RTOD: Efficient Outlier Detection With Ray Tracing Cores

Ziming Wang, Kai Zhang, Yangming Lv, Yinglong Wang, Zhigang Zhao, Zhenying He, Yinan Jing, *Member, IEEE*, and X. Sean Wang, *Senior Member, IEEE*

***Abstract*—Outlier detection in data streams is a critical component in numerous applications, such as network intrusion detection, financial fraud detection, and public health. To detect abnormal behaviors in real-time, these applications generally have stringent requirements for the performance of outlier detection. This paper proposes RTOD, a high-performance outlier detection approach that utilizes RT cores in modern GPUs for acceleration. RTOD transforms distance-based outlier detection in data streams into an efficient ray tracing job. By creating spheres centered at points within a window and casting rays from each point, RTOD identifies the outlier points according to the number of intersections between rays and spheres. Besides, we propose two optimization techniques, namely Grid Filtering and Ray-BVH Inversion, to further accelerate the detection efficiency of RT cores. Experimental results show that RTOD achieves up to 9.9× speedups over existing start-of-the-art outlier detection algorithms.**

***Index Terms*—Data streams, outlier detection, ray tracing.**

I. Introduction

An OUTLIER is a data object that deviates significantly from the rest of the objects [1], [2]. For instance, a distance-based outlier is defined as any data point with fewer than K neighbors within a distance of R [3], where K is the neighbor threshold and R is the distance threshold. Detecting outliers from a data stream in real-time is gaining increasing attention as many applications need to detect anomalies, such as fraudulent activities detection in network intrusion detection [4], [5], financial transactions [6], monitoring for abnormal beats in a patient's ECG [7], or unusual pattern detection in marketing [8], [9]. All of these applications would benefit from the real-time identification of those critical events.

In real-time applications, as new data points arrive continuously, outlier detection is typically performed on a sliding window, i.e., the set of most recent data objects, to ensure computation efficiency and outlier detection in a local context. As the window slides, the outlier status of each point can change. Expired points may cause nearby points to become outliers, while new points may turn nearby points into inliers. Consequently, it is necessary to calculate the neighbors of points in new windows to find the outliers, which requires a large amount of computation and poses a challenge for real-time outlier detection. Numerous studies [8], [10], [11], [12], [13], [14], [15] have focused on Distance-based Outlier Detection in Data Streams (DODDS).

Modern desktop and server class GPUs incorporate ray tracing (RT) cores to expedite the real-time rendering of three-dimensional scenes. NVIDIA supports the utilization of RT cores by providing options to create customized objects and launch rays in the space. With a set of optional shaders to control rendering, RT cores can be utilized to implement various tasks beyond rendering. A range of studies utilizes RT cores to accelerate data processing, including point location [16], [17], nearest neighbor search [18], [19], [20], DBSCAN [21], database indexing [22], and range minimum queries [23]. In these applications, primitives such as triangles or spheres are created and arranged in a three-dimensional space to represent data. These primitives are organized into a Bounding Volume Hierarchy (BVH) tree. Queries are turned into rays to traverse the BVH tree by RT cores, where intersections with primitives indicate that the corresponding data may meet certain conditions. For a task being efficiently mapped as a ray tracing problem, its performance can be notably improved because rays only intersect the eligible objects, and the intersection tests are significantly accelerated by RT cores.

In this paper, we transform DODDS into an efficient ray tracing job accelerated by RT cores. We create a sphere for each point where the radius of each sphere equals the distance threshold. Then, we cast a short ray from each point, and each ray will make intersections with its nearby spheres. In RTOD, a data point is taken as an outlier when the number of intersections for its ray is less than the neighbor threshold. It is worth noting that, the only nearby spheres will be intersected by rays, avoiding a large number of intersections and data accesses. Besides, we propose the following two techniques to further optimize the DODDS performance with RT cores. (1) *Grid Filtering:* We divide the data space into a grid, where all units have the same size. The unit where a point belongs can be quickly calculated by the point's value and the unit size. The number of rays can be reduced by skipping dense units (containing more than K points) and casting rays only from points in sparse units (containing no more than K points), alleviating much intersection test overhead. (2) *Ray-BVH Inversing:* Instead of building the BVH tree from all

Received 4 December 2023; revised 29 July 2024; accepted 22 August 2024. Date of publication 3 September 2024; date of current version 13 November 2024. Recommended for acceptance by S. Whang. *(Corresponding author: Kai Zhang.)*

The authors are with the School of Computer Science, Fudan University, Shanghai 200438, China (e-mail: wangzm22@m.fudan.edu.cn; zhangk@fudan.edu.cn; ymlv21@m.fudan.edu.cn; wangyinglong@qlu.edu.cn; zgzhao21@m.fudan.edu.cn; zhenying@fudan.edu.cn; jingyn@fudan.edu.cn; xywangCS@fudan.edu.cn).

Digital Object Identifier 10.1109/TKDE.2024.3453901

points in the current window and casting rays from the points in the sparse units, we build the BVH tree based on the sparse units and cast rays from all points. As a result, fewer points are used to build the BVH tree, suppressing BVH construction overhead, and most rays have fewer intersections, leading to a reduced load for each ray.

The contributions of this paper are as follows:

- We propose a systematic approach to transform DODDS into an efficient ray tracing job in a three-dimensional space.
- We propose two techniques to accelerate the detection: Grid Filtering to alleviate the intersection overhead and Ray-BVH Inversing to reduce the load of each ray.
- We provide an open-source implementation of our algorithm,[1] RTOD, and intensively evaluate its performance under diverse workloads. Our experimental results show that RTOD can achieve up to 9.9× higher performance than the state-of-the-art approaches.

The rest of the paper is organized as follows. Section II defines the problem formally and gives an introduction to ray tracing. Section III describes the detailed techniques in the design and implementation. Section IV provides a comprehensive evaluation of RTOD. Section V discusses the related work, and Section VI concludes the paper.

II. Background

In this section, we present the formal definition of DODDS and an introduction to ray tracing.

A. Problem Definition of DODDS

A data object is labeled an outlier if it deviates from the normal behavior, often indicative of noise or an anomaly. There are various approaches for defining outliers, such as distance-based [3], density-based [24], tree-based [25], or projection-based [26]. Among them, a *distance-based outlier* within a dataset is a data point having fewer than K other data points within a certain distance R, while these other data points within R are referred to as *neighbors*. The data point having K or more neighbors is considered a distance-based inlier. The euclidean distance metric is employed to calculate the distances between points.

A *data stream* is an infinite sequence of data points, $\cdots, o_{i-1}, o_i, o_{i+1}, \cdots$, which are sorted in increasing order of time. Data streams are typically handled by employing a sliding window that encompasses the most recent points, where a *window* denotes the set of the latest points and a *slide* refers to a set of points removed from or added to a window as it slides. The removed portion is termed an expired slide, while the added portion is termed a new slide. In this paper, we formally define the window size and slide size as the number of points, as in a majority of the related studies [3], [10], [11], [12], [13], [14], [15].

The *Distance-based Outlier Detection in Data Streams (DODDS)* is defined as follows: Given the window size W, the slide size S, the distance threshold R, and the neighbor count threshold K, DODDS detects the distance-based outliers in each sliding window $\cdots$, H_i ($o_{i-W+1}, \cdots, o_i$), H_{i+S} ($o_{i+S-W+1}, \cdots, o_{i+S}$), $\cdots$. In DODDS, expired and newly arrived neighbors can affect the status of existing points. Fig. 1 illustrates an example scenario of DODDS with the following parameter values: $W = 5$, $S = 2$, and $K = 4$. In window H_5, a data point o_3 has four neighbors, o_1, o_2, o_4 and o_5, within R and hence is regarded as an inlier. However, in window H_7, o_3 becomes an outlier as it loses two old neighbors, o_1 and o_2 , without acquiring any new neighbors. Therefore, whenever the window slides, it is essential to recalculate the number of neighbors for points in the new window to identify outliers, demanding a substantial computational workload.

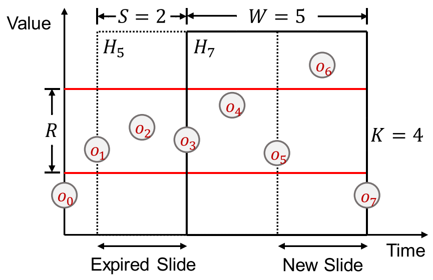


Fig. 1. An example of DODDS.

B. Background of Ray Tracing

Ray tracing is a computationally intensive rendering technology that realistically simulates the lighting of a scene and its objects, including effects like bouncing off surfaces, traversing through transparent materials, undergoing refraction, and occasionally being absorbed [27]. The latest commodity off-the-shelf GPUs incorporate RT cores, which are specialized hardware to enhance the ray tracing performance. The first-generation RT cores in the Turing architecture of NVIDIA achieve up to 10× speedup over software implementations [27]. RT cores are featured in Nvidia GPUs with Turing, Ampere, and Ada architectures. In addition to full support for hardware-based ray tracing on Nvidia's and AMD's latest GPUs, such as Nvidia GeForce RTX 4090 and AMD Radeon RX 7800 XT, chips like the Apple M3 and A17 Pro also integrate RT cores for efficient hardware-accelerated ray tracing.

The core of ray tracing involves calculating intersections between a ray and a three-dimensional scene, typically depicted as a collection of geometric primitives like triangles and spheres. This intersection test primarily dictates rendering time [28] and is expedited through RT cores. To perform the intersection test, the scene's primitives are divided into partitions. In particular, primitives are wrapped by bounding volumes, commonly in the form of Axis-Aligned Bounding Boxes (AABBs). The AABBs that directly wrap primitives are called *leaf AABBs*. AABBs are organized hierarchically as a tree structure known as the Bounding Volume Hierarchy (BVH) tree. Fig. 2 shows a scene containing three triangles. Leaf AABBs store the scene primitives, and larger AABBs enclose other smaller AABBs.

[1]The code of RTOD is available at https://github.com/Wang-Zm/rtod.

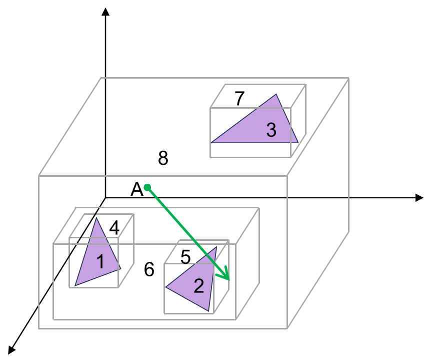


Fig. 2. An example of ray tracing

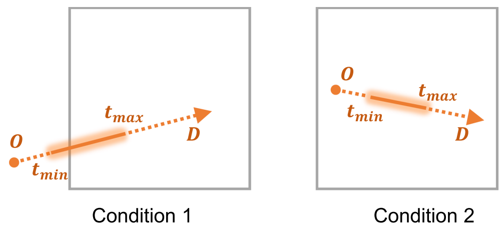


Fig. 3. Conditions for ray-AABB intersection

A ray is parameterized by an origin coordinate O and a direction vector D [29]:

$$R(t) = O + Dt. \tag{1}$$

The ray is usually specified as an interval on a line by setting $t \in [t_{min}, t_{max}]$. Once cast, the rays traverse the BVH tree, and their intersection tests with the primitives are accelerated by RT cores. If a ray intersects an AABB, the AABBs or primitives enclosed by that AABB are tested for intersections with the ray. Otherwise, all enclosed by that AABB are skipped because they definitely do not intersect the ray. As shown in Fig. 2, Ray A does not intersect AABB 7, so primitive 3 is skipped. Conversely, Ray A intersects AABB 6, prompting further testing for intersections with AABB 4 and AABB 5. It is crucial to understand when a ray intersects an AABB, forming the basis of our algorithm. A ray intersects an AABB under two conditions [18], as shown in Fig. 3. First, the ray intersects the AABB's bounds (its six faces). Second, the ray's origin lies within the AABB, even if the ray does not intersect the bounds.

In the Nvidia OptiX programming model, CUDA kernels are launched by specifying the number of threads, and each thread can cast rays. Then, CUDA cores and RT cores are utilized for intersection tests between rays and the scene during the ray traversal. OptiX presents shaders to perform user-defined behaviors to control different stages in the ray traversal. During traversal, the Intersection shader is called whenever leaf AABBs are intersected. This shader is responsible for conducting the ray-primitive intersection test. If a ray intersects a primitive after the intersection test, the user-defined Any-Hit shader is called optionally to process the intersection information. Closest-Hit shader and Miss shader are also defined by users and are called when a ray intersects the closest primitive or does not intersect any primitive, respectively. With these shaders, RT cores can be used to accelerate diverse tasks.

Recent papers have begun employing RT cores to expedite data processing tasks. RTNN [18] and RT-kNNS [19] formulate neighbor search as a ray tracing problem. RTXRMQ [23] computes range minimum queries with RT cores by mapping elements into triangles that are positioned and shaped based on the element's value and position in the array, respectively, such that the closest hit of a ray cast from a point specified by the query corresponds to the result of that query. RTIndeX [22] expresses database indexing in a ray tracing pipeline to utilize the built-in hardware acceleration of RT cores. In RTNN, the second intersection condition is relied on to implement neighbor search. Data points are represented by spheres in space, and the radius of the spheres is determined by the distance threshold of the neighbor search problem. Then, the spheres are used to build a BVH tree. Query points are mapped into very short rays that traverse the BVH tree, where an intersection with a sphere means the data point corresponding to the sphere is the query point's neighbor.

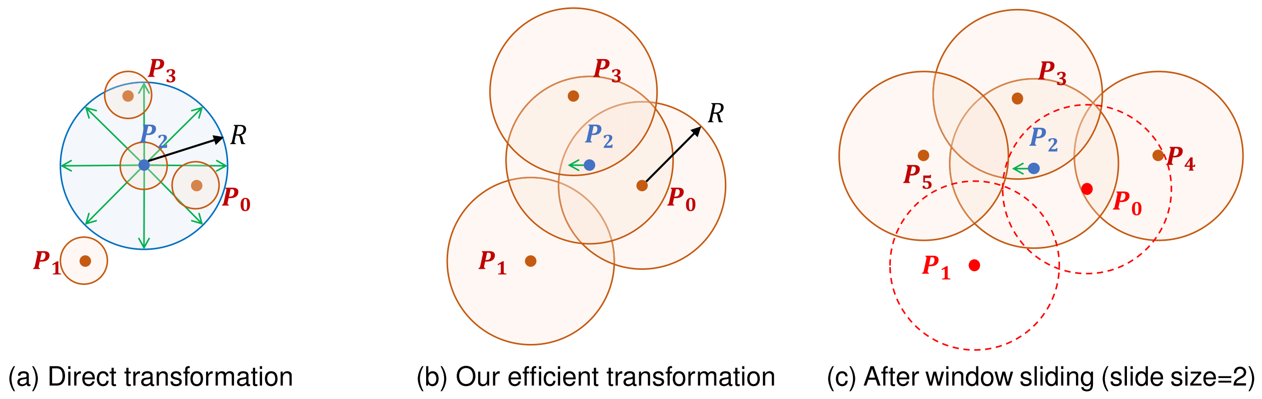
III. RTOD: Design and Implementation

We describe how to formulate DODDS as ray tracing (Section III-A). We then present two performance enhancement techniques (Section III-B and III-C), followed by an in-depth description of our algorithm (Section III-D).

A. DODDS as Ray Tracing: Basic Idea

DODDS involves identifying points with fewer than K neighbors within a distance of R. The DODDS problem can be transformed as a ray tracing problem by creating a sphere for each point in the current window, with the point serving as the sphere's center, and casting R-length rays around from each point (shown in Fig. 4(a)). For a three-dimensional point (a, b, c), the sphere's center and the ray's origin have the same coordinates (a, b, c). For a one-dimensional point a and a two-dimensional point (a, b), their coordinates are both $(a, 0, 0)$ and $(a, b, 0)$, respectively. The intersection between a ray and a sphere means the corresponding points may be neighbors, and then the distance between them is calculated to do an exact check. If the distance is less than R, the two points are neighbors, such as P_2 with P_0 and P_3 in Fig. 4(a). There is no need to calculate the distance between the ray's origin and other points whose spheres are not intersected because those points are too far to be neighbors. However, multiple rays have to be cast from a point to guarantee all neighbors are obtained, resulting in massive intersection tests. For example, $\frac{2\pi}{\arcsin(\frac{r}{R})}$ rays are cast from each point when a r-radius sphere is created around each point.

In order to reduce the number of intersection tests and alleviate the load of RT cores, we propose RTOD, which creates a R-radius sphere for each point, where the point serves as the sphere's center, and casts a single short ray (e.g., $t \in$ [0, `FLOAT_MIN`]) in any direction (e.g., [1, 0, 0]) from each point (shown in Fig. 4(b)). Any point enclosed by a sphere is the neighbor to the sphere's center. This design significantly reduces the rays to be cast. Since a point is enclosed by the sphere



P_3
P_2
R
P_0
P_1
P_4
P_5
(a) Direct transformation
(b) Our efficient transformation
(c) After window sliding (slide size=2)

Fig. 4. Transforming DODDS as a ray tracing job. Take the search of P_2's neighbors as an example. (a) Casting multiple R-length rays from P_2 to search its neighbors; (b) Casting a short ray from P_2 and testing whether P_2 is enclosed by R-radius spheres centered on other points; (c) Testing whether P_2 is enclosed by R-radius spheres centered on all other points in the new window after sliding (P_0 and P_1 expire, while P_4 and P_5 arrive).

centered at this point, the point enclosed by at least $K+1$ spheres is an inlier; otherwise, it is an outlier. The process is repeated as the window slides (shown in Fig. 4(c)).

The procedure of transformation is as follows. An R-radius sphere is created around each point, and an AABB (leaf AABB) that circumscribes each sphere, essentially the smallest AABB that encompasses the sphere, is generated. Then, a BVH tree is built from the AABBs, and FLOAT_MIN-length rays are cast from all points to traverse the BVH tree. We rely on the second intersection condition in Section II-B to implement the transformation. When a ray originating from a point Q intersects a leaf AABB, the Intersection shader in OptiX is called. It is tested whether Q resides in (a.k.a., *intersects* in this paper) the sphere encompassed by the AABB by comparing the distance between Q and the sphere's center P with R. If the distance is less than R, the ray cast from Q intersects the sphere, and if Q and P are not the same point, P is recorded as a neighbor of Q. If the number of intersected spheres exceeds K for a ray, the traversal is terminated, and the point from which the ray is cast is classified as an inlier. Conversely, the corresponding point is an outlier if a ray intersects no more than K spheres when the traversal ends.

Whenever the window slides, the BVH tree is built and rays are cast from all points in the new window. Furthermore, a large portion of rays (considered inliers) can intersect $K+1$ spheres. As a result, there is a huge BVH construction overhead and a large number of intersection tests. To address these inefficiencies, we propose Grid Filtering to reduce the costs for intersection tests and Ray-BVH Inversing to alleviate BVH construction overhead and load of each ray.

B. Grid Filtering

Casting rays from all points results in a large number of intersection tests. For example, for STK [3] dataset under default parameters described in Section IV-A, 100,000 rays are cast in every window, leading to 5.6×10^6 intersection tests on average. This subsection introduces a technique that divides the data space into a grid to filter most inliers quickly and only cast rays

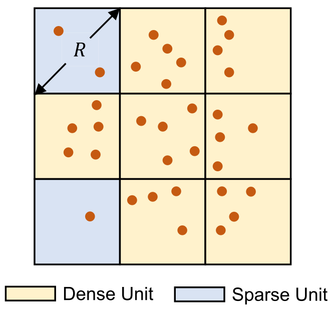


Fig. 5. A grid with $K=3$.

from the points in some grid units to alleviate the load of RT cores for intersection tests.

For the DODDS problem, outliers are the unusual points in a dataset, accounting for a small proportion. Inliers are spatially close by definition. If many points gather together, they tend to be inliers. RTOD divides the data space into multiple grid *units*, and the diagonal length of each unit is R. The length of each side of the unit is the same. For example, the side length of the unit in a two-dimensional space is $\frac{\sqrt{2}}{2}R$ and that in a three-dimensional space is $\frac{\sqrt{3}}{3}R$. The points in the current window are grouped into units. Therefore, the distance between any two points in a unit is no longer than R, meaning that any two points in the unit are neighbors to each other. In a unit containing N points, each point has at least $N-1$ neighbors. If N exceeds K, all points in the unit have no fewer than K neighbors, classifying the points as inliers. Such a unit is called a *dense unit*; otherwise, it is called a *sparse unit*. Fig. 5 shows a grid with $K=3$. The blue units are sparse units because no more than four points exist in each. With Grid Filtering, rays are cast only from the points in sparse units to check if these points are outliers, leading to fewer rays and fewer intersection tests and alleviating the load of RT cores.

With a grid, the detection process is as follows. When the window slides, the belonging units for the expired and new points are calculated based on their values and the unit side length. For instance, for a point (x, y, z) in three-dimensional

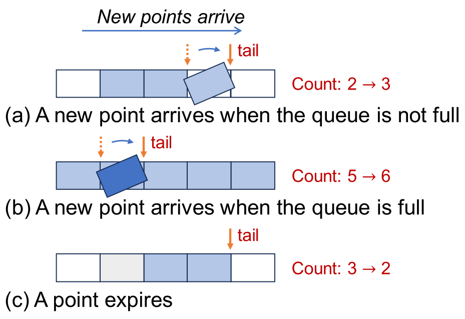

New points arrive
tail
Count: 2 → 3
(a) A new point arrives when the queue is not full
tail
Count: 5 → 6
(b) A new point arrives when the queue is full
tail
Count: 3 → 2
(c) A point expires

Fig. 6. Maintaining the fixed-capacity queue.

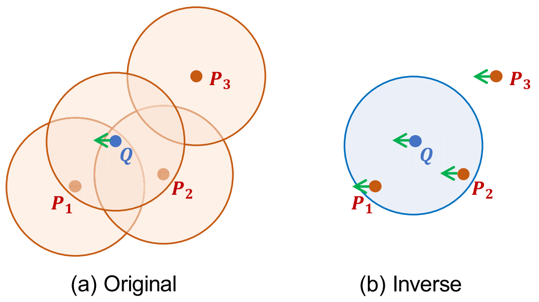


Fig. 7. Inversion. Q is in the sparse unit, and P_1, P_2, P_3 are in the dense units.

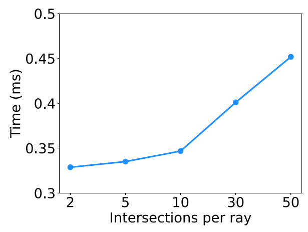


Fig. 8. Performance of ray traversal under different numbers of intersections per ray for STK.

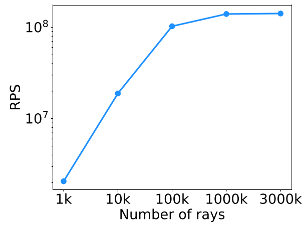


Fig. 9. RPS under different numbers of rays for STK.

space, the index of the unit to which the point belongs can be calculated as $(\frac{x}{\frac{\sqrt{3}}{3}R}, \frac{y}{\frac{\sqrt{3}}{3}R}, \frac{z}{\frac{\sqrt{3}}{3}R})$, where $\frac{\sqrt{3}}{3}R$ is the side length of the unit. Subsequently, the expired points are removed from their corresponding units, and the new points are added to theirs. After that, all units are checked for sparse units or dense units. Rays are then cast from the points within sparse units to find the neighbors of these points, while dense units can be skipped because the points in them have been inliers.

In addition to the grid-based approach in this paper, other data structures, such as micro-clusters in MCOD [13], can also be used to filter most inliers. With micro-clusters in MCOD, the distances between new data points and cluster centers should be calculated to determine the cluster where new points are inserted. Moreover, points should be merged when a new cluster appears, while a range query is done for each point in the cluster when its size shrinks below $K + 1$. As maintaining clusters incurs much higher overhead, grid-based units are more efficient at outlier detection.

As a sparse unit contains up to K points, we can store only the most recent K points within a unit rather than storing all the points that belong to it. If a dense unit transitions to a sparse unit, all points in the sparse unit are retained because K points are stored in the unit before. We implement a fixed-capacity queue to store points for the units. In this design, a queue can hold a maximum of K points, which consumes less memory than storing all points. Fig. 6 illustrates how to maintain the queue. When sliding, new points are assigned to their respective units (shown in Fig. 6(a)), and the most recent points replace the older ones in the queue as it is full (shown in Fig. 6(b)). The unit's point count is incremented whenever new points arrive. In the case of the expired slide, only the count of points within the units to which the expired points belong is updated (shown in Fig. 6(c)). The approach to handling the expired slide is much faster because it does not involve accessing or modifying individual points in the units.

C. Ray-BVH Inversing

While Grid Filtering reduces the number of rays to be cast, each remaining ray still intersects numerous spheres. For example, using the STK dataset under default parameters described in Section IV-A, each remaining ray intersects 35.6 spheres on average after Grid Filtering. Fig. 8 compares the performance of ray traversal under different numbers of intersections with spheres per ray for STK. For each window, 800 rays are cast, which is approximately equal to the number of points after Grid Filtering, as detailed in Section IV-D. The performance with 50 intersections is 37.4% lower than that with 2 intersections. Therefore, the load of each remaining ray is non-trivial. As the number of RT cores is much fewer than that of CUDA cores, e.g., 82 RT cores but 10496 CUDA cores on NVIDIA GeForce RTX 3090, it is critical to utilize RT cores effectively. Since there is no official tool to evaluate the utilization of RT cores, we cast various numbers of rays and measure the number of rays processed per second (RPS) to assess the utilization of RT cores. As depicted in Fig. 9, for the STK dataset under default parameters, an increase in the number of rays leads to a corresponding rise in RPS. 1000K rays and 3000K rays mean casting 10 and 30 rays from each point of the current window, respectively. RPS tends to plateau upon reaching 100K rays. Therefore, casting 800 rays cannot effectively leverage RT cores. Moreover, the BVH tree is always built from all points in the new window whenever the window slides, leading to a long index construction time. We propose Ray-BVH Inversing, a technique that inverses the role of points used to build the BVH

Algorithm 1: DODDS as Ray Tracing.

```
Input: A data stream $S$, a grid $G$, radius $R$, neighbor threshold $K$
Output: outliers $O$ in every sliding window
1: while NewSlideArrives($S$, $S_{new}$) do ▷ New slide $S_{new}$ arrives
2:    $W \leftarrow$ UpdateCurrentWindow($S_{new}$)
3:    $S_{expired} \leftarrow$ GetExpiredSlide($S$)
4:    $U_s \leftarrow$ UpdateUnits($G$, $S_{expired}$, $S_{new}$)
5:    $bvh \leftarrow$ BuildBVH($U_s, R$)
6:    $O \leftarrow$ CastRays($W, bvh, R, K$)
7:    return $O$
8: end while
```

tree and those used to cast rays. Instead of using all points in the window to build the BVH tree and the points in sparse units to cast rays, we build the BVH tree from the points in sparse units and cast rays from all points, which better leverages RT cores while alleviating the BVH construction overhead and load of each ray.

Checking if a point is enclosed by other R-radius spheres has the same effect as if the R-radius sphere centered at this point encloses other spheres' centers. Fig. 7 illustrates the inversion. In Fig. 7(a), R-radius spheres are created around each point and a ray is cast from the point Q, which is in the sparse unit. The point Q is enclosed by the spheres centered at Q, P_1 and P_2. In Fig. 7(b), an R-radius sphere is created around Q and rays are cast from all points. The R-radius sphere centered at Q encloses the points Q, P_1 and P_2. With the inversion, when rays originating from P_1 and P_2 intersect a sphere centered at Q, instead of updating the neighbors of P_1 and P_2, we record P_1 and P_2 as neighbors of Q. During traversal, the spheres with $K+1$ intersections have been inliers, and the subsequent intersection tests can be skipped. In the end, we can obtain the number of rays intersecting each sphere. The spheres intersected by no more than K rays are selected, and the points corresponding to these spheres are outliers. With Ray-BVH Inversing, BVH construction utilizes only the points in sparse units, reducing the number of points used and ensuring high build performance. The resulting BVH tree is small, leading to fewer intersections per ray and thus a lower load of each ray. While more rays are cast, CUDA cores and RT cores can support high parallelism, which has the potential for better detection performance.

D. Algorithm Details

Algorithm 1 outlines the transformation with two techniques. When the window slides, the new slide is acquired first and is used to update the current window (Line 2). As rays are cast on the device, the points in the current window must be transferred to the device. However, there is no need to transfer the entire window; only the new slide needs to be transferred to the device to update the window by replacing the expired slide. Then, the expired slide is obtained (Line 3). The new slide and the expired slide are used to update units and identify the sparse units (Line 4), which is illustrated in Algorithm 2. Subsequently, the sparse units are transferred to the device and used to build the BVH tree (Line 5), followed by casting rays from all points in the current window to traverse the BVH tree (Line 6). Every time a ray intersects a leaf AABB, the Intersection shader is called. The Intersection shader is illustrated in Algorithm 3. In the end, the count of neighbors for each point in the sparse units is transferred back to the host, and the points with less than K neighbors are selected as outliers.

Algorithm 2: UpdateUnits.

```
Input: a grid $G$, $S_{expired}$, $S_{new}$
Output: $U_s$
1: for each point $p \in S_{expired}$ do
2:    $u \leftarrow$ CalcUnit($p$)
3:    $u$.remove($p$)
4: end for
5: for each point $p \in S_{new}$ do
6:    $u \leftarrow$ CalcUnit($p$)
7:    $u$.add($p$)
8: end for
9: $U_s \leftarrow \emptyset$
10: for each unit $u \in G$ do
11:    if $u.pointCount \leq K$ then
12:       $U_s \leftarrow U_s \cup u$
13:    end if
14: end for
15: return $U_s$
```

Algorithm 3: Intersection_Shader.

```
Input: radius $R$, neighbor threshold $K$
1: $curPoint \leftarrow$ GetAABBCenter()
2: if $curPoint.neighborCount \geq K$ then return
3: end if
4: $rayOrigin \leftarrow$ GetCurRayOrigin()
5: if Distance($rayOrigin$, $curPoint$) $< R$ and $rayOrigin \neq curPoint$ then
6:    $curPoint.neighborCount++$
7: end if
```

In function UpdateUnits(), Algorithm 2, RTOD calculates the belonging units for the expired or new points and removes the expired points or adds the new points (Lines 1-8). After that, the sparse units are identified by comparing the number of points in the units with K (Lines 9-14).

In the Intersection shader, Algorithm 3, the current AABB's center is acquired at first (Line 1), corresponding to the enclosed sphere's center. Then, the sphere's center is checked for an inlier (Lines 2-3), and if so, the Intersection shader ends instantly. Otherwise, the ray's origin is acquired (Line 4), and the distance between the sphere's center and the ray's origin is calculated and compared with R. When the distance is less than R and the two points are not the same, a neighbor is added to the sphere's center (Lines 5-7).

RTOD supports outlier detection for one-dimensional, two-dimensional, and three-dimensional data. In fact, Grid

TABLE I
DATASETS AND DEFAULT PARAMETER VALUES

Dataset	Dim	Size	W	S	R	K
GAU [3]	1	1.0M	100,000	5,000	0.028	50
STK [3]	1	1.1M	100,000	5,000	0.45	50
TAO [13]	3	0.6M	10,000	500	1.9	50

Filtering can be used in higher-dimensional data. Unlike three-dimensional data, where a unit is a cube with grid-based partitioning, a unit becomes a hypercube with higher dimensions. For n-dimensional data, the diagonal length of the hypercube is the distance threshold R (the distance between the farthest two points on the hypercube), and the side length is $R/\sqrt{n}$. Therefore, the distance between any two points in the hypercube does not exceed R. When the number of points within the hypercube exceeds the neighbor threshold K, it becomes a dense unit, and the contained points become inliers. NETS [14] has proven the effectiveness of utilizing a similar grid-based partitioning approach for detecting outliers in high-dimensional space. However, the ray tracing core architecture restricts our approach from being adopted in higher-dimensional data because it is only designed for processing three-dimensional data. In real-world scenarios, DODDS with no larger than three dimensions has important applications, such as oscillation detection in the ocean-atmosphere system [10], stock trading traces [11], and network intrusion detection [5]. For oscillation detection, sea surface temperature, relative humidity, and precipitation are collected from moored ocean buoys for outlier detection to understand and predict El Niño and La Niña, which are oscillations within the ocean-atmosphere system of the tropical Pacific that have a significant influence on global weather patterns.

IV. Experiments

After describing our evaluation methodology (Section IV-A), We conduct a comparative analysis of the overall running time and peak memory usage of RTOD against other methods(Section IV-B), followed by verifying the robustness of the performance of the algorithms (Section IV-C). After that, we analyze the contributions of different optimizations (Section IV-D) and provide the breakdown of RTOD running time (Section IV-E).

A. Experiment Setup

Datasets We use the two real-world data sets and a synthetic data set listed in Table I, used in [3], [14], [15]. The number of dimensions of the datasets ranges from 1 to 3. GAU [3] is generated by a Gaussian mixture model with three distributions. STK [3] contains stock trading records. TAO [10], [13] contains oceanographic data from the Tropical Atmosphere Ocean project. The datasets are used to simulate data streams, where data points in the dataset are read continuously, and each time, a slide size of data points is read as the new slide.

Parameters The main control parameters in DODDS are W (window size), S (slide size), R (distance threshold), and K (number of neighbors threshold). As suggested in the survey conducted by Tran et al. [3], the default parameter values for each dataset are configured to maintain an approximate 1% ratio of outliers. Table I provides an overview of the datasets and their respective default parameter values. Unless specified otherwise, all the parameters take on their default values in our experiments. For NETS, we set the number of sub-dimensions parameters equal to the dimensions of the dataset, as suggested in [14].

Competitors We compare and analyze the performance of RTOD with state-of-the-art approaches, including MCOD [13], NETS [14], and MDUAL [30]. MCOD and NETS focus on the DODDS problem. MCOD uses micro-clusters to filter unqualified outlier candidates. A micro-cluster is formed if more than K points exist within a circle with a radius of $R/2$, ensuring that all these points are inliers. Points that do not fall into any micro-clusters are stored in a list called PD. These points can be either outliers or inliers. Inliers in PD are managed by an event queue, which stores the earliest expiration time of their neighbors and is used to re-check the inliers when the window slides. We provide an open-source implementation of MCOD[2] based on [13]. NETS[3] employs a set-based index structure to form cells that are hyper-cubes in a high-dimensional space. A cell is categorized as an inlier cell if it contains at least $K+1$ points, signifying that all points within it are inliers. The cell becomes an outlier cell when the total number of points in neighboring cells within a range of $2R$ from the current cell is not greater than K. Otherwise, the cell is a non-determined cell. The points in the inlier and outlier cells can be skipped when detecting outliers to improve the detection performance. MDUAL[4] is a streaming distance-based outlier detection algorithm allowing multiple dynamic queries (MD-DODDS), which also supports outlier detection in data streams with a single query.

Hardware and Software We conduct all experiments on a machine with an Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz, 32GB DDR4 DRAM, and an NVIDIA GeForce RTX 3090 with 82 RT cores, 10496 CUDA cores, and 24GB VRAM. The operating system is 64-bit Ubuntu Server 20.04 with Linux Kernel 5.15.0-72-generic. We implement our algorithm in C++; the entire program is compiled with CUDA 12.1, OptiX 7.5, and G++ 7.5.0.

B. Overall Performance and Memory Comparison

We compare all the algorithms using all the datasets with the default value parameters. Fig. 10 shows the overall running time of the methods, where RTOD includes the data transfer time. RTOD runs much faster than the other algorithms with all datasets, outperforming MCOD by 161.6×, NETS by 5.1×, and MDUAL by 123.4× on average. Especially with the GAU dataset, where MCOD and MDUAL spend longer than 150ms, and NETS spends longer than 5ms processing one sliding window, RTOD requires only 0.49ms, which is 379.5×, 337.8×, and 9.9× faster than MCOD, MDUAL, and NETS, respectively. The reasons for the higher performance of RTOD are as follows. First, MCOD, NETS, and MDUAL require maintaining the

[2]The code of MCOD is available at https://github.com/Wang-Zm/mcod.

[3][Online]. Available: https://github.com/kaist-dmlab/NETS (last accessed 2023/12/04).

[4][Online]. Available: https://github.com/kaist-dmlab/MDUAL (last accessed 2023/12/04).

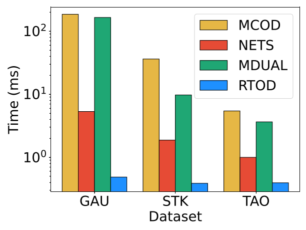


Fig. 10. Overall running time comparison.

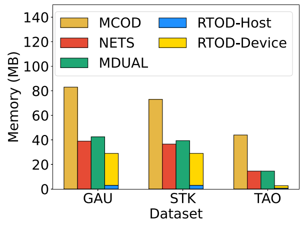


Fig. 11. Overall peak memory comparison.

arrival times and neighbors of points. MCOD also maintains the nearby micro-clusters of points and the nearby points of micro-clusters, and NETS and MDUAL need to find the cells' neighbor cells containing no more than K points. Second, MCOD, NETS, and MDUAL update the index and points' neighbors based on one point at a time, detecting outliers sequentially, whereas RTOD can perform outlier detection in parallel, leveraging the acceleration provided by RT cores. Meanwhile, MDUAL groups the points into a set of non-overlapping and non-empty cells and maintains the cells in each slide of the current window. To obtain all the points in a cell in the window, MDUAL always accumulates the points in cells with the same index as that cell across all slides. Additionally, the two optimization techniques proposed make DODDS more suitable for RT core acceleration.

Fig. 11 reports the peak memory of all the algorithms. RTOD stores all points of the current window, the points used for BVH construction, and the count of neighbors of the points used for BVH construction in both the host and device, while maintaining the units in the host and building the BVH tree in the device. All points in the current window stored in the host are used to update units, which store the index of the points in the current window. Storing the index is more efficient, especially for multi-dimensional points, as only an `Integer` is operated for a point. All points in the window stored in the device are used for casting rays. The points for BVH construction are obtained from sparse units and transferred to the device for building the BVH tree. After traversal, the count of neighbors of the points for BVH construction is transferred from the device to the host for outlier checking. So both host memory and device memory are used in RTOD, accounting for an average of 15.5% and 84.5% across all datasets, respectively. Though considering both types of memory, RTOD requires less memory than MCOD, NETS, and MDUAL, specifically the smallest for TAO and more than 30.3% less when averaged over the other datasets. This is because the arrival times of points, neighbors of points, and micro-cluster information do not need to be stored. RTOD only builds a grid to organize the points, and each unit contains up to K points. At the same time, the BVH tree is built from the points in sparse units, which are much fewer than the window size, even no more than 10% of the window size.

C. *Effects of Parameters on Performance*

We vary the parameter values to verify the robustness of the performance of the algorithms. We present the results for all datasets. When one parameter is varied, the remaining parameters take their default values.

1) Varying Window Size W: (see Fig. 12) The window size roughly indicates the workload imposed on the algorithms. In this experiment, we vary the window size W from 1K to 20K for TAO and from 10K to 200K for GAU and STK. The running time increases with W for all algorithms most of the time due to an increase in the number of points for neighbor search. However, for MCOD on the GAU and STK datasets, the running time decreases as W increases (Fig. 12(a) and (b)). This is because more micro-clusters are formed as W increases, and new coming points tend to be directly inserted into micro-clusters as inliers instead of being checked for outliers by searching for their neighbors. Moreover, the running time of NETS for GAU and STK (Fig. 12(a) and (b)) sometimes decreases as W increases. We believe this is because increasing the number of points in a window directly results in more inlier cells and fewer non-determined cells. This, in turn, enhances the advantages of early detection. The running time of RTOD stays stable as W increases. At the same time, RTOD delivers higher performance in the entire range of W, which is 377.8× faster than MCOD, 5.8× faster than NETS, and 104.9× faster than MDUAL on average over all W values. This is because CUDA cores and RT cores in the GPU support high parallelism, which can efficiently handle larger workloads.

2) Varying Slide Size S: (See Fig. 13) The slide size influences the speed of data streams and specifies the number of points arriving and expiring in each update. In this experiment, we vary the slide size S from 5% to 100% of the default value of S for each dataset. The running time of MCOD, NETS, and RTOD increases with S because with a larger S, more points in a window are affected by expired or new neighbors, and therefore, the time for indexing points increases. While for MDUAL, the running time decreases first and then increases as S increases. The reasons are as follows. MDUAL needs to accumulate the points in cells with the same index across all slides to obtain the points in the cell with that index in the current window. Meanwhile, given a point that is not determined through cell-wise checking, MDUAL finds its neighbors in cells within each slide of the current window, and when a cell spans multiple slides, it will be checked numerous times. A larger S initially decreases the number of slides in the window, which explains the decreasing running time. However, as the slide size (S) further increases, the index update time also increases, resulting in an overall increase in running time. RTOD achieves the least running time in all cases, delivering 835.5×, 8.5×, and 43.6× higher performance than MCOD, NETS, and MDUAL on average over all S values, respectively. This is because RTOD

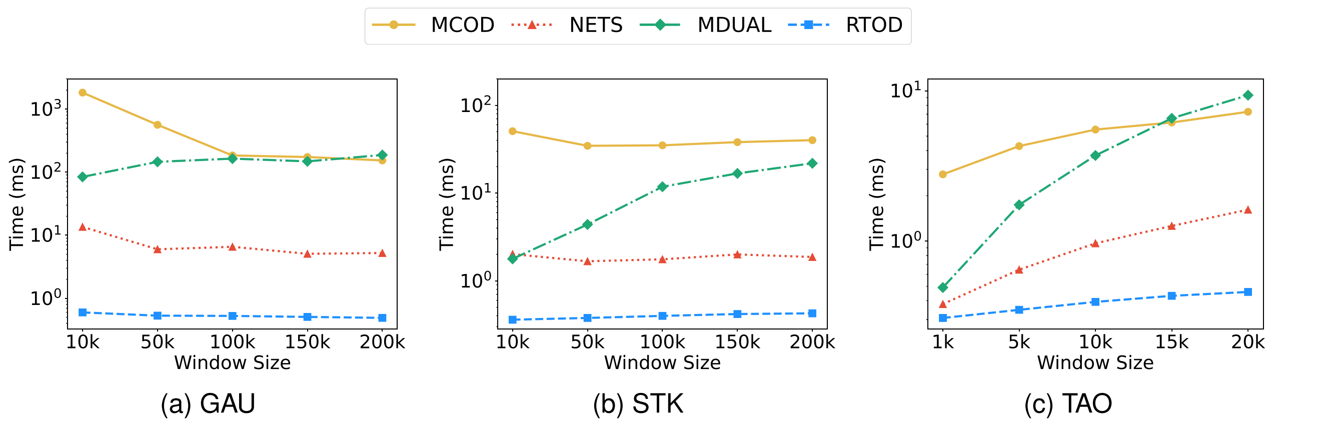


Fig. 12. Varying window size - running time comparison.

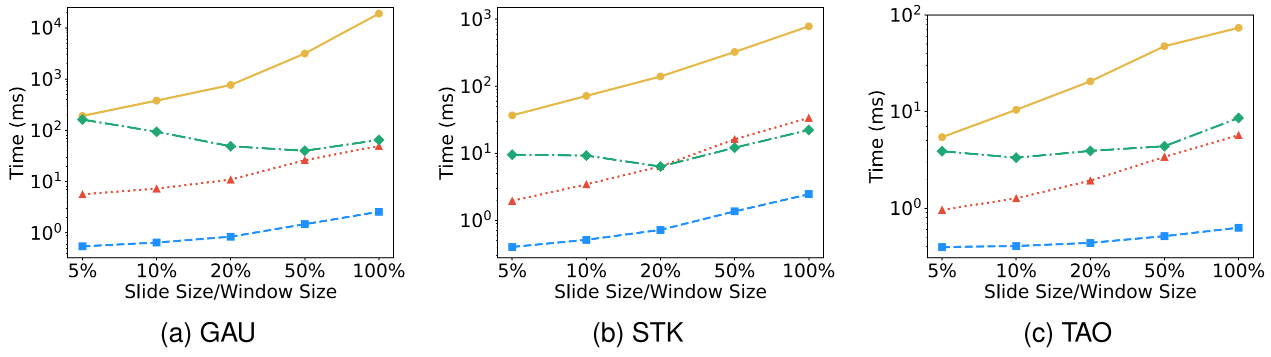


Fig. 13. Varying slide size - running time comparison.

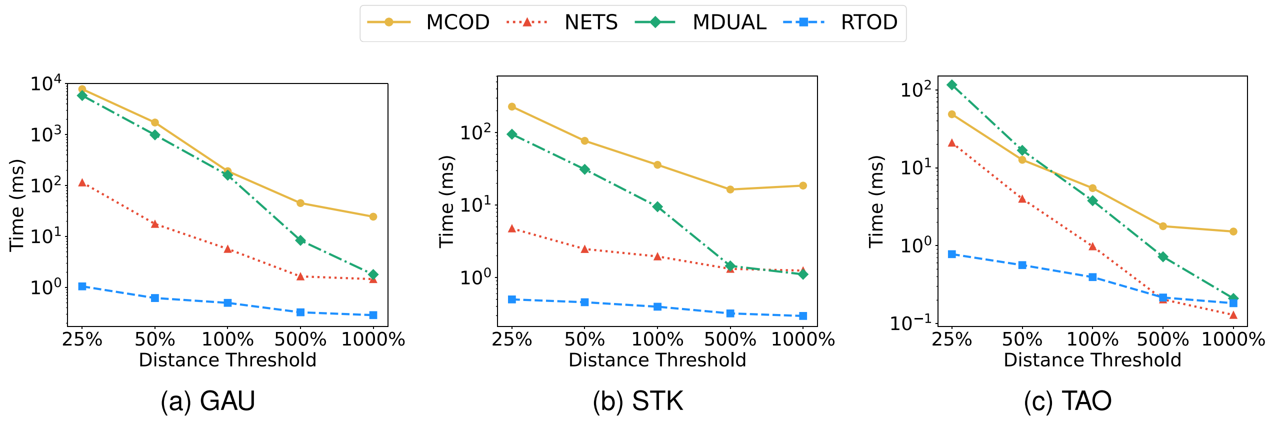


Fig. 14. Varying distance threshold - running time comparison.

only updates units without maintaining neighbor information and recording the points' arrival time.

3) Varying Distance Threshold R: (See Fig. 14) The distance threshold R determines the size of the neighborhood. As R increases, each data point can have more neighbors. In this experiment, we vary R from 25% to 1000% of the default value of R for each dataset. When R increases, almost all the methods run faster because each point can have more neighbors, reducing the likelihood of becoming an outlier. Also, more points can be rapidly identified as inliers due to the increased number of points in micro-clusters (MCOD), cells (NETS, MDUAL) or units (RTOD). RTOD achieves the fastest running time in most cases, with an average of 784.2×, 14.1×, and 532.0× higher performance than MCOD, NETS, and MDUAL, respectively. However, when R equals 1000% of the default value, RTOD is not the fastest. This is because even if very few points selected

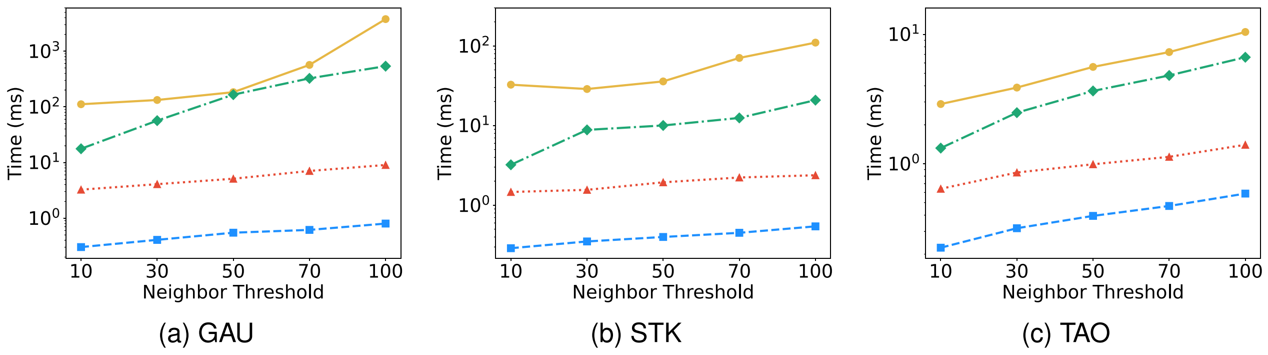


Fig. 15. Varying neighbor threshold - running time comparison.

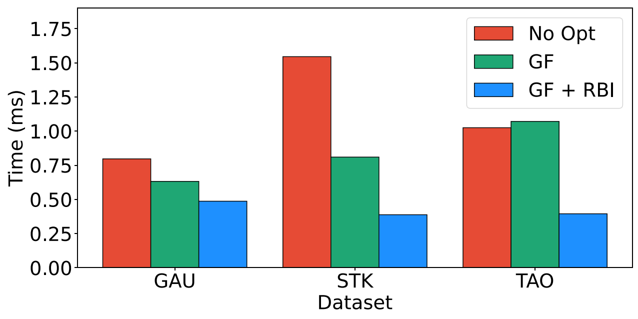


Fig. 16. Effects of our optimizations on RTX 3090.

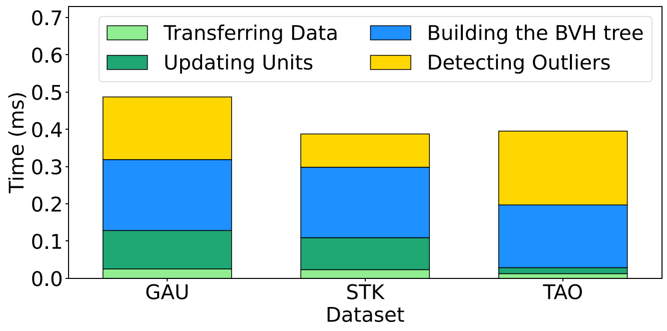


Fig. 17. The breakdown of RTOD running time.

with Grid Filtering are used to build the BVH tree, rays are cast from all points in the current window, leading to inevitable overhead.

4) Varying Neighbor Threshold K: (See Fig. 15) The neighbor threshold K defines the minimum number of neighbors needed for a data point to be considered an inlier. Consequently, the number of outliers increases with K. In general, the running time of the methods increases along with K. This is because fewer points are quickly confirmed as inliers and more points are searched for neighbors. Here again, RTOD is much faster than the other methods. In all cases, RTOD is averagely 492.0×, 5.0×, and 124.0× faster than MCOD, NETS, and MDUAL, respectively.

D. Analyzing the Effects of Optimization Techniques

The optimization techniques we propose in this paper are critical to the performance benefits of RTOD. Using three representative inputs, Fig. 16 compares the performance of three variants of our algorithm on RTX 3090:

- `No Opt`: no optimization;
- `GF`: Grid Filtering only;
- `GF + RBI`: Grid Filtering and Ray-BVH Inversing.

Grid Filtering Compared to `No Opt`, Grid Filtering improves performance by 26.1% and 90.7% for GAU and STK, respectively. This is because the number of rays is largely reduced with Grid Filtering, leading to fewer intersection tests. Specifically, the number of rays decreases from 100K to about 2,800 and 800 on average for GAU and STK, respectively. However, even if Grid Filtering reduces the number of rays from 10K to about 600 on average for TAO, the performance degrades by 4.3%. The reasons are as two-fold. First, after Grid Filtering, the number of spheres intersected by each remaining ray is reduced from about 51 to 42.6 and 35.6 on average for GAU and STK, respectively, while for TAO, it decreases from about 51 to 48.0 on average. There is no significant reduction in the load of each ray for TAO. Second, when the window slides, the units are updated based on the expired points and the new points, taking additional time.

Ray-BVH Inversing Ray-BVH Inversing is used with Grid Filtering and provides an additional 29.8%, 1.1×, and 1.7× performance gain for GAU, STK, and TAO, respectively. Recall that Ray-BVH Inversing reduces the BVH construction overhead and the number of intersection tests per ray. The number of intersection tests per ray decreases on average from 45.6 to 1.3 for GAU, from 37.9 to 0.5 for STK, and from 103.1 to 15.1 for TAO, resulting in an average of 96.0% performance improvement. Additionally, the number of points used to build the BVH tree decreases from the window size to the same number as the rays in the above Grid Filtering technique, leading to a 1.2× performance improvement on average.

E. The Breakdown of RTOD Running Time

Fig. 17 shows the proportion of time spent in key stages of RTOD, including transferring data, updating units, building the

BVH tree, and detecting outliers. When the window slides, the new points used to update the current window in the device, along with the points in the sparse units for BVH construction, are transferred to the device. After traversal, the count of respective neighbors for the points in the sparse units is returned to the host for outlier checking. Due to the relatively small number of transferred data when sliding, e.g., 5,000 new points and 813 points in the sparse units are transferred to the device, and 813 neighbor records to the host for STK [3], the proportion of transfer time is less than 5% for the three datasets. The total time taken by building the BVH tree and detecting outliers exceeds 0.28 ms for each dataset. Building the BVH tree accounts for 39.1%, 48.8%, and 42.6% of the total running time for the respective datasets while detecting outliers accounts for 34.5%, 23.0%, and 50.2%. More than 0.08 ms is spent in updating units for GAU and STK, accounting for more than 21.0%, while less than 0.02 ms is spent for TAO, accounting for less than 4.5%. This is because the slide size for GAU and STK is larger, 10 times over that for TAO, leading to more time to update units and identify sparse units. In addition, the time for detecting outliers is the least for STK compared to GAU and TAO. This is because the number of intersection tests for STK is the fewest, only 0.5, while for GAU and TAO, it is 1.3 and 15.1, respectively.

V. Related Work

Outlier detection The online detection of outliers has been actively explored over the past decades [31]. Recent detection methods belong to different algorithmic families, such as distance-based [3], density-based [24], tree-based [25], projection-based [26], clustering-based [32], [33], statistical-based [34], rotation-based [35], or high-dimensional [36]. Among them, distance-based outlier detection in data streams has gained popularity due to its intuitive definition of outliers and its applicability in unsupervised settings.

Considering the typically high-speed streaming nature of data streams, existing DODDS algorithms [10], [11], [12], [13], [14], [15] commonly adhere to three steps in each sliding window [3]: (1) expired slide processing, (2) new slide processing, and (3) outlier reporting. These algorithms focus on efficiently maintaining the current window and/or efficiently identifying outliers. For example, MCOD [13] and NETS [14] both use specific units to record points and fast filter most inliers first when detecting outliers. All of them are CPU-based methods. Instead, RTOD is a GPU-based method utilizing RT cores for acceleration. There are also other GPU-based methods to detect outliers on data streams. However, not like the currently discussed DODDS problem, their focus is either on static datasets [37] or on other algorithmic families, such as density-based detectors [38], [39].

Ray Tracing Ray tracing technology is extensively employed in graphics applications, including movie special effects [40], advanced computer-aided manufacturing [41], and games [42]. Recent papers have begun to leverage RT cores to accelerate rendering workloads, such as graph drawing [43], transparent objects rendering [44], dexel modeling [45], and particle movement [46], [47], [48]. There are other works that investigate the utilization of RT cores to accelerate data processing jobs, including point location [16], [17], nearest neighbor search [18], [19], [20], DBSCAN [21], database indexing [22], and range minimum queries [23]. Additionally, some studies focus on ray tracing acceleration, involving reducing the number of ray-object intersections and optimizing BVH tree traversal [49], BVH tree autotuning [50], and ray reordering [51].

VI. Conclusion

In this paper, we investigate the potential for accelerating Distance-based Outlier Detection in Data Streams (DODDS) using RT cores. We propose RTOD, a high-performance outlier detection approach that utilizes RT cores in modern GPUs for acceleration. RTOD effectively transforms DODDS as a ray tracing job by using efficient primitives and rays. Meanwhile, we propose Grid Filtering and Ray-BVH Inversing techniques to further accelerate the process by improving the performance of building BVH trees and dramatically reducing the load of each ray. We show that RTOD achieves high performance on various workloads and outperforms the state-of-the-art implementations while consuming lower memory. We believe the proposed approach demonstrates a promising direction for optimizing DODDS performance by leveraging novel hardware.

References

[1] G. Thamaraiselvi and A. Kaliammal, "Data mining: Concepts and techniques," *SRELS J. Inf. Manage.*, vol. 41, no. 4, pp. 339–348, 2004.

[2] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Comput. Surv.*, vol. 41, no. 3, pp. 1–58, 2009.

[3] L. Tran, L. Fan, and C. Shahabi, "Distance-based outlier detection in data streams," *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1089–1100, 2016.

[4] K. Xie et al., "On-line anomaly detection with high accuracy," *IEEE/ACM Trans. Netw.*, vol. 26, no. 3, pp. 1222–1235, Jun. 2018.

[5] S. C. Tan, K. M. Ting, and T. F. Liu, "Fast anomaly detection for streaming data," in *Proc. 22nd Int. Joint Conf. Artif. Intell.*, 2011, pp. 1511–1516.

[6] A. Abdallah, M. A. Maarof, and A. Zainal, "Fraud detection system: A survey," *J. Netw. Comput. Appl.*, vol. 68, pp. 90–113, 2016.

[7] Y. Lin, B. S. Lee, and D. Lustgarten, "Continuous detection of abnormal heartbeats from ECG using online outlier detection," in *Proc. 5th Int. Conf. Inf. Manage. Big Data*, Springer, 2019, pp. 349–366.

[8] L. Cao, Q. Wang, and E. A. Rundensteiner, "Interactive outlier exploration in Big Data streams," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1621–1624, 2014.

[9] D. Georgiadis, M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous outlier detection in data streams: An extensible framework and state-of-the-art algorithms," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1061–1064.

[10] F. Angiulli and F. Fassetti, "Detecting distance-based outliers in streams of data," in *Proc. 16th ACM Conf. Conf. Inf. Knowl. Manage.*, 2007, pp. 811–820.

[11] D. Yang, E. A. Rundensteiner, and M. O. Ward, "Neighbor-based pattern detection for windows over streaming data," in *Proc. 12th Int. Conf. Extending Database Technol.*, 2009, pp. 529–540.

[12] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 76–87.

[13] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos, "Continuous monitoring of distance-based outliers over data streams," in *Proc. IEEE 27th Int. Conf. Data Eng.*, 2011, pp. 135–146.

[14] S. Yoon, J.-G. Lee, and B. S. Lee, "Nets: Extremely fast outlier detection from a data stream via set-based processing," *Proc. VLDB Endowment*, vol. 12, no. 11, pp. 1303–1315, 2019.

[15] L. Tran, M. Y. Mun, and C. Shahabi, "Real-time distance-based outlier detection in data streams," *Proc. VLDB Endowment*, vol. 14, no. 2, pp. 141–153, 2020.

[16] N. Morrical, I. Wald, W. Usher, and V. Pascucci, "Accelerating unstructured mesh point location with RT cores," *IEEE Trans. Vis. Comput. Graph.*, vol. 28, no. 8, pp. 2852–2866, Aug. 2022.
[17] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location," in *Proc. Int. Conf. High Perform. Graph.*, 2019, pp. 7–13.
[18] Y. Zhu, "Rtnn: Accelerating neighbor search using hardware ray tracing," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 76–89.
[19] V. Nagarajan, D. Mandarapu, and M. Kulkarni, "RT-KNNs unbound: Using RT cores to accelerate unrestricted neighbor search," in *Proc. 37th Int. Conf. Supercomput.*, 2023, pp. 289–300.
[20] I. Evangelou, G. Papaioannou, K. Vardis, and A. Vasilakis, "Fast radius search exploiting ray-tracing frameworks," *J. Comput. Graph. Techn.*, vol. 10, no. 1, pp. 25–48, 2021.
[21] V. Nagarajan and M. Kulkarni, "RT-DBSCAN: Accelerating DBSCAN using ray tracing hardware," 2023, *arXiv:2303.09655*.
[22] J. Henneberg and F. Schuhknecht, "RTIndeX: Exploiting hardware-accelerated GPU raytracing for database indexing," 2023, *arXiv:2303.01139*.
[23] E. Meneses, C. A. Navarro, H. Ferrada, and F. A. Quezada, "Accelerating range minimum queries with ray tracing cores," 2023, *arXiv:2306.03282*.
[24] S. Yoon, J.-G. Lee, and B. S. Lee, "Ultrafast local outlier detection from a data stream with stationary region skipping," in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 1181–1191.
[25] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, "Robust random cut forest based anomaly detection on streams," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2016, pp. 2712–2721.
[26] E. Manzoor, H. Lamba, and L. Akoglu, "Xstream: Outlier detection in feature-evolving data streams," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 1963–1972.
[27] TURING RT CORES, "NVIDIA," 2018. [Online]. Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
[28] E. Vasiou, K. Shkurko, I. Mallett, E. Brunvand, and C. Yuksel, "A detailed study of ray tracing performance: Render time and energy cost," *Vis. Comput.*, vol. 34, pp. 875–885, 2018.
[29] P. Shirley, I. Wald, T. Akenine-Möller, and E. Haines, *What is a Ray?*. Berkeley, CA, USA: Apress, 2019, pp. 15–19.
[30] S. Yoon, Y. Shin, J.-G. Lee, and B. S. Lee, "Multiple dynamic outlier-detection from a data stream by exploiting duality of data and queries," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 2063–2075.
[31] A. Ntroumpogiannis, M. Giannoulis, N. Myrtakis, V. Christophides, E. Simon, and I. Tsamardinos, "A meta-level analysis of online anomaly detectors," *VLDB J.*, vol. 32, pp. 845–886, 2023.
[32] H. Liu, J. Li, Y. Wu, and Y. Fu, "Clustering with outlier removal," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 6, pp. 2369–2379, Jun. 2021.
[33] M. B. Toller, B. C. Geiger, and R. Kern, "Cluster purging: Efficient outlier detection based on rate-distortion theory," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 2, pp. 1270–1282, Feb. 2023.
[34] Z. Li, Y. Zhao, X. Hu, N. Botta, C. Ionescu, and G. Chen, "ECOD: Unsupervised outlier detection using empirical cumulative distribution functions," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12181–12193, Dec. 2023.
[35] Y. Almardeny, N. Boujnah, and F. Cleary, "A novel outlier detection method for multivariate data," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 9, pp. 4052–4062, Sep. 2022.
[36] P. Zhu, C. Zhang, X. Li, J. Zhang, and X. Qin, "A high-dimensional outlier detection approach based on local Coulomb force," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 6, pp. 5506–5520, Jun. 2023.
[37] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "Fast outlier detection using a GPU," in *Proc. IEEE Int. Conf. High Perform. Comput. Simul.*, 2013, pp. 143–150.
[38] C. HewaNadungodage, Y. Xia, and J. J. Lee, "GPU-accelerated outlier detection for continuous data streams," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 1133–1142.
[39] K. Yu, W. Shi, N. Santoro, and X. Ma, "Real-time outlier detection over streaming data," in *Proc. IEEE SmartWorld, Ubiquitous Intell. Comput. Adv. Trusted Comput. Scalable Comput. Commun. Cloud Big Data Comput. Internet People Smart City Innov.*, 2019, pp. 125–132.
[40] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali, "Ray tracing for the movie cars'," in *Proc. IEEE Symp. Interactive Ray Tracing*, 2006, pp. 1–6.
[41] A. Dietrich, A. Stephens, and I. Wald, "Exploring a Boeing 777: Ray tracing large-scale CAD data," *IEEE Comput. Graph. Appl.*, vol. 27, no. 6, pp. 36–46, 2007.
[42] H. Friedrich, J. Günther, A. Dietrich, M. Scherbaum, H.-P. Seidel, and P. Slusallek, "Exploring the use of ray tracing for future games," in *Proc. ACM SIGGRAPH Symp. Videogames*, 2006, pp. 41–50.
[43] S. Zellmann, M. Weier, and I. Wald, "Accelerating force-directed graph drawing with RT cores," in *Proc. IEEE Vis. Conf.*, 2020, pp. 96–100.
[44] X. Wang and R. Zhang, "Rendering transparent objects with caustics using real-time ray tracing," *Comput. Graph.*, vol. 96, pp. 36–47, 2021.
[45] M. Inui, K. Kaba, and N. Umezu, "Fast dexelization of polyhedral models using ray-tracing cores of GPU," *Comput. Aided Des. Appl*, vol. 18, no. 4, pp. 786–798, 2021.
[46] P. R. Bähr, B. Lang, P. Ueberholz, M. Ady, and R. Kersevan, "Development of a hardware-accelerated simulation kernel for ultra-high vacuum with nvidia RTX GPUs," *Int. J. High Perform. Comput. Appl.*, vol. 36, no. 2, pp. 141–152, 2022.
[47] S. Blyth, "Meeting the challenge of JUNO simulation with opticks: GPU optical photon acceleration via NVIDIA optixtm," in *Proc. EPJ Web Conf.*, EDP Sciences, 2020, Art. no. 11003.
[48] B. Wang et al., "An GPU-accelerated particle tracking method for Eulerian–Lagrangian simulations using hardware ray tracing cores," *Comput. Phys. Commun.*, vol. 271, 2022, Art. no. 108221.
[49] L. Liu et al., "Intersection prediction for accelerated GPU ray tracing," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2021, pp. 709–723.
[50] K. Herveau, P. Pfaffe, M. Tillmann, W. F. Tichy, and C. Dachsbacher, "Analysis of acceleration structure parameters and hybrid autotuning for ray tracing," *IEEE Trans. Vis. Comput. Graph.*, vol. 29, no. 2, pp. 1345–1356, Feb. 2023.
[51] W. Xiang et al., "Faster ray tracing through hierarchy cut code," 2023, *arXiv:2305.16652*.

Ziming Wang received the BS degree from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2022. He is currently working toward the MS degree with the School of Computer Science, Fudan University, Shanghai, China. His research interests include database systems, data stream processing, and ray tracing.

Kai Zhang received the PhD from the University of Science and Technology of China in 2016. He is an associate professor with the School of Computer Science, Fudan University. He was a research fellow with the National University of Singapore and a visiting scholar with The Ohio State University. His research interests are mainly in the fields of parallel and distributed computing and database systems.

Yangming Lv received the BS degree from the Fudan University, Shanghai, China, in 2021. He is currently working toward the MS degree with the School of Computer Science, Fudan University, Shanghai, China. His research interests include database systems and information security.

Yinglong Wang is a researcher with the Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Qilu University of Technology (Shandong Academy of Sciences). His research interests include high-performance computing and medical artificial intelligence.

Zhigang Zhao is currently working toward the PhD degree with the School of Computer Science, Fudan University. He is also a researcher with Shandong Computer Science Center (National Supercomputer Center in Jinan). His main research interests include distributed and parallel computing, data quality control and marine artificial intelligence.

Zhenying He received the BS, MS, and PhD degrees in computer science from the Harbin Institute of Technology, China, in 1998, 2000, and 2006 respectively. Currently, he is an associate professor with the School of Computer Science, Fudan University. His current research interests include keywords search on structured data, query processing on RDFdata, and Big Data.

Yinan Jing (Member, IEEE) received the PhD degree in computer science from Fudan University, Shanghai, China in 2007. He is an associate professor with the School of Computer Science at Fudan University. He was also a visiting scholar with the Department of Computer Science at the University of Southern California. His current research interests include Big Data analytics, database systems, spatial and temporal data management, and data security and privacy. He is a member of ACM.

X. Sean Wang (Senior Member, IEEE) received the PhD degree in computer science from the University of Southern California. He is a distinguished professor with the School of Computer Science, Fudan University, Shanghai, China. Before joining Fudan University in 2011, he was the Dorothean chair professor with the University of Vermont. His research interests include data systems and data security. He is the fellow of CCF and the member of ACM.