial, a spatial processing library within NVIDIA's RAPIDS suite [160], uses an octree to index points and implements PIP operations via a CUDA-based crossing number algorithm. In contrast, our approach builds a polygon-level index and uses RT cores to accelerate PIP tests, achieving much faster performance than cuSpatial [64]. Other representative works [12, 221] include GPP, which offers a robust polygon overlay library using a uniform grid. RasterJoin [43, 244] leverages the conventional graphics rendering pipeline for spatial aggregation. While these standalone libraries advance GPU spatial processing, none provide a native SQL interface for seamless database integration.

Spatial Join Techniques The filtering stage of spatial joins can be further improved by using two spatial indices with the synchronized traversal technique [87]. However, it requires accessing the internal node of a spatial index, and the prohibition of explicit traversal with RT cores renders this technique unusable. Recently, Georgiadis et al. introduced a new technique to improve the efficiency

of spatial joins with an intermediate filtering technique that utilizes raster interval approximations to deduce topological relations before evaluating exact geometries [67, 68]. These works are directly related to the work in this chapter.

5.10 Conclusion

Hardware acceleration is essential for spatial database systems due to their inherently geometric, data-intensive, and real-time workloads. Motivated by evidence that RT cores provide powerful support for geometric computation, we introduce *RayBooster*, the first production-grade integration of RT-core acceleration into SedonaDB. Targeting spatial joins, the dominant execution cost, we designed system-level mechanisms bridging the architectural gap between spatial query processing and RT hardware. Fully integrated into SedonaDB, *RayBooster* demonstrates up to a 5.93× performance improvement on SpatialBench while reducing operational costs by 59.02%. This shows that underutilized RT cores can be successfully transformed into a highly efficient, cost-effective engine for large-scale spatial analytics in production systems.

Chapter 6: Conclusion

In this dissertation, we have investigated the repurposing of hardware-accelerated Ray Tracing (RT) cores on modern GPUs to address the bottlenecks inherent in large-scale spatial data processing. By mapping spatial index structures, geometric primitives, and spatial queries onto ray-primitive intersection tests and Bounding Volume Hierarchy (BVH) traversals, this work demonstrates that RT cores can deliver significant performance improvements over traditional CPU and GPU implementations, spanning from fundamental spatial operators to a full-fledged spatial database.

Our investigation begins with *RayJoin*, a real-time spatial join framework powered by RT cores that accelerates fundamental spatial queries, including Line Segment Intersection (LSI) and Point-In-Polygon (PIP) tests. To address the hardware constraint of single-precision floating-point ray tracing on geospatial datasets, we developed the conservative-representation technique that achieves double precision under the hardware limitation.

Building upon spatial join operators, we proposed *LibRTS*, a general-purpose spatial indexing library that abstracts the low-level complexities of RT hardware for spatial application developers. LibRTS reformulates point and range queries into RT-friendly ray-casting problems and addresses the load-balancing issue on GPUs, providing a developer-friendly interface for high-performance spatial queries.

X-HD exemplifies how to deliver a fast and performance-robust spatial function using ray tracing to compute the Hausdorff Distance. X-HD reformulates HD computation as a nearest-neighbor

search accelerated by hardware BVH traversal. By incorporating a uniform grid partitioning strategy to group spatially proximate points into cells and an early-break pruning optimization, X-HD significantly reduces runtime and device memory footprint.

Finally, to bridge the gap between individual operator acceleration and industrial spatial database systems, we developed *RayBooster*, an end-to-end RT-core acceleration engine integrated into the production-grade Apache SedonaDB. RayBooster resolves key system barriers, including data format mismatches, micro-indexing overheads, and combinatorial predicate complexity. RayBooster achieves both high performance and low cost over conventional database architectures by leveraging RT cores.

Looking forward, the techniques established in this dissertation lay the groundwork for several promising research directions. First, scaling RT-core acceleration across multi-GPU nodes and distributed frameworks will enable real-time spatial processing over massive datasets. In addition, harmonizing RT cores, Tensor Cores, and CUDA cores will help push the boundaries of next-generation spatial computing platforms and AI-driven geospatial analytics. Beyond the spatial domain, the huge potential of RT hardware offers opportunities to accelerate diverse non-rendering workloads, including computational lithography and CAD tools.

Bibliography

[1] Laila Abdelhafeez, Amr Magdy, and Vassilis J Tsotras. Sgpac: generalized scalable spatial groupby aggregations over complex polygons. *GeoInformatica*, pages 1–28, 2023.

[2] Danial Aghajarian and Sushil K Prasad. 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*, SIGSPATIAL '17, pages 1–4, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: a high performance spatial data warehousing system over mapreduce. *Proceedings of the VLDB Endowment*, 6(11):1009–1020, 2013.

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

[5] Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G Aref. A survey of learned indexes for the multi-dimensional space. *arXiv preprint arXiv:2403.06456*, 2024.

[6] Benjamin S Allen, James Anchell, Victor Anisimov, Thomas Applencourt, Abhishek Bagusetty, Ramesh Balakrishnan, Riccardo Balin, Solomon Bekele, Colleen Bertoni, Cyrus Blackworth, et al. Aurora: Architecting argonne's first exascale supercomputer for accelerated scientific discovery. *arXiv preprint arXiv:2509.08207*, 2025.

[7] Apache Sedona. Spatialbench single node benchmarks, Jan. 2026. Benchmark conducted September 22, 2025. Accessed: 2026-01-27.

[8] Apache Software Foundation. Apache sedona-db documentation, 2025. Accessed: 2026-01-12.

[9] Apache Software Foundation. SpatialBench - Apache Sedona. `https://sedona.apache.org/spatialbench/`, 2026. Accessed: 2026-02-21.

[10] Nicolas Aspert, Diego Santa-Cruz, and Touradj Ebrahimi. Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings. IEEE international conference on multimedia and expo*, volume 1, pages 705–708. IEEE, 2002.

[11] Mikhail J Atallah and Michael T Goodrich. Efficient plane sweeping in parallel. In *Proceedings of the second annual symposium on Computational geometry*, pages 216–225, 1986.

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

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

[14] Muhammad A Awad, Serban D Porumbescu, and John D Owens. A gpu multiversion b-tree. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 481–493, 2022.

[15] Kolawole O Babalola, Brian Patenaude, Paul Aljabar, Julia Schnabel, David Kennedy, William Crum, Stephen Smith, Tim F Cootes, Mark Jenkinson, and Daniel Rueckert. Comparison and evaluation of segmentation techniques for subcortical structures in brain mri. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2008: 11th International Conference, New York, NY, USA, September 6-10, 2008, Proceedings, Part I 11*, pages 409–416. Springer, 2008.

[16] Spyridon Bakas, Hamed Akbari, Aristeidis Sotiras, Michel Bilello, Martin Rozycki, Justin S Kirby, John B Freymann, Keyvan Farahani, and Christos Davatzikos. Advancing the cancer genome atlas glioma mri collections with expert segmentation labels and radiomic features. *Scientific data*, 4(1):1–13, 2017.

[17] Spyridon Bakas, Mauricio Reyes, Andras Jakab, Stefan Bauer, Markus Rempfler, Alessandro Crimi, Russell Takeshi Shinohara, Christoph Berger, Sung Min Ha, Martin Rozycki, Marcel Prastawa, Esther Alberts, Jana Lipkova, John Freymann, Justin Kirby, Michel Bilello, Hassan Fathallah-Shaykh, Roland Wiest, Jan Kirschke, Benedikt Wiestler, Rivka Colen, Aikaterini Kotrotsou, Pamela Lamontagne, Daniel Marcus, Mikhail Milchenko, Arash Nazeri, Marc-Andre Weber, Abhishek Mahajan, Ujjwal Baid, Elizabeth Gerstner, Dongjin Kwon, Gagan Acharya, Manu Agarwal, Mahbubul Alam, Alberto Albiol, Antonio Albiol, Francisco J. Albiol, Varghese Alex, Nigel Allinson, Pedro H. A. Amorim, Abhijit Amrutkar, Ganesh Anand, Simon Andermatt, Tal Arbel, Pablo Arbelaez, Aaron Avery, Muneeza Azmat, Pranjal B., W Bai, Subhashis Banerjee, Bill Barth, Thomas Batchelder, Kayhan Batmanghelich, Enzo Battistella, Andrew Beers, Mikhail Belyaev, Martin Bendszus, Eze Benson, Jose Bernal, Halandur Nagaraja Bharath, George Biros, Sotirios Bisdas, James Brown, Mariano Cabezas, Shilei Cao, Jorge M. Cardoso, Eric N Carver, Adrià Casamitjana, Laura Silvana Castillo, Marcel Catà, Philippe Cattin, Albert Cerigues, Vinicius S. Chagas, Siddhartha Chandra,

Yi-Ju Chang, Shiyu Chang, Ken Chang, Joseph Chazalon, Shengcong Chen, Wei Chen, Jefferson W Chen, Zhaolin Chen, Kun Cheng, Ahana Roy Choudhury, Roger Chylla, Albert Clérigues, Steven Colleman, Ramiro German Rodriguez Colmeiro, Marc Combalia, Anthony Costa, Xiaomeng Cui, Zhenzhen Dai, Lutao Dai, Laura Alexandra Daza, Eric Deutsch, Changxing Ding, Chao Dong, Shidu Dong, Wojciech Dudzik, Zach Eaton-Rosen, Gary Egan, Guilherme Escudero, Théo Estienne, Richard Everson, Jonathan Fabrizio, Yong Fan, Longwei Fang, Xue Feng, Enzo Ferrante, Lucas Fidon, Martin Fischer, Andrew P. French, Naomi Fridman, Huan Fu, David Fuentes, Yaozong Gao, Evan Gates, David Gering, Amir Gholami, Willi Gierke, Ben Glocker, Mingming Gong, Sandra González-Villá, T. Grosges, Yuanfang Guan, Sheng Guo, Sudeep Gupta, Woo-Sup Han, Il Song Han, Konstantin Harmuth, Huiguang He, Aura Hernández-Sabaté, Evelyn Herrmann, Naveen Himthani, Winston Hsu, Cheyu Hsu, Xiaojun Hu, Xiaobin Hu, Yan Hu, Yifan Hu, Rui Hua, Teng-Yi Huang, Weilin Huang, Sabine Van Huffel, Quan Huo, Vivek HV, Khan M. Iftekharuddin, Fabian Isensee, Mobarakol Islam, Aaron S. Jackson, Sachin R. Jambawalikar, Andrew Jesson, Weijian Jian, Peter Jin, V Jeya Maria Jose, Alain Jungo, B Kainz, Konstantinos Kamnitsas, Po-Yu Kao, Ayush Karnawat, Thomas Kellermeier, Adel Kermi, Kurt Keutzer, Mohamed Tarek Khadir, Mahendra Khened, Philipp Kickingereder, Geena Kim, Nik King, Haley Knapp, Urspeter Knecht, Lisa Kohli, Deren Kong, Xiangmao Kong, Simon Koppers, Avinash Kori, Ganapathy Krishnamurthi, Egor Krivov, Piyush Kumar, Kaisar Kushibar, Dmitrii Lachinov, Tryphon Lambrou, Joon Lee, Chengen Lee, Yuehchou Lee, M Lee, Szidonia Lefkovits, Laszlo Lefkovits, James Levitt, Tengfei Li, Hongwei Li, Wenqi Li, Hongyang Li, Xiaochuan Li, Yuexiang Li, Heng Li, Zhenye Li, Xiaoyu Li, Zeju Li, XiaoGang Li, Wenqi Li, Zheng-Shen Lin, Fengming Lin, Pietro Lio, Chang Liu, Boqiang Liu, Xiang Liu, Mingyuan Liu, Ju Liu, Luyan Liu, Xavier Llado, Marc Moreno Lopez, Pablo Ribalta Lorenzo, Zhentai Lu, Lin Luo, Zhigang Luo, Jun Ma, Kai Ma, Thomas Mackie, Anant Madabushi, Issam Mahmoudi, Klaus H. Maier-Hein, Pradipta Maji, CP Mammen, Andreas Mang, B. S. Manjunath, Michal Marcinkiewicz, S McDonagh, Stephen McKenna, Richard McKinley, Miriam Mehl, Sachin Mehta, Raghav Mehta, Raphael Meier, Christoph Meinel, Dorit Merhof, Craig Meyer, Robert Miller, Sushmita Mitra, Aliasgar Moiyadi, David Molina-Garcia, Miguel A. B. Monteiro, Grzegorz Mrukwa, Andriy Myronenko, Jakub Nalepa, Thuyen Ngo, Dong Nie, Holly Ning, Chen Niu, Nicholas K Nuechterlein, Eric Oermann, Arlindo Oliveira, Diego D. C. Oliveira, Arnau Oliver, Alexander F. I. Osman, Yu-Nian Ou, Sebastien Ourselin, Nikos Paragios, Moo Sung Park, Brad Paschke, J. Gregory Pauloski, Kamlesh Pawar, Nick Pawlowski, Linmin Pei, Suting Peng, Silvio M. Pereira, Julian Perez-Beteta, Victor M. Perez-Garcia, Simon Pezold, Bao Pham, Ashish Phophalia, Gemma Piella, G. N. Pillai, Marie Piraud, Maxim Pisov, Anmol Popli, Michael P. Pound, Reza Pourreza, Prateek Prasanna, Vesna Prkovska, Tony P. Pridmore, Santi Puch, Élodie Puybareau, Buyue Qian, Xu Qiao, Martin Rajchl, Swapnil Rane, Michael Rebsamen, Hongliang Ren, Xuhua Ren, Karthik Revanuru, Mina Rezaei, Oliver Rippel, Luis Carlos Rivera, Charlotte Robert, Bruce Rosen, Daniel Rueckert, Mohammed Safwan, Mostafa Salem, Joaquim Salvi, Irina Sanchez, Irina Sánchez, Heitor M. Santos, Emmett Sartor, Dawid Schellingerhout, Klaudius Scheufele, Matthew R.

Scott, Artur A. Scussel, Sara Sedlar, Juan Pablo Serrano-Rubio, N. Jon Shah, Nameetha Shah, Mazhar Shaikh, B. Uma Shankar, Zeina Shboul, Haipeng Shen, Dinggang Shen, Linlin Shen, Haocheng Shen, Varun Shenoy, Feng Shi, Hyung Eun Shin, Hai Shu, Diana Sima, M Sinclair, Orjan Smedby, James M. Snyder, Mohammadreza Soltaninejad, Guidong Song, Mehul Soni, Jean Stawiaski, Shashank Subramanian, Li Sun, Roger Sun, Jiawei Sun, Kay Sun, Yu Sun, Guoxia Sun, Shuang Sun, Yannick R Suter, Laszlo Szilagyi, Sanjay Talbar, Dacheng Tao, Dacheng Tao, Zhongzhao Teng, Siddhesh Thakur, Meenakshi H Thakur, Sameer Tharakan, Pallavi Tiwari, Guillaume Tochon, Tuan Tran, Yuhsiang M. Tsai, Kuan-Lun Tseng, Tran Anh Tuan, Vadim Turlapov, Nicholas Tustison, Maria Vakalopoulou, Sergi Valverde, Rami Vanguri, Evgeny Vasiliev, Jonathan Ventura, Luis Vera, Tom Vercauteren, C. A. Verrastro, Lasitha Vidyaratne, Veronica Vilaplana, Ajeet Vivekanandan, Guotai Wang, Qian Wang, Chiatse J. Wang, Weichung Wang, Duo Wang, Ruixuan Wang, Yuanyuan Wang, Chunliang Wang, Guotai Wang, Ning Wen, Xin Wen, Leon Weninger, Wolfgang Wick, Shaocheng Wu, Qiang Wu, Yihong Wu, Yong Xia, Yanwu Xu, Xiaowen Xu, Peiyuan Xu, Tsai-Ling Yang, Xiaoping Yang, Hao-Yu Yang, Junlin Yang, Haojin Yang, Guang Yang, Hongdou Yao, Xujiong Ye, Changchang Yin, Brett Young-Moxon, Jinhua Yu, Xiangyu Yue, Songtao Zhang, Angela Zhang, Kun Zhang, Xuejie Zhang, Lichi Zhang, Xiaoyue Zhang, Yazhuo Zhang, Lei Zhang, Jianguo Zhang, Xiang Zhang, Tianhao Zhang, Sicheng Zhao, Yu Zhao, Xiaomei Zhao, Liang Zhao, Yefeng Zheng, Liming Zhong, Chenhong Zhou, Xiaobing Zhou, Fan Zhou, Hongtu Zhu, Jin Zhu, Ying Zhuge, Weiwei Zong, Jayashree Kalpathy-Cramer, Keyvan Farahani, Christos Davatzikos, Koen van Leemput, and Bjoern Menze. Identifying the best machine learning algorithms for brain tumor segmentation, progression assessment, and overall survival prediction in the brats challenge. *arXiv preprint arXiv:1811.02629*, 2019.

[18] Josh Baker. Tg: Geometry library for C. `https://github.com/tidwall/tg`, 2026. Accessed: 2026-02-05.

[19] Aaron Barnes, Fangjia Shen, and Timothy G. Rogers. Extending gpu ray-tracing units for hierarchical search acceleration. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1027–1040, 2024.

[20] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 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*, pages 322–331, 1990.

[21] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Jurgen Gall. Semantickitti: A dataset for semantic scene understanding of lidar sequences. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9297–9307, 2019.

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

[23] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[24] Pedro Gabriel Kohl Bertella, Yuri Kaszubowski Lopes, Rafael Alves Paes de Oliveira, and Anderson Chaves Carniel. A systematic review of spatial approximations in spatial database systems. *Journal of Information and Data Management*, 13(2), 2022.

[25] Guy E Blelloch and Magdalen Dobson. Parallel nearest neighbors in low dimensions with batch updates. In *2022 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 195–208. SIAM, 2022.

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

[27] Panagiotis Bouros and Nikos Mamoulis. Spatial joins: what's next? *SIGSPATIAL Special*, 11(1):13–21, 2019.

[28] Sebastian Breß. The design and implementation of cogadb: A column-oriented gpu-accelerated dbms. *Datenbank-Spektrum*, 14(3):199–209, 2014.

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

[30] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. Gpu database systems characterization and optimization. *Proceedings of the VLDB Endowment*, 17(3):441–454, 2023.

[31] Antonio Cardone, Satyandra K Gupta, Abhijit Deshmukh, and Mukul Karnik. Machining feature-based similarity assessment algorithms for prismatic machined parts. *Computer-Aided Design*, 38(9):954–972, 2006.

[32] Anderson Chaves Carniel and Cristina Dutra Aguiar. Spatial index structures for modern storage devices: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 35(9):9578–9597, 2023.

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

[34] Anderson Chaves Carniel. Defining and designing spatial queries: the role of spatial relationships. *Geo-spatial Information Science*, 27(6):1868–1892, 2024.

[35] Yilin Chen, Fazhi He, Yiqi Wu, and Neng Hou. A local start search algorithm to compute exact hausdorff distance for arbitrary point sets. *Pattern Recognition*, 67:139–148, 2017.

[36] Mingmin Chi, Antonio Plaza, Jon Atli Benediktsson, Zhongyi Sun, Jinsheng Shen, and Yangyong Zhu. Big data for remote sensing: Challenges and opportunities. *Proceedings of the IEEE*, 104(11):2207–2219, 2016.

[37] Haili Chui and Anand Rangarajan. A new point matching algorithm for non-rigid registration. *Computer Vision and Image Understanding*, 89(2-3):114–141, 2003.

[38] James H Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[39] Eliseo Clementini, Paolino Di Felice, and Peter Van Oosterom. A small set of formal topological relationships suitable for end-user interaction. In *International symposium on spatial databases*, pages 277–295. Springer, 1993.

[40] Conservation.gov. Us county boundaries. `https://hub.arcgis.com/datasets/17b89622df5643cda5339ae6649247a6_0`, 2025. Feature Layer, ArcGIS Hub.

[41] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[42] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[43] Harish Doraiswamy and Juliana Freire. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1875–1885, 2020.

[44] Alex Dunn. Tips and tricks: Ray tracing best practices, March 2019.

[45] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics (tog)*, 9(1):66–104, 1990.

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

[47] Esri. Arcgis hub, 2023.

[48] Esri. Five reasons every data science team needs a geographer. `https://web.archive.org/web/20250121213004/https://www.esri.com/about/newsroom/arcuser/five-reasons-every-data-science-team-needs-a-geographer/`, 2025. Accessed: 2025-03-26.

[49] Esri. Usa detailed water bodies. `[https://hub.arcgis.com/datasets/esri::usa-detailed-water-bodies/about](https://hub.arcgis.com/datasets/esri::usa-detailed-water-bodies/about)`, 2025. Feature Layer, ArcGIS Hub.

[50] Esri. Usa zip code areas. https://hub.arcgis.com/maps/esri::usa-zip-code-areas/about, 2025. Web Map, ArcGIS Hub.

[51] Esri Federal Data. U.s. census blocks. https://hub.arcgis.com/datasets/d795eaa6ee7a40bdb2efeb2d001bf823_0/about, 2025. Feature Layer, ArcGIS Hub, utilizing data from the U.S. Census Bureau.

[52] Giannis Evagorou and Thomas Heinis. Mambo-indexing dead space to accelerate spatial queries. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, pages 73–84, 2021.

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

[54] Jianping Fan, David KY Yau, Ahmed K Elmagarmid, and Walid G Aref. Automatic image segmentation by integrating color-edge extraction and seeded region growing. *IEEE transactions on image processing*, 10(10):1454–1466, 2001.

[55] Peng Fan, Wei Wang, Ruofeng Tong, Hailong Li, and Min Tang. gdist: Efficient distance computation between 3d meshes on gpu. In *SIGGRAPH Asia 2024 Conference Papers*, pages 1–11, 2024.

[56] Ulrich Finke and Klaus Hinrichs. A spatial data model and a topological sweep algorithm for map overlay. In *International Symposium on Spatial Databases*, pages 162–177. Springer, 1993.

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

[58] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Nestgpu: Nested query processing on gpu. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1008–1019. IEEE, 2021.

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

[60] William Randolph Franklin, Venkateshkumar Sivaswami, David Sun, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. Calculating the area of overlaid polygons without constructing the overlay. *Cartography and Geographic Information Systems*, 21(2):81–89, 1994.

[61] Roger Frye and Mark McKenney. Per segment plane sweep line segment intersection on the gpu. In *Proceedings of the 30th International Conference on Advances in Geographic Information Systems*, pages 1–4, 2022.

[62] Yongsheng Gao and Maylor KH Leung. Face recognition using line edge map. *IEEE transactions on pattern analysis and machine intelligence*, 24(6):764–779, 2002.

[63] Andrew Gardner, Chris Tchou, Tim Hawkins, and Paul Debevec. Linear light source reflectometry. *ACM Transactions on Graphics (TOG)*, 22(3):749–758, 2003.

[64] Liang Geng, Rubao Lee, and Xiaodong Zhang. RayJoin: Fast and precise spatial join. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 124–136, 2024.

[65] Liang Geng, Rubao Lee, and Xiaodong Zhang. LibRTS: A spatial indexing library by ray tracing. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 396–411, 2025.

[66] Thanasis Georgiadis and Nikos Mamoulis. Raster intervals: An approximation technique for polygon intersection joins. *Proceedings of the ACM on Management of Data*, 1(1):1–18, 2023.

[67] Thanasis Georgiadis and Nikos Mamoulis. Scalable spatial topology joins. In *EDBT*, pages 110–116, 2026.

[68] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. Raster interval object approximations for spatial intersection joins. *The VLDB Journal*, 34(1):8, 2025.

[69] GEOS developers. *GEOS – Geometry Engine, Open Source*, 2026. Accessed: 2026-02-05.

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

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

[72] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M. Aamodt. Generalizing ray tracing accelerators for tree traversals on gpus. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1041–1057, 2024.

[73] Anton Hackl and Eleni Tzirita Zacharatou. Pierce: Gpu ray tracing for spatial joins over complex 3d data. *arXiv preprint arXiv:2609.24513*, 2026.

[74] Ali Hadian and Thomas Heinis. Considerations for handling updates in learned index structures. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2019.

[75] Herman Johannes Haverkort. *Results on geometric networks and data structures*. PhD thesis, Utrecht University, 2004.

[76] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. Query processing on tensor computation runtimes. *Proc. VLDB Endow.*, 15(11):2811–2825, July 2022.

[77] HEAVY.AI. Heavy.ai documentation, 2026. Accessed: 2026-06-22.

[78] Justus Henneberg and Felix Schuhknecht. Rtindex: Exploiting hardware-accelerated gpu raytracing for database indexing. *Proc. VLDB Endow.*, 16(13):4268–4281, sep 2023.

[79] Justus Henneberg, Felix Schuhknecht, Rosina Kharal, and Trevor Brown. More bang for your buck (et): Fast and space-efficient hardware-accelerated coarse-granular indexing on gpus. *arXiv preprint arXiv:2406.03965*, 2024.

[80] Dongxiao Hu, Lijun Chen, Hao Fang, Zhiwei Fang, Tao Li, and Yanjie Gao. Spatio-temporal trajectory similarity measures: A comprehensive survey and quantitative study. *IEEE Transactions on Knowledge and Data Engineering*, 36(5):2191–2212, May 2024.

[81] Daniel P Huttenlocher, Klara Kedem, and Jon M Kleinberg. On dynamic voronoi diagrams and the minimum hausdorff distance for point sets under euclidean motion in the plane. In *Proceedings of the eighth annual symposium on Computational geometry*, pages 110–119, 1992.

[82] Daniel P Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence*, 15(9):850–863, 1993.

[83] IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[84] Kinetica DB Inc. Kinetica: The database for time & space, 2023.

[85] ISO/IEC. Iso/iec 13249-3:2016, 2021.

[86] Thiago Ize. Robust bvh ray traversal. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):12–27, 2013.

[87] Edwin H Jacox and Hanan Samet. Spatial join techniques. *ACM Transactions on Database Systems (TODS)*, 32(1):7–es, 2007.

[88] Wenqi Jiang, Oleh-Yevhen Khavrona, Martin Parvanov, and Gustavo Alonso. Swiftspatial: Spatial joins on modern hardware. *Proceedings of the ACM on Management of Data*, 3(3):1–27, 2025.

[89] William Kahan. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

[90] Benjamin Kahl. Hardware acceleration of progressive refinement radiosity using nvidia rtx. *arXiv preprint arXiv:2303.14831*, 2023.

[91] Kohei KaiGai. PG-Strom: Master development repository, 2025. Accessed: 2026-06-22.

[92] Victor Kamel, Hanxueyu Yan, and Sean Chester. Clover: A gpu-native, spatio-graph-based approach to exact knn. In *Proceedings of the 2025 International Conference on Supercomputing (ICS '25)*, pages 1–14, Salt Lake City, UT, USA, 2025. ACM.

[93] Davood Karimi and Septimiu E Salcudean. Reducing the hausdorff distance in medical image segmentation with convolutional neural networks. *IEEE Transactions on medical imaging*, 39(2):499–513, 2019.

[94] Tomas Karnagel, René Müller, and Guy M Lohman. Optimizing gpu-accelerated group-by and aggregation. *ADMS@ VLDB*, 8:20, 2015.

[95] Charles FF Karney. Transverse mercator with an accuracy of a few nanometers. *Journal of Geodesy*, 85:475–485, 2011.

[96] Tero Karras. 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*, pages 33–37, 2012.

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

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

[99] S. Keely. Reduced precision for hardware ray tracing in gpus. In *Proceedings of High Performance Graphics*, HPG '14, page 29–40, Goslar, DEU, 2014. Eurographics Association.

[100] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Ip. lsh. dbscan: Integrated parallel density-based clustering through locality-sensitive hashing. In *European Conference on Parallel Processing*, pages 268–284. Springer, 2022.

[101] Hassan Khotanlou, Olivier Colliot, Jamal Atif, and Isabelle Bloch. 3d brain tumor segmentation in mri using fuzzy classification, symmetry analysis and spatially constrained deformable models. *Fuzzy sets and systems*, 160(10):1457–1473, 2009.

[102] Young Woo Kim, Jaehong Lee, and Duksu Kim. Rt-hdist: Ray-tracing core-based hausdorff distance computation. In *Computer Graphics Forum*, page e70229. Wiley Online Library, 2025.

[103] YoungWoo Kim, Sungmin Kwon, and Duksu Kim. Rtpd: Penetration depth calculation using hardware accelerated ray-tracing. *arXiv preprint arXiv:2502.12463*, 2025.

[104] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. Duckdb-wasm: Fast analytical processing for the web. *Proceedings of the VLDB Endowment*, 15(12):3574–3577, 2022.

[105] Vasileios Kopsachilis, Michalis Vaitis, Nikos Mamoulis, and Dimitris Kotzinos. Recommending geo-semantically related classes for link discovery. *Journal on Data Semantics*, 9:151–177, 2020.

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

[107] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 313–324, 1996.

[108] Moritz Laass. Point in polygon tests using hardware accelerated ray tracing. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, pages 666–667, 2021.

[109] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. Apache arrow datafusion: A fast, embeddable, modular analytic query engine. In *Companion of the 2024 International Conference on Management of Data*, pages 5–17, 2024.

[110] Hai Lan, Jiong Xie, Zhifeng Bao, Feifei Li, Wei Tian, Fang Wang, Sheng Wang, and Ailin Zhang. Vre: a versatile, robust, and economical trajectory data system. *Proceedings of the VLDB Endowment*, 15(12):3398–3410, Aug. 2022.

[111] Margaret Lawson, William Gropp, and Jay Lofstead. Exploring spatial indexing for accelerated feature retrieval in hpc. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 605–614. IEEE, 2022.

[112] Jae-Gil Lee, Jiawei Han, Xiaolei Li, and Hector Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. *Proceedings of the VLDB Endowment*, 1(1):1081–1094, Aug. 2008.

[113] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The art of balance: a rateupdb™ experience of building a cpu/gpu hybrid database product. *Proceedings of the VLDB Endowment*, 14(12):2999–3013, 2021.

[114] Noppon Lertchuwongsa, Michele Gouiffes, and Bertrand Zavidovique. Enhancing a disparity map by color segmentation. *Integrated Computer-Aided Engineering*, 19(4):381–397, 2012.

[115] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment*, 9(14):1647–1658, 2016.

[116] Ke Li, Lisi Chen, Shuo Shang, Haiyan Wang, Yang Liu, Panos Kalnis, and Bin Yao. Towards controlling the transmission of diseases: Continuous exposure discovery over massive-scale moving objects. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*. International Joint Conferences on Artificial Intelligence Organization, 2022. Supported by NSFC (U21B2046, U2001212, 62032001, 61932004), Sichuan Science and Technology Program (2021YFS0007), and others.

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

[118] Ruiyuan Li, Huajun He, Rubin Wang, Sijie Ruan, Yuan Sui, Jie Bao, and Yu Zheng. Trajmesa: A distributed nosql storage engine for big trajectory data. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 2002–2005, Dallas, TX, USA, 2020. IEEE.

[119] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, et al. Scaling gpu-accelerated databases beyond gpu memory size. *Proceedings of the VLDB Endowment*, 18(11):4518–4531, 2025.

[120] Yun Li, Yixiang Fang, Reynold Cheng, and Wenjie Zhang. Spatial pattern matching: A new direction for finding spatial objects. *SIGSPATIAL Special*, 11(1):3–12, 2019.

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

[122] Jiaxin Liu, Rubao Lee, Cathy H Xia, and Xia Odong Zhang. A stable marriage requires a shared residence with low contention and mutual complementarity. In *2025 34th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 416–430. IEEE, 2025.

[123] 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.

G-learned index: Enabling efficient learned index on gpu. *IEEE Transactions on Parallel and Distributed Systems*, 35(6):950–967, 2024.

[124] Qiyu Liu, Maocheng Li, Yuxiang Zeng, Yanyan Shen, and Lei Chen. How good are multi-dimensional learned indices? an experimental survey. *arXiv preprint arXiv:2405.05536*, 2024.

[125] Yiming Liu and Satish Puri. Efficient filters for geometric intersection computations using gpu. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, pages 487–496, 2020.

[126] Yiming Liu, Jie Yang, and Satish Puri. 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)*, pages 141–151. IEEE, 2019.

[127] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. Juno: Optimizing high-dimensional approximate nearest neighbour search with sparsity-aware algorithm and ray-tracing core mapping. *arXiv preprint arXiv:2312.01712*, page 549–565, 2023.

[128] Helen Lockett and Marin Guenov. Similarity measures for mid-surface quality evaluation. *Computer-Aided Design*, 40(3):368–380, 2008.

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

[130] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. RTScan: Efficient scan with ray tracing cores. *Proceedings of the VLDB Endowment*, 17(6):1460–1472, 2024.

[131] Salles VG Magalhães, Marcus VA Andrade, W Randolph Franklin, and Wenli Li. 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*, pages 45–54, 2015.

[132] Durga Mandarapu, Vani Nagarajan, and Milind Kulkarni. Generalized neighbor search using commodity hardware acceleration. *arXiv preprint arXiv:2311.09168*, 2023.

[133] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. Arkade: k-nearest neighbor search with non-euclidean distances using gpu ray tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing*, pages 14–25, 2024.

[134] Adam Marrs, Peter Shirley, and Ingo Wald. *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. Springer Nature, 2021.

[135] J Maul, Sarah Said, N Ruiter, and Torsten Hopp. 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*, pages 87–96. Springer, 2021.

[136] Calvin R Maurer, Rensheng Qi, and Vijay Raghavan. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(2):265–270, 2003.

[137] Mark McKenney, Gabriel De Luna, Schiller Hill, and Logan Lowell. Geospatial overlay computation on the gpu. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 473–476, 2011.

[138] Mark McKenney and Tynan McGuire. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 392–395, 2009.

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

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

[141] Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. Efficient multi-gpu graph processing with remote work stealing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 191–204. IEEE, 2023.

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

[143] Bjoern H Menze, Andras Jakab, Stefan Bauer, Jayashree Kalpathy-Cramer, Keyvan Farahani, Justin Kirby, Yuliya Burren, Nicole Porz, Johannes Slotboom, Roland Wiest, Levente Lanczi, Elizabeth Gerstner, Marc-André Weber, Tal Arbel, Brian B Avants, Nicholas Ayache, Patricia Buendia, D Louis Collins, Nicolas Cordier, Jason J Corso, Antonio Criminisi, Tilak Das, Hervé Delingette, Çağatay Demiralp, Christopher R Durst, Michel Dojat, Senan Doyle, Joana Festa, Florence Forbes, Ezequiel Geremia, Ben Glocker, Polina Golland, Xiaotao Guo, Andac Hamamci, Khan M Iftekharuddin, Raj Jena, Nigel M John, Ender Konukoglu, Danial Lashkari, José Antonió Mariz, Raphael Meier, Sérgio Pereira, Doina Precup, Stephen J Price, Tammy Riklin Raviv, Syed M S Reza, Michael Ryan, Duygu Sarikaya, Lawrence Schwartz, Hoo-Chang Shin, Jamie Shotton, Carlos A Silva, Nuno Sousa, Nagesh K Subbanna, Gabor

Szekely, Thomas J Taylor, Owen M Thomas, Nicholas J Tustison, Gozde Unal, Flor Vasseur, Max Wintermark, Dong Hye Ye, Liang Zhao, Binsheng Zhao, Darko Zikic, Marcel Prastawa, Mauricio Reyes, and Koen Van Leemput. The multimodal brain tumor image segmentation benchmark (brats). *IEEE transactions on medical imaging*, 34(10):1993–2024, 2014.

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

[145] Deng Min, Li Zhilin, and Chen Xiaoyong. Extended hausdorff distance for spatial objects in gis. *International Journal of Geographical Information Science*, 21(4):459–475, 2007.

[146] Jakub Mitura, Beata E Chrapko, and Oliwia Bachanek-Mitura. Medvoxelhd: Improved cuda-accelerated morphological hausdorff distances in medical image analysis. *SoftwareX*, 26:101744, 2024.

[147] Frederic Morain-Nicolier, Stephane Lebonvallet, Etienne Baudrier, and Su Ruan. Hausdorff distance based 3d quantification of brain tumor evolution from mri images. In *2007 29th Annual international conference of the IEEE engineering in medicine and biology society*, pages 5597–5600. IEEE, 2007.

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

[149] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing. *Proceedings of the VLDB Endowment*, 16(4):670–683, 2022.

[150] Kyriakos Mouratidis, Jing Li, Yu Tang, and Nikos Mamoulis. Joint search by social and spatial proximity. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):781–793, 2014.

[151] Vani Nagarajan, Rohan Gangaraju, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. Rt-barneshut: Accelerating barnes-hut using ray-tracing hardware. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 43–56, 2025.

[152] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. Rt-knns unbound: Using rt cores to accelerate unrestricted neighbor search. In *Proceedings of the 37th International Conference on Supercomputing*, pages 289–300, 2023.

[153] New York City Taxi and Limousine Commission. TLC Trip Record Data. `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, 2025. Accessed: 2026-02-21.

[154] Jürg Nievergelt and Franco P Preparata. Plane-sweep algorithms for intersecting geometric figures. *Communications of the ACM*, 25(10):739–747, 1982.

[155] Sarana Nutanong, Edwin H Jacox, and Hanan Samet. An incremental hausdorff distance calculation algorithm. *Proceedings of the VLDB Endowment*, 4(8):506–517, 2011.

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

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

[158] NVIDIA. cuspatial. https://docs.rapids.ai/api/cuspatial/stable/, 2023. Last accessed 2024-08-07.

[159] NVIDIA. Nvidia optix 8.0 programming guide. https://raytracing-docs.nvidia.com/optix8/guide/index.html, 2024. Last accessed 2024-08-07.

[160] NVIDIA Corporation. cuspatial: Gpu-accelerated geospatial and spatiotemporal algorithms. https://github.com/rapidsai/cuspatial, 2023. Software available from github.com.

[161] NVIDIA Corporation. NVIDIA OptiX 8.1 - Programming Guide, 2024. Available at: https://raytracing-docs.nvidia.com/optix8/guide/index.html. Accessed: 11 December 2025.

[162] NVIDIA Corporation. *NVIDIA DGX Spark User Guide*. NVIDIA, 2026. Accessed: 2026-05-02.

[163] Observable, Inc. Observable: The collaborative data platform, 2026. Accessed: 2026-02-28.

[164] Open Geospatial Consortium. Geoparquet: Server-native spatial data in parquet, 2025. Accessed: 2026-01-12.

[165] Jack A Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD international conference on Management of data*, pages 326–336, 1986.

[166] Muhammad Osama, Serban D Porumbescu, and John D Owens. A programming model for gpu load balancing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 79–91, 2023.

[167] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment*, 11(11):1661–1673, 2018.

[168] Varun Pandey, Alexander van Renen, Andreas Kipf, Jialin Ding, Ibrahim Sabek, and Alfons Kemper. The case for learned spatial indexes. In Bingsheng He, Berthold Reinwald, and Yingjun Wu, editors, *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*, 2020.

[169] Dimitris Papadias, Yufei Tao, Kyriakos Mouratidis, and Chun Kit Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems (TODS)*, 30(2):529–576, 2005.

[170] Shibin Parameswaran and Jeffrey Ellen. Identifying outliers in human movement trajectories clustered by hausdorff distance. In *Proceedings of the International Conference on Data Science (ICDATA)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2014.

[171] Steven G Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, et al. Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog)*, 29(4):1–13, 2010.

[172] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus. *Proceedings of the VLDB Endowment*, 14(2):202–214, 2020.

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

[174] Jeff M. Phillips and Pingfan Tang. Simple distances for trajectories via landmarks. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19)*, pages 468–471, New York, NY, USA, 2019. Association for Computing Machinery.

[175] PostGIS Development Team. ST_HausdorffDistance — PostGIS Documentation, 2025. Accessed: 2025-04-01.

[176] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 618–627. IEEE, 2015.

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

[178] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. Fair benchmarking considered difficult: Common pitfalls in database performance testing. In *Proceedings of the Workshop on Testing Database Systems*, pages 1–6, 2018.

[179] Mark Raasveldt and Hannes Mühleisen. DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[180] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, 1(3):244–256, 1972.

[181] RAPIDS Development Team. RMM: RAPIDS Memory Manager. `https://github.com/rapidsai/rmm`, 2026. Accessed: 2026-06-22.

[182] Yeasir Rayhan and Walid G Aref. Simd-ified r-tree query processing and optimization. In *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*, pages 1–10, 2023.

[183] John T Robinson. The kdb-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, 1981.

[184] Marcel Roeloffzen. *Point Location*, pages 1–5. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[185] Jérôme Rony, Jonas Adler, James Lottes, Aapo Kyrola, Tri Huynh, and Adam Lerer. Distorch: A fast gpu implementation of 3d hausdorff distance. In *Medical Imaging with Deep Learning (MIDL)*, 2025.

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

[187] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-gpu environment. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, volume 14, page 708, 2020.

[188] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on gpus: Design and implementation. In *Proceedings of the 29th international conference on scientific and statistical database management*, pages 1–12, 2017.

[189] Jegoon Ryu and Sei-ichiro Kamata. An efficient computational algorithm for hausdorff distance based on points-ruling-out and systematic random sampling. *Pattern Recognition*, 114:107857, 2021.

[190] Justin Salmon and Simon McIntosh-Smith. 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)*, pages 19–29. IEEE, 2019.

[191] Markus Schneider and Thomas Behr. Topological relationships between complex spatial objects. *ACM Transactions on Database Systems (TODS)*, 31(1):39–81, 2006.

[192] Philip J Schneider and Barbara A Schauer. Hazus—its development and its future. *Natural Hazards Review*, 7(2):40–44, 2006.

[193] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 1617–1632, 2020.

[194] Jiachen Shi, Gao Cong, and Xiao-Li Li. Learned index benefits: Machine learning based index performance estimation. *Proceedings of the VLDB Endowment*, 15(13):3950–3962, 2022.

[195] Xuri Shi, Kai Zhang, X Sean Wang, Xiaodong Zhang, and Rubao Lee. Raydb: Building databases with ray tracing cores. *Proceedings of the VLDB Endowment*, 19(1):43–55, 2025.

[196] Moshe Shimrat. Algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5(8):434, 1962.

[197] Jaewoo Shin, Ahmed R Mahmood, and Walid G Aref. 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*, pages 169–178, 2019.

[198] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 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)*, pages 2285–2290. IEEE, 2021.

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

[200] Darius Šidlauskas, Simonas Šaltenis, and Christian S Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 37–48, 2012.

[201] Dong-Gyu Sim, Oh-Kyu Kwon, and Rae-Hong Park. Object matching algorithms using robust hausdorff distance measures. *IEEE Transactions on image processing*, 8(3):425–429, 1999.

[202] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709. IEEE, 2019.

[203] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)*, 6(1):1–55, 1974.

[204] Abdel Aziz Taha and Allan Hanbury. An efficient algorithm for calculating the exact hausdorff distance. *IEEE transactions on pattern analysis and machine intelligence*, 37(11):2153–2163, 2015.

[205] Abdel Aziz Taha and Allan Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC medical imaging*, 15:1–28, 2015.

[206] Mingjie Tang, Yongyang Yu, Qutaibah M Malluhi, Mourad Ouzzani, and Walid G Aref. Locationspark: A distributed in-memory data management system for big spatial data. *Proceedings of the VLDB Endowment*, 9(13):1565–1568, 2016.

[207] RAPIDS Development Team. *RAPIDS: Libraries for End to End GPU Data Science*, 2026. Accessed: 2026-06-22.

[208] Dejun Teng, Furqan Baig, Qiheng Sun, Jun Kong, and Fusheng Wang. Ideal: a vector-raster hybrid model for efficient spatial queries over complex polygons. In *2021 22nd IEEE International Conference on Mobile Data Management (MDM)*, pages 99–108. IEEE, 2021.

[209] The GeoPandas Developers. Geopandas documentation, 2026. Accessed: 2026-01-21.

[210] Marjorie Thomas. A generalization of poisson's binomial limit for use in ecology. *Biometrika*, 36(1/2):18–25, 1949.

[211] Diego G Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A Boncz. Optimizing group-by and aggregation using gpu-cpu co-processing. In *ADMS@ VLDB*, pages 1–10, 2018.

[212] Hanieh Toutouni, Suman Chakraborty, Yicheng Tu, and Jiajun Huang. Octane: Breaking the neighbor-list bottleneck in gpu molecular dynamics. In *Proceedings of the 40th ACM International Conference on Supercomputing*, pages 30–41, 2026.

[213] Transaction Processing Performance Council (TPC). TPC Benchmark H (Decision Support) Standard Specification. `http://www.tpc.org/tpch/`, 2021. Version 3.0.0.

[214] Kuo-Kun Tseng, Ran Zhang, Chien-Ming Chen, and Mohammad Mehedi Hassan. Dnetunet: a semi-supervised cnn of medical image segmentation for super-computing ai service. *The Journal of Supercomputing*, 77(4):3594–3615, 2021.

[215] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318, 1994.

[216] Jan W van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems*, 18(1):49–67, 1991.

[217] Ingo Wald. Gpu-friendly, parallel, and (almost-) in-place construction of left-balanced kd trees. *arXiv preprint arXiv:2211.00120*, 2022.

[218] Ingo Wald, Will Usher, Nathan Morrical, Laura Lediaev, and Valerio Pascucci. Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *High Performance Graphics (Short Papers)*, volume 7, pages 7–13, 2019.

[219] Congying Wang, Jia Yu, and Zhuoyue Zhao. 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*, pages 1–12, 2023.

[220] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 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*, pages 38–52, 2019.

[221] Kaibo Wang, Yin Huai, Rubao Lee, Fusheng Wang, Xiaodong Zhang, and Joel H Saltz. Accelerating pathology image data cross-comparison on cpu-gpu hybrid systems. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 5, page 1543. NIH Public Access, 2012.

[222] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with gpus. *Proc. VLDB Endow.*, 7(11):1011–1022, 2014.

[223] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 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*, pages 1–12, 2016.

[224] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel dbscan. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2555–2571, 2020.

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

[226] Randall T Whitman, Michael B Park, Sarah M Ambrose, and Erik G Hoel. Spatial indexing and analytics on hadoop. In *Proceedings of the 22nd ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 73–82, 2014.

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

[228] Andrew Wooler. Air force's digital directorate awards kinetica contract with $100m ceiling for real-time intelligence, 2021.

[229] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 44–54, 2014.

[230] Yiqi Wu, Fazhi He, and Soonhung Han. Collaborative cad synchronization based on a symmetric and consistent modeling procedure. *Symmetry*, 9(4):59, 2017.

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

[232] Mengbai Xiao, Hao Wang, Liang Geng, Rubao Lee, and Xiaodong Zhang. An rdma-enabled in-memory computing platform for r-tree on clusters. *ACM Transactions on Spatial Algorithms and Systems (TSAS)*, 8(2):1–26, 2022.

[233] Zhixiong Xiao, Mengbai Xiao, Yuan Yuan, Dongxiao Yu, Rubao Lee, and Xiaodong Zhang. A case study for ray tracing cores: Performance insights with breadth-first search and triangle counting in graphs. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 9(2):1–25, 2025.

[234] Dong Xie, Feifei Li, and Jeff M. Phillips. Distributed trajectory similarity search. *Proceedings of the VLDB Endowment*, 10(11):1478–1489, Aug. 2017.

[235] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient in-memory spatial analytics. In *Proceedings of the 2016 international conference on management of data*, pages 1071–1085, 2016.

[236] Dingqi Yao, Gao Cong, Chengliang Zhang, and Jingjing Bi. Computing trajectory similarity in linear time: A generic seed-guided neural metric learning approach. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1358–1369, Macao, China, 2019. IEEE.

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

[238] Simin You, Jianting Zhang, and Le Gruenwald. Large-scale spatial join query processing in cloud. In *2015 31st IEEE international conference on data engineering workshops*, pages 34–41. IEEE, 2015.

[239] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 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*, pages 1–4, 2015.

[240] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. Spatial data management in Apache Spark: the GeoSpark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.

[241] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment*, 6(10):817–828, 2013.

[242] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283. IEEE, 2016.

[243] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A survey of autonomous driving: Common practices and emerging technologies. *IEEE access*, 8:58443–58469, 2020.

[244] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freire. Gpu rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment*, 11(3):352–365, 2017.

[245] Fatemeh Zardbani, Nikos Mamoulis, Stratos Idreos, and Panagiotis Karras. Adaptive indexing of objects with spatial extent. *Proceedings of the VLDB Endowment*, 16(9):2248–2260, 2023.

[246] Dejun Zhang, Fazhi He, Soonhung Han, Lu Zou, Yiqi Wu, and Yilin Chen. An efficient approach to directly compute the exact hausdorff distance for 3d point sets. *Integrated Computer-Aided Engineering*, 24(3):261–277, 2017.

[247] Dongxiang Zhang, Zimu Chang, Shuangjie Wu, Yuan Yuan, Kian-Lee Tan, and Gang Chen. Continuous trajectory similarity search for online outlier detection. *IEEE Transactions on Knowledge and Data Engineering*, 34(10):4690–4704, Oct. 2022.

[248] Hongrui Zhang, Yunan Zhang, and Hung-Wei Tseng. Rtspmspm: Harnessing ray tracing for efficient sparse matrix computations. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, pages 359–373, 2025.

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

[250] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[251] Songnian Zhang, Suprio Ray, Rongxing Lu, and Yandong Zheng. Sprig: A learned spatial index for range and knn queries. In *17th International Symposium on Spatial and Temporal Databases*, pages 96–105, 2021.

[252] Shiwei Zhao, Zhengshou Lai, and Jidong Zhao. Leveraging ray tracing cores for particle-based simulations on gpus. *International Journal for Numerical Methods in Engineering*, 124(3):696–713, 2023.

[253] Baihua Zheng, Lisi Weng, Xiaofeng Zhao, Kai Zeng, Xiaofang Zhou, and Christian S. Jensen. Repose: Distributed top-k trajectory similarity search with local reference point tries. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 708–719, Chania, Greece, 2021. IEEE.

[254] Yu Zheng. Trajectory data mining: An overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):1–41, 2015.

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

[256] Silin Zhou, Shuo Shang, Lisi Chen, Christian S. Jensen, and Panos Kalnis. RED: Effective trajectory representation learning with comprehensive information. *arXiv preprint arXiv:2411.15096*, Nov. 2024. Accepted by VLDB 2025.

[257] Xingquan Zhu, Jianping Fan, Ahmed K. Elmagarmid, and Walid G. Aref. Hierarchical video summarization for medical data. In Minerva M. Yeung, Chung-Sheng Li, and Rainer W. Lienhart, editors, *Storage and Retrieval for Media Databases 2002*, volume 4676 of *Proceedings of SPIE*, San Jose, CA, USA, 2002. SPIE. Contact: {zhuxq,ake,aref}@cs.purdue.edu; jfan@uncc.edu.

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