RESEARCH-ARTICLE

StreamingRT: Stream KNN Join with Ray Tracing Core

SHIXI YANG, Fudan University, Shanghai, China

KAI ZHANG, Fudan University, Shanghai, China

ZHIGANG ZHAO, Qilu University of Technology, Jinan, Shandong, China

CHUNXIAO WANG, Qilu University of Technology, Jinan, Shandong, China

ZHENGYING HE, Fudan University, Shanghai, China

YINAN JING, Fudan University, Shanghai, China

View all

Published: 10 November 2025

CIKM '25: The 34th ACM International Conference on Information and Knowledge Management
November 10 - 14, 2025
Seoul, Republic of Korea

Conference Sponsors:
SIGWEB
SIGIR

https://doi.org/10.1145/3746252.3761409
ISBN: 9798400720406

StreamingRT: Stream KNN Join with Ray Tracing Core

Shixi Yang
sxyang22@m.fudan.edu.cn
Fudan University
Shanghai, China

Kai Zhang*
zhangk@fudan.edu.cn
Fudan University
Shanghai, China

Zhigang Zhao
zhaozg@sdas.org
Shandong Computer Science Center (National Supercomputing Center in Jinan)
Jinan, Shandong, China

Chunxiao Wang
wangchx@sdas.org
Shandong Computer Science Center (National Supercomputing Center in Jinan)
Jinan, Shandong, China

Zhenying He
zhenying@fudan.edu.cn
Fudan University
Shanghai, China

Yinan Jing
jingyn@fudan.edu.cn
Fudan University
Shanghai, China

X. Sean Wang
xywangCS@fudan.edu.cn
Fudan University
Shanghai, China

Abstract

Efficient processing of k-nearest neighbor (kNN) join operations on streaming data is critical for applications in location-aware services, recommendation systems, and spatial analytics. To serve users in real time, these applications generally require a high-performance kNN join on continuously changing streaming data. This paper introduces StreamingRT, a framework that leverages ray tracing (RT) cores in GPUs to accelerate stream kNN joins in 3D space. By modeling stream data into large primitives and transferring queries into short rays, StreamingRT transforms the kNN join problem into an efficient ray tracing task. To address the ray tracing index updating overhead on stream data, we propose two key techniques, i.e., boundary-extended point partitioning and query-driven BVH lazy updating. Moreover, we also adopt multi-BVH coprocessing and CPU-GPU pipelining to improve performance. These techniques enable efficient stream kNN join on ray tracing cores, delivering unprecedented performance improvement. Experimental evaluations show that StreamingRT can achieve up to 2.2× and 5.8× speedup over the state-of-the-art approach on RT cores and CUDA cores, respectively.

CCS Concepts

• **Computing methodologies** → **Vector / streaming algorithms**; **Ray tracing**; • **Information systems** → **Join algorithms**.

Keywords

stream data processing; kNN join; ray tracing

ACM Reference Format:
Shixi Yang, Kai Zhang, Zhigang Zhao, Chunxiao Wang, Zhenying He, Yinan Jing, and X. Sean Wang. 2025. StreamingRT: Stream KNN Join with Ray Tracing Core. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM '25), November 10–14, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3746252.3761409

*Corresponding Author. This work was supported by the Project of Key R&D Program of Shandong Province (Grant No. 2024CXGC010113).

CIKM '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2040-6/2025/11
https://doi.org/10.1145/3746252.3761409

1 Introduction

The k-nearest neighbor (kNN) join is a fundamental operation in spatial data processing, widely used in applications that require proximity-based data association. Given two datasets R and S, the kNN join operation finds, for each tuple in R, its k-nearest neighbors in S. This operation is essential in various domains, including geographic information systems, location-based recommendation engines, and spatial database queries [12, 19, 31, 33]. For example, in location-aware social networking, kNN join can be used to find the nearest users for friend recommendations. Similarly, in location-based advertising, it enables businesses to target users with relevant promotions based on their proximity to specific stores or points of interest.

In real-time scenarios, this operation extends to stream kNN join, where both R and S are continuously incoming data streams. This is particularly relevant in applications such as recommender systems [3], real-time location-based services [28] and anomaly detection [42]. A prominent example is the taxi-hailing service [18], where a stream of customer requests must be continuously matched with the closest available taxis in real time. The effectiveness of such applications hinges on the ability to process spatial data streams efficiently; otherwise, delayed or irrelevant responses can degrade the quality of service. Stream kNN join operations pose significant challenges due to the dual requirements of high-speed data ingestion and the computational complexity of spatial operations, such as distance calculations and neighbor ranking [39, 43]. Moreover, these operations must adhere to strict latency constraints to ensure that users receive responses tailored to their location and preferences in real time. Failure to meet these constraints can result in increased waiting times, inefficient taxi assignments, and a suboptimal user experience.

The stream kNN join operation combines the complexities of both the join process and kNN query execution while contending with the dynamic and often unpredictable nature of data streams. These challenges become particularly pronounced in high-velocity,

high-volume environments, where maintaining low latency is critical. Traditional CPU-based processing usually uses some special indexes, such as R-tree [41], which are inadequate for such workloads due to limited parallelism and massive updates. To address these challenges, distributed processing frameworks for stream kNN join, such as the approach described in [30], partition data streams and distribute kNN join tasks across multiple machines. In addition, GPU-based solutions [11, 40] have been designed to accelerate stream kNN join with their massive parallelism and thousands of CUDA cores.

Modern desktop and server-class GPUs are equipped with ray tracing (RT) cores designed to accelerate real-time rendering of complex 3D scenes. Beyond graphics, NVIDIA has extended RT core functionality through APIs that enable custom objects and ray casting within a 3D space, unlocking new possibilities for high-performance geometric computations. Recent studies have demonstrated the potential of RT cores in accelerating various data processing tasks, including point location [22, 36], nearest neighbor search [7, 27, 45], DBSCAN clustering [26], database indexing [13, 20], and range minimum queries [21]. These applications leverage RT cores by representing data points as geometric primitives, such as triangles or spheres, which are structured within a Bounding Volume Hierarchy (BVH) for efficient spatial queries. Queries are then mapped to rays traversing the BVH, where intersections indicate relevant data points. This approach reduces computational overhead, avoiding unnecessary calculations. By reframing traditional data processing challenges as ray tracing problems, RT cores enable substantial performance gains, particularly in previously computationally prohibitive tasks. Their ability to accelerate nearest neighbor search, spatial analytics, and real-time query processing expands the role of GPUs beyond rendering, demonstrating their potential for broader applications in high-performance computing.

RTNN [45] is currently the state-of-the-art algorithm that leverages RT cores for kNN search, but it is designed only for static data. RTNN analyzes the static dataset and constructs primitives of different sizes for different points, and casts short rays from query points. From the intersected points, the k nearest neighbors are selected according to the relative distance. However, through experiments, we find that RTNN is inefficient in performing kNN on stream data. In stream kNN, queries are performed on each sliding window where data changes dynamically. When performing RTNN on stream data, the data needs to be analyzed for each window, and the entire BVH also has to be rebuilt. These analyses and index updating operations may take more than half of the overall execution time, resulting in suboptimal performance and ray tracing computational resource underutilization.

In this paper, we present a novel approach, StreamingRT, which effectively accelerate stream kNN join with ray tracing cores. StreamingRT effectively addresses the high overhead caused by frequent data structure updates in streaming data scenarios. It is primarily based on two key techniques: boundary-extended point partitioning and BVH lazy updating. We first partition the entire space into small regions and build an independent BVH for each region, which significantly reduces the cost of index reconstruction compared to updating a single large BVH for each window. Then, based on dynamically arriving queries and data points from the stream, we dynamically determine whether a BVH update is necessary for a given query. This adaptive mechanism helps to minimize unnecessary computational overhead and significantly enhances stream kNN query performance. By incorporating the optimizations, our approach provides a highly efficient solution for stream kNN join.

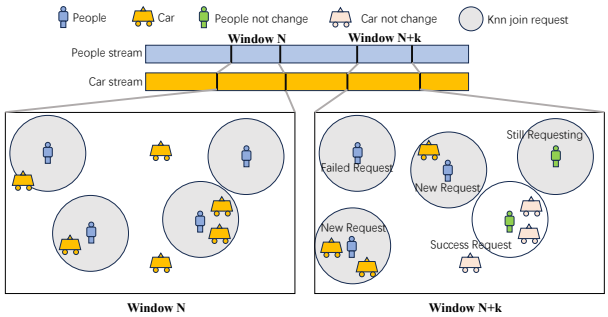


Figure 1: Example of Stream kNN Join

The contributions of this paper are as follows:

- We propose StreamingRT, a solution that efficiently accelerates stream kNN join with ray tracing cores.
- We propose two techniques to address the data updating overhead in stream processing with RT cores: boundary-extended point partitioning and BVH lazy updating.
- We evaluate its performance under diverse workloads and parameters. Our experimental results show that StreamingRT can achieve 2.2× and 5.8× speedup over the state-of-the-art approach on RT cores and CUDA cores, respectively.

To the best of our knowledge, StreamingRT is the first work to study stream kNN join on ray tracing cores. We believe the proposed techniques would benefit other stream processing jobs with RT cores.

2 Background

2.1 Problem Definition of Stream kNN Join

kNN search. Given the tuple t and the tuple set S, the kNN of t over S is denoted as $kNN(t, S)$ and defined as follows: $kNN(t, S)$ is a subset of S with a maximum size of k, where the distance of all other elements of S to the tuple t is greater than or equal to the distance of each element of $kNN(t, S)$ to the tuple t.

$$kNN(t, S) \subset S, |kNN(t, S)| = min(k, |S|)$$
$$\forall s_i \in kNN(t, S), s_j \in S - kNN(t, S) \rightarrow |t, s_i| \leq |t, s_j|$$

kNN join. Given the tuple set R and the tuple set S, the kNN of R over S is denoted as $R \bowtie_{kNN} S$ and defined as follows: $R \bowtie_{kNN} S$ is a subset of S and each point in R finds its kNN in S and return all the (r, s) pairs where point s in S is one of k nearest neighbor of point r. The result set contains $k \times n(R)$ pairs.

Stream kNN join. Given two streams R and S, the kNN of R over S is denoted as $R \bowtie_{kNN} S$. For a specific time t, $R \bowtie_{kNN} S$ generates the result of $W_r(t) \bowtie W_s(t)$ where $W_r(t)$ is the sliding window of R and $W_s(t)$ is the sliding window of S at the time of t.

In a stream kNN join process, the goal is to continuously process incoming data streams and find the k-nearest neighbors (kNN) for each new data point in real-time. This means that, for each window of query stream, we need to locate its k-closest points from the corresponding point stream window, which is particularly

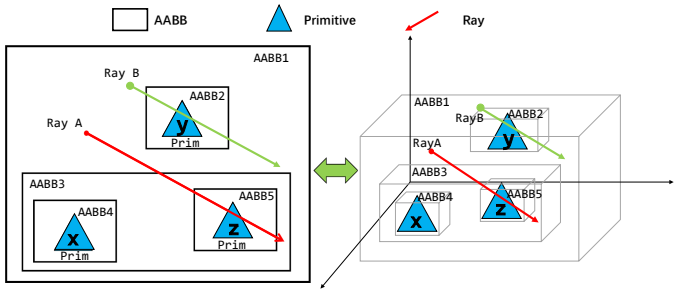


Figure 2: Example of ray tracing

useful for applications like anomaly detection, recommendation systems, and traffic monitoring. For example, in Figure 1, consider a live stream of GPS locations from taxi passengers as the query stream, alongside another stream of GPS locations from various taxis. At each time step with a specific window, a stream kNN join operation identifies the k nearest taxis for each passenger, enabling prioritized pickups and optimized scheduling decisions based on spatial distance. However, executing this process efficiently in real-time presents several challenges. First, high throughput demands can be challenging , requiring the system to handle a high query rate. Second, as the data stream changes, it is also difficult to maintain search index and in time updating. When the query and point datasets in a stream kNN join are the same, the operation is referred to as a stream self kNN join. In this case, each incoming data point is compared with other points within the same stream to identify its k-nearest neighbors. This self-join variant is particularly useful for identifying similar items or detecting anomalies within a single data stream, as it allows for real-time proximity analysis among points from the same source.

2.2 Background of Ray Tracing

Ray tracing is a computationally demanding rendering technique that accurately models light interactions within a scene, capturing effects such as reflections, refractions, and absorption [29]. To enhance performance, modern GPUs incorporate dedicated RT cores—specialized hardware designed to accelerate ray tracing computations. NVIDIA introduced its first-generation RT cores with the Turing architecture, achieving up to a 10× speedup over software-based implementations [29]. Since then, RT cores have been integral to NVIDIA's Turing, Ampere, and Ada architectures. Beyond NVIDIA, other manufacturers have also adopted hardware-accelerated ray tracing, with AMD integrating support in GPUs like the Radeon RX 7800 XT. This broad adoption underscores the growing importance of ray tracing in modern graphics processing.

At the heart of ray tracing is the computation of intersections between rays and a 3D scene, which is typically represented as a collection of geometric primitives such as triangles and spheres. These intersection tests are the primary factor influencing rendering time [35] and are significantly accelerated by RT cores. To optimize these computations, the scene's primitives are partitioned and enclosed within bounding volumes, most commonly Axis-Aligned Bounding Boxes (AABBs). Primitives directly contained within AABBs are referred to as leaf AABBs. These bounding volumes are further organized hierarchically into a tree structure known as the BVH. Figure 2 illustrates a scene with three triangles, where leaf

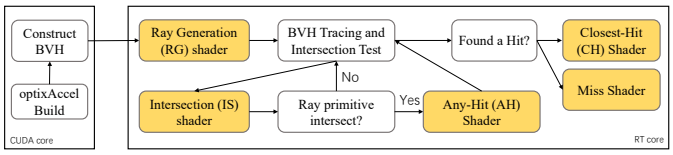


Figure 3: The simplified programming model of OptiX

AABBs enclose individual primitives, while larger AABBs encapsulate smaller ones to facilitate efficient traversal.

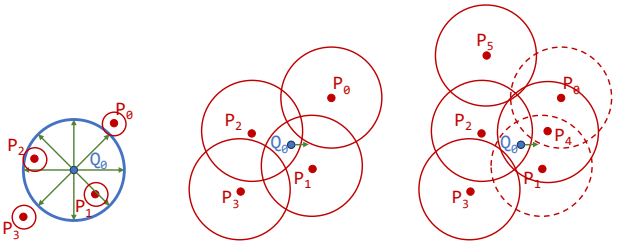
A ray is defined by an origin point O and a direction vector D, parameterized as:

$$R(t) = O + tD$$

where the parameter t is typically constrained to an interval, $t \in [t_{min}, t_{max}]$, to define a finite segment of the ray. Once a ray is cast, it traverses the BVH tree, with intersection tests against geometric primitives efficiently accelerated by RT cores. During traversal, if a ray intersects an AABB, further intersection tests are performed on the primitives or AABBs enclosed within it. Conversely, if a ray does not intersect an AABB, all enclosed primitives can be safely skipped, as they are guaranteed not to intersect the ray. This hierarchical pruning significantly reduces the computational workload of ray tracing. As illustrated in Figure 2, Ray A bypasses AABB 2, allowing primitive y to be skipped entirely. In contrast, since Ray A intersects AABB 3, further intersection tests must be performed on its child nodes, AABB 4 and AABB 5 and finally intersects with primitive z. Similarly, Ray B intersects AABB 2, bypasses AABB 3 and its enclosed AABB 4 and AABB 5, and eventually intersects with primitive y.

Figure 3 illustrates the programming model of Nvidia OptiX. The OptiX framework offers a flexible, shader-based programming model that enables developers to define custom behaviors at various stages of the ray traversal process. This level of control allows users to tailor the ray tracing pipeline for a wide range of applications beyond traditional rendering tasks. During ray traversal, the Intersection shader is triggered whenever a ray intersects a leaf node in the AABB hierarchy. This shader is responsible for performing the critical ray-primitive intersection test, determining whether the ray intersects with any geometric primitives in the 3D space. If an intersection is detected, the user-defined Any-Hit shader may be invoked to process the intersection data. This shader provides the flexibility to handle rays that intersect multiple primitives, such as by determining transparency or conditionally accepting intersections. In addition to the Any-Hit shader, the OptiX model supports other programmable shaders that further enhance control over the ray tracing process. The Closest-Hit shader is executed when the ray intersects the closest primitive along its path, providing precise control over how intersection data is processed, such as shading or recording the result of the nearest object. Conversely, the Miss shader is called when a ray does not intersect any primitives, enabling users to define behavior for rays that miss all objects, such as assigning a background color or other fallback logic.

By leveraging this rich shader-based model, RT cores are not only utilized to accelerate geometric intersection tests, including movie special effects [4], advanced computer-aided manufacturing [6], and video games [10], but also enable complex, customizable operations across various computational tasks. This versatility makes



(a) Direct Transformation (b) StreamingRT Transformation (c) With Streaming Point

Figure 4: Transforming kNN join as a ray tracing job. (a) Direct transformation into a ray tracing problem. (b) Optimized ray tracing approach using short rays and spherical primitives. (c) Stream processing with dynamic updates.

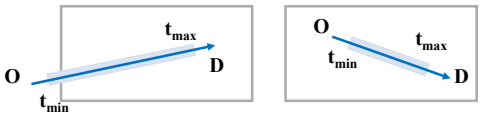
OptiX a powerful tool for both rendering and non-rendering applications, such as physics simulations, data processing, and spatial queries, all of which benefit from the highly parallelized ray tracing capabilities provided by Nvidia's hardware. Recent studies have increasingly explored the use of RT cores to accelerate rendering workloads, such as graph drawing [44], rendering transparent objects [37], dexel modeling [15], and particle movement [1, 2]. Beyond graphics, several works have investigated leveraging RT cores for data processing tasks. These include point location [22, 36], nearest neighbor search [27, 45], DBSCAN clustering [26], database indexing [13], range minimum queries [20] and efficient outlier detection [38].

2.3 Challenges of Accelerating Stream KNN Join with RT Cores

Building an efficient transformation of stream kNN join into a ray tracing task involves two key aspects: 1) mapping data and query into primitives and rays, and 2) consistently updating the data set and the index on stream data.

For data mapping, Figure 4(a) demonstrates a naive transformation where each data point is represented by a sphere with a small radius, while rays are cast from the query point Q_0 in multiple directions with a predefined length r, testing for intersections with points P_0, P_1, P_2, and P_3. This leads to at least $\frac{2\pi}{\arcsin(\frac{r}{R})}$ ray casting where r is the primitive radius and R is ray length [38]. This approach casts an excessive amount of rays in the 3D space, while the number of ray tracing cores is limited, resulting in unnecessary resource consumption. RTNN[45] is a pioneering work that transforms the kNN join into a ray tracing job. RTNN performs a grid-based static analysis on the dataset and generates primitives of different sizes for each point, and a short ray is cast from each query point to reduce intersection tests. This approach improves kNN performance because it launches much fewer rays and reduces the intersection count and this method is also used by other works such as RT-KNNS [27].

Through experiments, we find that the approach of RTNN suffers from inefficiency in handling stream data. First, analyzing the data set and setting an appropriate radius for each primitive incurs a huge overhead. Second, BVH reconstruction across streaming windows also introduces significant costs. For the NBody dataset[32], it takes about 26% of the overall execution time on data analysis and optimization, while the BVH construction takes around 24% of the time [45]. This is extremely inefficient in processing streaming data, as data analysis and BVH reconstruction are required for every window, which means the utilization of the RT core is less than 40%.



Condition 1 Condition 2

Figure 5: Two conditions for ray-AABB intersection

3 Design and Implementation

3.1 Stream KNN Join as a Ray Tracing Job

In order to reduce the number of intersection tests and unnecessary computation on stream data, we propose StreamingRT, which maps stream kNN join into ray tracing jobs with lightweight updating. Figure 4(b) demonstrates our optimized transformation of the kNN join using RT cores. The transformation procedure creates a fixed r-radius sphere around each point. Instead of casting rays in all directions, short rays of length `FLOAT_MIN` are then cast from all queries to traverse the BVH. For example, in Figure 4(b), the query point Q_0 emits a short ray, only testing for intersections with nearby primitives. Rays intersect with the spheres representing P_1 and P_2, marking them as potential nearest neighbors, while P_0 and P_3 do not intersect the rays and are excluded from the kNN set. Figure 4(c) illustrates how StreamingRT works on streaming data, where two points, P_0 and P_1, are removed from the dataset, and two new points, P_4 and P_5, are added.

A ray is considered to intersect an AABB under two specific conditions [45]. First, the ray physically intersects one or more of the AABB's bounds, shown in Figure 5 condition 1. Second, the ray's origin lies within the AABB, even if the ray does not intersect any of the bounding faces directly, shown in Figure 5 condition 2. These conditions serve as the foundation for efficient traversal and pruning in BVH. StreamingRT leverages condition 2 for its short ray intersection mechanism. When a ray originating from Q_0 intersects a leaf AABB, the Intersection Shader in OptiX is invoked. The shader checks whether Q_0 intersects the sphere enclosed by the AABB by computing the Euclidean distance between Q_0 and the sphere's center point. If the distance is less than r, the ray cast from Q_0 is considered to have intersected the sphere and the intersected point will be considered as one of the neighbors.

The mechanism adopted by StreamingRT brings several advantages: it reduces unnecessary computations by focusing the ray casting on specific regions of interest, improving overall scalability by allowing the BVH (Bounding Volume Hierarchy) structure to efficiently exclude distant points from consideration. Moreover, the reduced number of intersection tests, combined with the use of RT cores, significantly accelerates the kNN join on streaming data. To meet the demands of high throughput and frequent updates, this paper proposes two key optimization techniques to improve algorithm performance. First, a boundary-extended point partitioning strategy is employed to divide points in 3D space into different regions, with a separate BVH tree constructed for each region. This approach effectively reduces the overhead of BVH reconstruction and avoids the cost of rebuilding a single large BVH for all points.

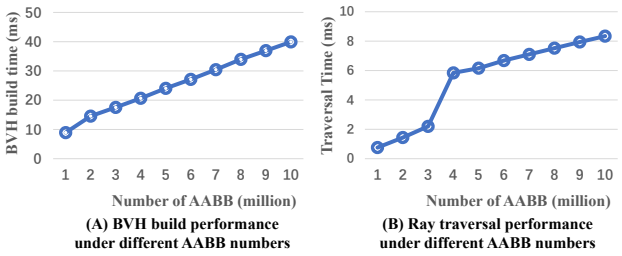


Figure 6: Variation of BVH Build Time and Ray Traversal Time with the Number of AABBs.

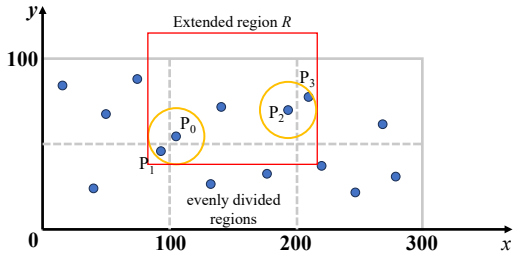


Figure 7: Illustration of the Partitioning Method: Each partition is evenly divided based on the total number of partitions, with boundaries extended by the search radius to ensure accuracy.

Second, a BVH lazy updating scheme is utilized to reduce the redundant computations during dynamic data updates. In addition, a multi-BVH coprocessing and a CPU-GPU pipelining mechanism are used to improve performance and fully exploit heterogeneous computational resources. In the following parts of this section, we provide detailed descriptions of these three techniques.

3.2 Boundary-Extended Point Partitioning

In ray tracing, constructing acceleration structures (e.g., BVH) is a critical yet time-consuming process. The time required to build a BVH tree is closely related to the number of AABBs present in the scene. For instance, in Figure 6, constructing a BVH tree for five million points takes approximately 25 milliseconds, while the time increases to around 40 milliseconds for ten million points. As the number of points increases, the time taken for BVH construction escalates significantly, and the ray tracing time also increases, leading to potential performance degradation in overall ray tracing processing on stream data. To address this challenge, we propose a boundary-extended point partitioning strategy. This method involves dividing the point set within the scene into multiple smaller subsets, effectively reducing the overhead of globally reconstructing the BVH tree during point flow.

In detail, we implement point partitioning using a grid-based scheme along the x, y, and z axes in a three-dimensional space shown in algorithm 1 line 1 to line 9. Each axis is divided independently based on predefined partition counts. For example, setting the partition counts to $p_x = 3$, $p_y = 2$, and $p_z = 1$ means dividing the x-axis into three partitions, the y-axis into two partitions, and keeping the z-axis as a single partition. Given a range of 0 to 300

Algorithm 1 Boundary-extended Point Partitioning

Require: Point $P(x, y, z)$, max range R_x, R_y, R_z, partition counts N_x, N_y, N_z, search radius r, 3D partition list L

1: Compute cell sizes:
2: $C_x \leftarrow R_x/N_x, C_y \leftarrow R_y/N_y, C_z \leftarrow R_z/N_z$
3: Compute partition index:
4: $I_x \leftarrow \lfloor x/C_x \rfloor, I_y \leftarrow \lfloor y/C_y \rfloor, I_z \leftarrow \lfloor z/C_z \rfloor$
5: Add P to its partition L_i according to index
6: Compute cell boundaries:
7: $B_x^{min} \leftarrow I_x \times C_x, B_x^{max} \leftarrow (I_x + 1) \times C_x$
8: $B_y^{min} \leftarrow I_y \times C_y, B_y^{max} \leftarrow (I_y + 1) \times C_y$
9: $B_z^{min} \leftarrow I_z \times C_z, B_z^{max} \leftarrow (I_z + 1) \times C_z$
10: Compute distance from P to 6 boundaries:
11: $N \leftarrow$ number of distance $< r$
12: **if** $N = 1$ **then**
13: Add P to 1 face neighbor partition L_f accordingly
14: **else if** $N = 2$ **then**
15: Add P to 2 face and 1 edge neighbor partitions L_e accordingly
16: **else if** $N = 3$ **then**
17: Add P to 3 face, 3 edge and 1 vertex neighbor partitions L_v accordingly
18: **else**
19: No additional partitions added
20: **end if**

for x and 0 to 100 for both y and z, the x-axis partitions span 0–100, 100–200, and 200–300, while the y-axis partitions cover 0–50 and 50–100. The z-axis remains unchanged. This results in a total of $3 \times 2 \times 1 = 6$ partitions. The gray dashed lines in Figure 7 illustrate the partitioning along the x- and y-axes.

A key challenge in partitioning is handling boundary conditions. In our intersection tests, we determine whether the ray origin of each query lies within a spherical primitive. Queries near partition boundaries require special treatment to ensure accuracy. Specifically, if the maximum radius r of a spherical primitive is 5, we expand each partition's boundaries by this value in both directions. This adjustment ensures that points near the boundary are correctly included in the corresponding BVH trees. For instance, P_1 is a neighbor of P_0 and P_3 is a neighbor of P_2, but they are not in the partition directly, so we need to extend the boundaries by r and include P_1 and P_3 to extended region, as illustrated by the red solid box in Figure 7. This approach ensures that points near boundaries are covered by multiple BVH trees, eliminating the need for boundary queries to traverse multiple trees while maintaining computational accuracy and improving traversal efficiency.

To determine whether a boundary point belongs to other partitions, we calculate the distance between the point's coordinates and the six boundary planes of its enclosing partitioning cube based on the search radius r which is shown in algorithm 1 line 10 to line 20. If the distance to a boundary plane is less than r, the point is added to the adjacent cube across that plane. If the point is within r of a single boundary, it is added to one face neighbor. If it is within r of two boundaries, it is added to two face neighbors and one edge neighbor. If it is within r of three boundaries, it is added to three face neighbors, three edge neighbors, and one vertex neighbor. Once the space is partitioned, each data point is assigned to a specific region

based on its coordinates. We do direct partitioning for query points using its calculated index because queries don't need extended and queries within each partition are also sorted in Z-order to enhance ray coherence, which improves memory access locality and boosts performance. During the search process, we construct individual BVH trees for each partition, forming a BVH forest. Since each tree contains only a subset of the dataset, this partitioned approach dramatically reduces traversal time and improve ray coherence compared to searching through a single large BVH tree.

This approach not only optimizes the BVH building process but also alleviates performance bottlenecks caused by an increasing number of data points, thereby providing a more efficient solution for real-time ray tracing in dynamic scenes. In addition, point partitioning brings another significant advantage: it enhances ray coherence. Improving ray coherence significantly reduces memory access inconsistencies during tree traversal, increases opportunities for memory coalescing, lowers data access latency, and improves cache hit rates. This optimization further enhances computational efficiency in the ray tracing process.

3.3 BVH Lazy Updating

Our optimization strategy combines a lazy updating mechanism to enhance efficiency in handling nearest neighbor searches across dynamic partitions. The lazy updating approach addresses the issue of unnecessary BVH rebuilds that occur due to independent movements of queries and data points across partitions and data locality across sliding window. Instead of rebuilding the BVH each time a partition updates, we trigger a rebuild only when both conditions are met: data points in a partition change, and a query accesses this updated partition shown in algorithm 2 line 4 to line 16. For instance, if data points shift in a particular partition but no query accesses it, the BVH remains unaltered, conserving computational resources. Similarly, if a query changes its position without the data points in a partition being modified, the BVH for that partition is left as is. This lazy updating approach thus minimizes unnecessary BVH rebuilds, allowing queries to proceed through previous traversal structures and reducing time and computational load.

We first do extended partitioning for incomming point window, and direct partitioning for query window. For each partition, a previous point set is maintained for comparison. The previous set and the BVH are in a one-to-one correspondence, so it only update when BVH is updated, which is shown in line 15. To efficiently compare the differences between previous point set and current point set, we use sets to store the points in each window. Since the coordinates are floating-point numbers, to avoid errors caused by floating-point hashing, we encode the coordinates as strings. For instance, we represent the value of each axis as an 8-character string, where the integer part occupies the first 3 characters and the fractional part up to 5 decimal places occupies the remaining 5 characters. Padding is applied to both parts as needed. This encoding scheme can be adjusted according to specific application needs and data distribution. Finally, the three encoded axis values are concatenated into a 24-character string, which is used as the key for storage.

Algorithm 2 Partitioned Lazy Update Decision Algorithm

Require: Previous point window set list P_{prev}, current point window W_{cur}, current query window Q_{cur}

1: Extended partition W_{cur} into $\{W^1_{\text{cur}}, W^2_{\text{cur}}, \ldots, W^N_{\text{cur}}\}$
2: Direct partition Q_{cur} into $\{Q^1_{\text{cur}}, Q^2_{\text{cur}}, \ldots, Q^N_{\text{cur}}\}$
3: **for all** W^i_{cur} in $\{W^1_{\text{cur}}, W^2_{\text{cur}}, \ldots, W^N_{\text{cur}}\}$ **do**
4: **if** Q^i_{cur} is empty **then**
5: **continue**
6: **end if**
7: Initialize empty set P^i_{cur}
8: **for all** $p \in W^i_{\text{cur}}$ **do**
9: $key \leftarrow \text{EncodePoint}(p)$
10: Add key to P^i_{cur}
11: **end for**
12: **if** $P^i_{\text{cur}} = P^i_{\text{prev}}$ **then**
13: (Perform query without rebuilding BVH)
14: **else**
15: $P^i_{\text{prev}} \leftarrow P^i_{\text{cur}}$
16: (Rebuild BVH structure and perform query)
17: **end if**
18: **end for**

3.4 Implementation and Optimization

3.4.1 Multi-BVH coprocessing In our ray tracing approach, we assemble the traversal of all queries into a single kernel launch rather than using multiple CUDA streams. This decision is based on the fact that the RT cores are typically operating at maximum capacity, making the use of multiple streams effectively serial in nature. Additionally, introducing multiple streams would lead to extra CPU scheduling overhead and context-switching costs between streams, which could increase tracing time and reduce overall performance.

OptiX inherently supports a single-ray programming model, where each `optixTrace` call initiates a single ray and allows for flexible use of multiple traversable handles for different acceleration structures (AS), which can be chosen per ray in a single launch. This flexibility allows us to set up a BVH forest where each partition has its own BVH. We organize queries for each partition in advance and, during tracing, select the appropriate BVH for each query based on its identifier. By consolidating queries across different partitions into a single launch, we maximize the GPU's workload efficiency, reducing kernel launch overhead while minimizing unnecessary stream-switching and resource contention. OptiX does allow for parallel `optixLaunch` calls in separate CUDA streams. However, if the launch dimensions are large enough to saturate the GPU, there is minimal advantage to running multiple launches in parallel[9]. Each launch would share the same hardware units, and the asynchronous processing could introduce substantial overhead. For our approach, a single `optixLaunch` covering all queries from different partitions in one call not only minimizes performance losses from stream-switching but also simplifies the pipeline by eliminating the need for separate launch parameters and output buffers across streams. Each partition is assigned a unique BVH tree, and every BVH is associated with an identifier. During the traversal process, each ray retains the identifier of its corresponding BVH, allowing it to selectively traverse only the BVH tree with the matching identifier. By ensuring that each ray operates independently on its assigned

BVH, this design minimizes contention for shared resources and reduces the overhead typically associated with managing multiple kernel launches.

3.4.2 CPU-GPU Pipeline In StreamingRT, the CPU is responsible for extended point partitioning and the BVH lazy update checking mechanism. For each window, the CPU first partitions the data points using an extended partitioning algorithm and directly assigns queries to their corresponding partitions. After partitioning, the memory layout of the points is reorganized, and the queries within each partition are sorted in Z-order. Once this process is complete, all data is transferred to the GPU. The GPU is responsible for BVH rebuilding and BVH traversal. It updates the BVH structures based on the lazy update results provided by the CPU, and then performs traversal to compute the kNN join results.

In StreamingRT, we build a CPU-GPU pipelining strategy to accelerate the overall processing workflow. With the pipelining, when the GPU is performing the kNN on the ith window with RT cores, the CPU is performing data partitioning and checking lazy update in parallel. This parallelism ensures that the CPU continuously prepares partition data in advance, synchronizing with the GPU's ray tracing completion. When a new `optixLaunch` occurs, the necessary partitioned data is already preprocessed, reducing setup times and preventing GPU idle time.

4 Experiments

4.1 Experimental Setup

Dataset In our experiments, we use four 3D datasets with distinct characteristics to evaluate the performance of the stream kNN join. The Millennium Simulation Project [24, 25] provides a cosmological N-body simulation dataset [32] containing 10M points, with coordinates ranging from 0 to 500 along each of the three axes. The Buddha and Dragon [5] model from the Stanford 3D Scanning Repository [17] has 4.6M and 2.7M points respectively and most points coordinate in [-1, 1] along the three axes. These three datasets are three-dimensional and used for stream self kNN join: the Millennium dataset contains numerous dense clusters representing galactic structures, where points are tightly packed within clusters but widely spaced between clusters. The Buddha and Dragon model has highly dense points distributed in a relatively small space, capturing the detailed contours of the object. The fourth dataset, the NYC Taxi Dataset of 2017 with about 3M points [34] ranging from [0, 250] along x and y axes, is used for standard stream kNN join, where passenger pick-up locations serve as queries, and drop-off locations act as points. This dataset is primarily distributed in a two-dimensional xy-plane (latitude and longitude), reflecting New York City's layout. The data points are more uniformly spread due to the nature of citywide travel patterns, introducing randomness across spatial areas. For GPU memory usage, the AABB of a single point requires two 3D floating-point vectors(32B with padding) for representation. For 1 million points, the total memory consumption is approximately 60MB(for inner-nodes and leaf-nodes), which is much smaller than the available GPU memory. Thus, memory capacity is not a limiting factor in this case.

Competitors To evaluate the performance of our approach, we compare it with two state-of-the-art nearest neighbor search methods: RTNN [45, 46] and FRNN [40]. RTNN utilizes NVIDIA's RTCore technology, originally designed for ray tracing in graphics, to accelerate the neighbor search process. By treating search queries as rays that intersect spatial structures, RTNN can efficiently traverse large datasets, leveraging RT core's hardware-accelerated ray-tracing capabilities. It also adopts two optimizations: query scheduling and query partition. This approach allows RTNN to achieve significant speed-ups in query handling by optimizing traversal steps. FRNN, by contrast, operates on a fixed-radius nearest neighbor search principle and is fully optimized for CUDA-based GPU computation and it is also a drop-in replacement for kNN in Pytorch [8]. It divides the space into partitions with a grid cell index, allowing it to focus on a specific radius for each query, which leads to consistent, efficient performance. FRNN's reliance on spatial partitioning and memory caching minimizes memory transfer latency and maximizes throughput, making it particularly well-suited for dense data applications.

Hardware and Software We conduct all experiments on a machine equipped with an Intel(R) Xeon(R) Silver 4316 CPU @ 2.30GHz, 128GB DDR4 DRAM, and an NVIDIA GeForce RTX 4090 GPU featuring 128 RT cores, 16384 CUDA cores, and 24GB VRAM. The operating system is 64-bit Ubuntu Server 20.04 with Linux Kernel 5.15.0-72-generic. Our algorithm is implemented in C++ 17, and the program is compiled using CUDA 12.1 and OptiX 7.5.

4.2 Performance Analysis with Different k Neighbor Counts

This experiment evaluates the scalability of the three algorithms as the number of neighbors k increases from 10 to 50 across four datasets. As shown in Figure 8, the results highlight distinct performance trends among the algorithms. The search radius for NBody and NYTaxi is fixed at 5, while for Buddha and Dragon, it is set to 0.0001, with a window size of one million points and step size is of 20% window size.

StreamingRT consistently achieves the lowest execution times across all datasets, benefiting from its streamlined design, which delegates preprocessing tasks to the CPU and executes ray tracing in a single kernel launch. Unlike RTNN, which relies on multi-stream and multi-kernel execution, StreamingRT minimizes coordination overhead, leading to an average speedup of 1.4× on the NBody dataset and over 2.2× on Buddha and Dragon. FRNN exhibits the highest processing times, primarily due to its fixed-radius search strategy. It constructs grid-based indices for each window, introducing substantial overhead as k increases. This limitation is especially pronounced in the Buddha and Dragon datasets, where FRNN is outperformed by StreamingRT by more than 5.5×. The inefficiencies in FRNN's index-building approach make it less suited for scenarios with varying query parameters.

4.3 Performance Analysis Across Different Window Sizes

Figure 9 presents the processing times of the three algorithms across varying window sizes for four datasets. The search radius is set to 5 for NBody and NYTaxi, while Buddha and Dragon use a radius of 0.0001. The number of neighbors k is fixed at 10. Due to dataset size, we use different window size on different dataset.

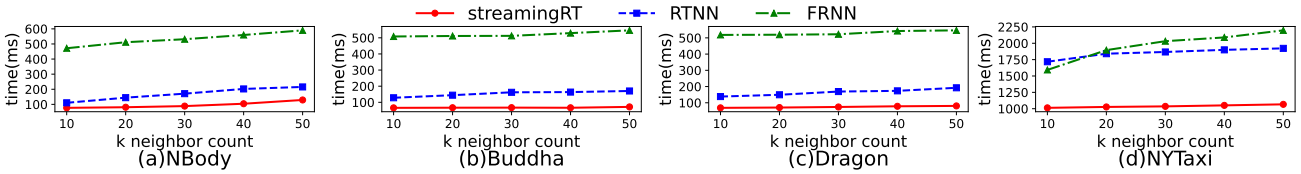


Figure 8: Performance for one window of StreamingRT, RTNN, and FRNN across varying k values.

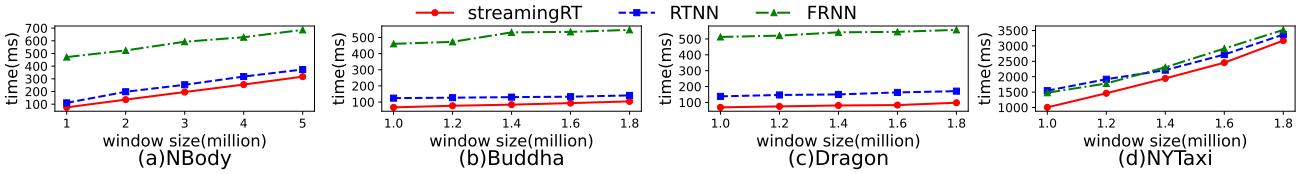


Figure 9: Performance for one window of StreamingRT, RTNN, and FRNN across varying window sizes.

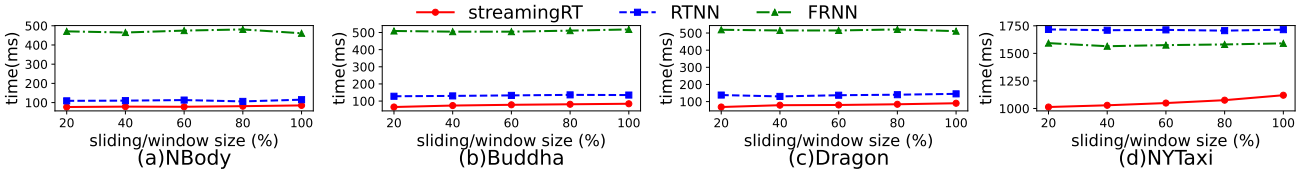


Figure 10: Performance for one window of StreamingRT, RTNN, and FRNN as sliding size varies from 20% to 100% of the window.

For NBody dataset, StreamingRT achieves a speedup of over 1.6× compared to RTNN and more than 5.8× over FRNN. This efficiency arises from its streamlined CPU-GPU pipeline and optimized BVH updates, which minimize redundant computations. RTNN performs well at smaller window sizes but suffers from increasing synchronization and management overhead due to its multi-stream, multi-kernel execution. FRNN exhibits the steepest performance degradation, as its grid-based indexing must be rebuilt for each window, making it inefficient for clustered data.

For the Buddha and Dragon dataset, StreamingRT maintains the lowest processing times, benefiting from its lazy BVH updating and single-kernel execution. StreamingRT is more than 1.8× faster than RTNN and over 5.8× faster than FRNN. For the NYTaxi dataset, StreamingRT continues to outperform the alternatives, and achieves speedups of over 1.6× against RTNN and more than 1.5× over FRNN.

4.4 Performance Analysis with Varying Sliding Sizes

Figure 10 shows the execution time (in milliseconds) of three algorithms as the sliding size varies from 20% to 100% of the window. This experiment evaluates how each algorithm scales with increasing data updates. The experiment is conducted on the four datasets with a search radius of 5 on NBody and 0.0001 on Buddha and Dragon, $k = 10$, and a window size of one million points. The sliding size represents the proportion of points updated within each window.

StreamingRT achieves the lowest execution time across all sliding sizes, demonstrating superior adaptability as updating data volume increases. At the largest sliding size, it is over 1.4 times faster than RTNN and more than 5.6 times faster than FRNN. This efficiency

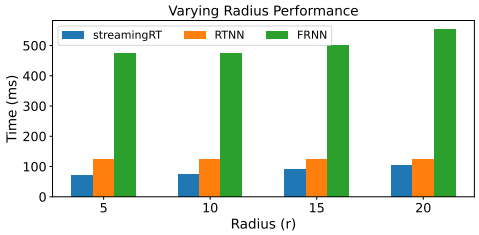


Figure 11: Time performance of StreamingRT, RTNN, and FRNN across varying radius for NBody datasets.

stems from its lazy BVH updates and pipeline optimization, which minimize redundant computations. The relatively shallow increase in its processing time is caused by the data preprocessing on CPU when facing more data updating. Even when performing a full update for each window, StreamingRT still outperforms the other two algorithms. RTNN exhibits a moderate increase in processing time as the sliding size grows. Its reliance on ray tracing hardware provides efficiency but requires optimization at each step, making it less responsive to sliding variations. Since RTNN processes all points within the window regardless of updates, its performance remains relatively stable compared to StreamingRT. FRNN, similarly, shows limited sensitivity to sliding size changes, as it reconstructs grid-based indices for every window. However, this approach incurs significant overhead, leading to the highest execution times among the three methods. The high cost of index rebuilding makes FRNN less suitable for applications requiring dynamic data adaptation.

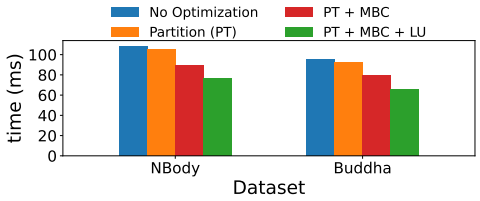


Figure 12: Impact of various optimization techniques on performance across NBody and Buddha datasets.

4.5 Performance Analysis Across Different Radius

Figure 11 presents the processing time for three algorithms as the search radius varies from 5 to 20. We only use NBody dataset because of the limitation of coordination range. This experiment evaluates how increasing the radius affects performance, as a larger radius expands the search space, increasing intersection checks and computational overhead. The experiment is conducted on the NBody dataset with $k = 10$ and a window size of one million points.

For small radii (e.g., $r = 5$), StreamingRT exhibits the best performance, achieving significantly lower processing time than RTNN and FRNN. This advantage arises from its efficient hardware utilization and multi-BVH coprocessing, which allows rapid processing of smaller queries. As the radius grows, StreamingRT maintains relatively stable performance, dynamically adjusting resources to minimize redundant operations and balance the workload. RTNN shows stable performance across different radii, as it determines an optimal search radius globally. This design prevents performance degradation as the specified radius increases, making RTNN robust and efficient. However, for small radii, its preprocessing overhead may outweigh its benefits, leading to slightly higher runtime than StreamingRT. FRNN consistently exhibits the highest processing time, particularly for large radii. Its fixed-radius search structure lacks adaptability, leading to increased computational costs as the search space expands. This limitation makes FRNN less efficient for scenarios requiring flexible search parameters.

4.6 Impact of Optimization Techniques on Performance

Figure 12 illustrates the impact of different optimization techniques on processing time for the NBody and Buddha datasets. The methods include No Optimization, Partitioning (PT), PT with Multi-BVH coprocessing (PT+MBC), and PT+MBC with Lazy Updating and CPU-GPU pipeline (PT+MBC+LU), evaluating their effectiveness in reducing computation overhead. No Optimization serves as a baseline, showing high processing times, particularly for the Buddha dataset due to its dense distribution. Partitioning (PT) improves performance by dividing the search space, reducing BVH rebuild time, but introduces partitioning overhead that limits its standalone efficiency. Adding Multi-BVH coprocessing (PT+MBC) enhances performance by reducing kernel launch overhead and stream-switching costs. The final optimization, PT+MBC+LU, achieves the lowest processing times by minimizing redundant updates and reducing memory transfer time.

5 Related Work

Stream KNN and kNN Join Several works have focused on processing kNN join queries on streaming data, proposing various innovative solutions to address the unique challenges of this task. Yang et al. [41] introduced a high-dimensional R-tree (HDR-tree) to tackle the reverse kNN problem on data streams, specifically for use in recommender systems. Koudas et al. [16] developed an approximate kNN method for sliding windows, employing adaptive indexing based on space-filling curves to improve efficiency. Expanding on the concept of continuous queries, Mouratidis et al. [23] proposed a solution for continuous nearest neighbor queries on data streams using conceptual partitioning, as well as a distributed kNN processing system designed for publish/subscribe environments via a content-addressable network overlay. To address scalability and dynamic data challenges, Shahvarani et al. [30] introduced ADS-kNN, an adaptive and scalable stream kNN join system tailored for highly dynamic data streams. Hu et al. [14] proposed LSHI, a locality-sensitive hashing-based index, to handle kNN join queries in recommender systems efficiently. Gowanlock [11] presented HybridKNN-Join, which incorporates three key optimizations to fully utilize both CPU and GPU resources, demonstrating significant improvements in kNN join task performance. Zhu [47] presented GTS, a GPU-based tree index designed for the parallel processing of similarity search with great performance gains. These complementary approaches collectively highlight the diversity of techniques developed to advance kNN join query processing on streaming data.

Ray Tracing Ray tracing technology is widely utilized in various graphics applications, including movie special effects [4], advanced computer-aided manufacturing [6], and video games [10]. Recent studies have increasingly explored the use of RT cores to accelerate rendering workloads, such as graph drawing [44], rendering transparent objects [37], dexel modeling [15], and particle movement [1, 2]. Beyond graphics, several works have investigated leveraging RT cores for data processing tasks. These include point location [22, 36], nearest neighbor search [27, 45], DBSCAN clustering [26], database indexing [13], and range minimum queries [20]. All these methods share a similar approach of transforming questions into RT core operations, but they focus on different domains.

6 Conclution

This paper presents StreamingRT, a high-performance framework for stream kNN joins leveraging GPU ray tracing (RT) cores in 3D space. By formulating stream kNN joins as ray tracing problems and proposing boundary extended point partitioning to reduce BVH reconstruction overhead, and BVH lazy updating to minimize redundant computations, with CPU-GPU pipeline and multi-BVH coprocessing improving performance, StreamingRT delivers over 2.2× and 5.8× speedup over the state-of-the-art approach on RT cores and CUDA cores, respectively, paving the way for further innovations in real-time analytics and high-dimensional query handling.

7 Disclose Use of GenAI Tools

This paper was manually written by the authors, with generative AI tools used for spell checking and grammar correction.

References

[1] Pascal R Bähr, Bruno Lang, Peer Ueberholz, Marton Ady, and Roberto Kersevan. 2022. Development of a hardware-accelerated simulation kernel for ultra-high vacuum with Nvidia RTX GPUs. *The International Journal of High Performance Computing Applications* 36, 2 (2022), 141–152.

[2] Simon Blyth. 2020. Meeting the challenge of JUNO simulation with Opticks: GPU optical photon acceleration via NVIDIA® OptiXTM. In *EPJ Web of Conferences*, Vol. 245. EDP Sciences, 11003.

[3] Zhiyong Cheng and Jialie Shen. 2014. Just-for-me: an adaptive personalization system for location-aware social music recommendation. In *Proceedings of international conference on multimedia retrieval*. 185–192.

[4] Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*. 1–6. doi:10.1109/RT.2006.280208

[5] Brian Curless and Marc Levoy. 1996. A Volumetric Method for Building Complex Models from Range Images. *Seminal Graphics Papers: Pushing the Boundaries, Volume 2* (1996). https://api.semanticscholar.org/CorpusID:260380609

[6] Andreas Dietrich, Abe Stephens, and Ingo Wald. 2007. Exploring a Boeing 777: Ray Tracing Large-Scale CAD Data. *IEEE Computer Graphics and Applications* 27, 6 (2007), 36–46. doi:10.1109/MCG.2007.147

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

[8] Facebook. [n. d.]. pytorch3d.ops.knn_points. https://pytorch3d.readthedocs.io/en/latest/modules/ops.html#pytorch3d.ops.knn_points.

[9] NVIDIA Developer Forums. 2023. Access multiple BVH parallel. https://forums.developer.nvidia.com/t/access-multiple-bvh-parallel/260142 Accessed: 2025.2.

[10] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. 2006. Exploring the use of ray tracing for future games. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames* (Boston, Massachusetts) *(Sandbox '06)*. Association for Computing Machinery, New York, NY, USA, 41–50. doi:10.1145/1183316.1183323

[11] Michael Gowanlock. 2021. Hybrid KNN-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features. *J. Parallel and Distrib. Comput.* 149 (2021), 119–137. doi:10.1016/j.jpdc.2020.11.004

[12] Dan He, Sibo Wang, Xiaofang Zhou, and Reynold Cheng. 2019. GLAD: A Grid and Labeling Framework with Scheduling for Conflict-Aware k k NN Queries. *IEEE Transactions on Knowledge and Data Engineering* 33, 4 (2019), 1554–1566.

[13] Justus Henneberg and Felix Schuhknecht. 2023. RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing. *arXiv preprint arXiv:2303.01139* (2023).

[14] Yupeng Hu, Chong Yang, Peng Zhan, Jia Zhao, Yujun Li, and Xueqing Li. 2021. Efficient continuous KNN join processing for real-time recommendation. *Personal and Ubiquitous Computing* 25 (2021), 1001–1011.

[15] Masatomo Inui, Kohei Kaba, and Nobuyuki Umezu. 2021. Fast dexelization of polyhedral models using ray-tracing cores of GPU. *Comput. Aided Des. & Appl* 18, 4 (2021), 786–798.

[16] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang. 2004. Approximate NN queries on streams with guaranteed error/performance bounds. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, 804–815.

[17] Stanford University Computer Graphics Laboratory. 2014. The Stanford 3D Scanning Repository. http://graphics.stanford.edu/data/3Dscanrep/.

[18] Ruiyuan Li, Rubin Wang, Junwen Liu, Zisheng Yu, Huajun He, Tianfu He, Sijie Ruan, Jie Bao, Chao Chen, Fuqiang Gu, et al. 2021. Distributed spatio-temporal k nearest neighbors join. In *Proceedings of the 29th international conference on advances in geographic information systems*. 435–445.

[19] Danzhou Liu, Ee-Peng Lim, and Wee-Keong Ng. 2002. Efficient k nearest neighbor queries on remote spatial databases using range estimation. In *Proceedings 14th International Conference on Scientific and Statistical Database Management*. IEEE, 121–130.

[20] Yangming Lv, Kai Zhang, Ziming Wang, Xiaodong Zhang, Rubao Lee, Zhenying He, Yinan Jing, and X Sean Wang. 2024. RTScan: Efficient Scan with Ray Tracing Cores. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1460–1472.

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

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

[23] Kyriakos Mouratidis and Dimitris Papadias. 2007. Continuous Nearest Neighbor Queries over Sliding Windows. *IEEE Trans. on Knowl. and Data Eng.* 19, 6 (June 2007), 789–803.

[24] MPA. [n. d.]. The Millennium Simulation Project. https://wwwmpa.mpa-garching.mpg.de/galform/virgo/millennium/. Accessed 2025.2.

[25] MPA. 2005. Millennium Run Semi-Analytic Galaxy Catalogue. https://wwwmpa.mpa-garching.mpg.de/galform/virgo/millennium/. Accessed: 2025.2.

[26] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 963–973.

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

[28] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services. In *2011 IEEE 12th International Conference on Mobile Data Management*, Vol. 1. IEEE, 7–16.

[29] NVIDIA. [n. d.]. NVIDIA Turing Architecture Whitepaper. https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. Accessed: 2025.2.

[30] Amirhesam Shahvarani and Hans-Arno Jacobsen. 2021. Distributed Stream KNN Join. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1597–1609. doi:10.1145/3448016.3457269

[31] Bilong Shen, Ying Zhao, Guoliang Li, Weimin Zheng, Yue Qin, Bo Yuan, and Yongming Rao. 2017. V-tree: Efficient knn search on moving objects with road-network constraints. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 609–620.

[32] Volker Springel et al. 2005. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *Nature* 435 (2005), 629–636. doi:10.1038/nature03597 arXiv:astro-ph/0504097

[33] David Taniar and Wenny Rahayu. 2013. A taxonomy for nearest neighbour queries in spatial databases. *J. Comput. System Sci.* 79, 7 (2013), 1017–1039.

[34] The New York City Taxi and Limousine Commission. 2021. TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[35] Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel. 2018. A detailed study of ray tracing performance: render time and energy cost. *The Visual Computer* 34 (2018), 875–885.

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

[37] Xin Wang and Risong Zhang. 2021. Rendering transparent objects with caustics using real-time ray tracing. *Computers Graphics* 96 (2021), 36–47. doi:10.1016/j.cag.2021.03.003

[38] Ziming Wang, Kai Zhang, Yangming Lv, Yinglong Wang, Zhigang Zhao, Zhenying He, Yinan Jing, and X Sean Wang. 2024. RTOD: Efficient Outlier Detection With Ray Tracing Cores. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[39] Zhengdao Xu and Arno Jacobsen. 2007. Adaptive location constraint processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 581–592.

[40] Lixin Xue. n.d.. Fixed Radius NN Search. https://github.com/lxxue/FRNN.

[41] Chong Yang, Xiaohui Yu, and Yang Liu. 2014. Continuous KNN join processing for real-time recommendation. In *2014 IEEE International Conference on Data Mining*. IEEE, 640–649.

[42] Jihun Yi and Sungroh Yoon. 2024. CKNN: Cleansed k-Nearest Neighbor for Unsupervised Video Anomaly Detection. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*. 3002–3011.

[43] Hongzhi Yin, Yizhou Sun, Bin Cui, Zhiting Hu, and Ling Chen. 2013. Lcars: a location-content-aware recommender system. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 221–229.

[44] Stefan Zellmann, Martin Weier, and Ingo Wald. 2020. Accelerating Force-Directed Graph Drawing with RT Cores. In *2020 IEEE Visualization Conference (VIS)*. 96–100. doi:10.1109/VIS47514.2020.00026

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

[46] Yuhao Zhu. n.d.. RTNN: Real-Time Nearest Neighbor Search. https://github.com/horizon-research/rtnn.

[47] Yifan Zhu, Ruiyao Ma, Baihua Zheng, Xiangyu Ke, Lu Chen, and Yunjun Gao. 2024. GTS: GPU-based Tree Index for Fast Similarity Search. *Proc. ACM Manag. Data* 2, 3, Article 142 (May 2024), 27 pages. doi:10.1145/3654945