2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00100

RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware

Vani Nagarajan
Purdue University, USA
nagara16@purdue.edu

Milind Kulkarni
Purdue University, USA
milind@purdue.edu

***Abstract*—General Purpose computing on Graphical Processing Units (GPGPU) has resulted in unprecedented levels of speedup over its CPU counterparts, allowing programmers to harness the computational power of GPU shader cores to accelerate other computing applications. But this style of acceleration is best suited for *regular* computations (e.g., linear algebra). Recent GPUs feature new *Ray Tracing (RT)* cores that instead speed up the irregular process of ray tracing using Bounding Volume Hierarchies. While these cores seem limited in functionality, they can be used to accelerate n-body problems by leveraging RT cores to accelerate the required distance computations. In this work, we propose RT-DBSCAN, the first RT-accelerated DBSCAN implementation. We use RT cores to accelerate Density-Based Clustering of Applications with Noise (DBSCAN) by translating fixed-radius nearest neighbor queries to ray tracing queries. We show that leveraging the RT hardware results in speedups between 1.3x to 4x over current state-of-the-art, GPU-based DBSCAN implementations.**

***Index Terms*—DBSCAN, clustering, ray tracing**

I. Introduction

Graphics Processing Units (GPUs) were created to service graphics applications and accelerate parts of the rasterization pipeline. The acceleration was due to optimized floating point arithmetic using a large number of arithmetic cores. Researchers wanted to harness this massive computational capability to accelerate non-rendering applications. However, this was not straightforward as it required re-formulating problems as 3D rendering problems. Over time, the emergence of platforms such as CUDA [1] and OpenCL [2] provided a programming model for general-purpose computations on GPUs. The General-purpose GPU (GPGPU) programming model lets users offload parallelizable and compute-intensive components (*e.g.*, training a neural network) to the GPU by leveraging the programmable shader cores. Unfortunately, GPGPU acceleration using shader cores remains largely the province of *regular* applications that rely on dense loops over dense structures, such as matrix operations, convolutions, etc.

a) GPU acceleration of ray tracing: Interestingly, while GPU shader cores are well-suited for one style of graphics rendering—rasterizing—they are not at all well suited to a different, more accurate, rendering algorithm, *ray tracing*. Ray tracing (or, more accurately, *ray casting*) flips the approach of rasterizing. Rather than considering each object and how it affects one or more pixels on the screen (e.g., whether the object is visible or occluded by other objects), ray tracing considers each *pixel* in the scene and determines what color it should be based on the objects and lights it interacts with [3]. In ray tracing, a ray is cast from the source, which is typically a pinhole camera, for every pixel in the image plane. These primary rays pass through the image plane and their interactions with objects in the scene determine the color of the pixel. The ray could intersect an object in the scene, creating new reflected, refracted and/or shadow secondary rays. This hierarchy of rays is represented as a ray tree, with the primary ray as the root and the spawned secondary rays as internal nodes (secondary ray intersections can spawn tertiary rays) or leaf nodes.

Ray-object intersection tests are the biggest bottleneck in the ray tracing pipeline due to their computational intensity. As *each* ray has to be tested for intersection against *every* object in the scene, performance suffers greatly. However, it is not *always* necessary to test for intersection against every object. We can represent the objects in the scene using a Bounding Volume Hierarchy (BVH), a type of *spatial acceleration tree* [4]. A BVH, like other spatial trees, captures the relationship of objects to each other in space by recursively subdividing the space into smaller cells until the leaf cells contain bounding volumes that contain single objects. Ray-object intersection can thus be performed hierarchically: if a ray does not intersect a bounding volume, then it cannot intersect any of the subordinate bounding volumes in the tree, eliminating large numbers of intersection tests. (Section II-A describes this process in more detail.)

Unfortunately, BVH-based ray-tracing is *highly* irregular: each ray performs a tree traversal whose extent is highly input-dependent. While prior work has shown that tree traversals can be performed reasonably well on GPGPUs [5], shader cores are simply not the best-suited accelerator for BVH traversals. Hence, recent GPUs from NVIDIA and AMD have added *ray tracing (RT) cores*. These cores provide specialized hardware for building and traversing bounding volume hierarchies, significantly accelerating the process of ray tracing [6].

b) Re-purposing RT cores: In the same way that early GPU programmers looked to re-purpose shader cores to

We thank the anonymous IPDPS reviewers for their valuable feedback. We thank Dr. Eleazar Leal for providing the G-DBSCAN code and Dr. Andrey Prokopenko for answering our questions about FDBSCAN. We are grateful to Kirshanthan Sundararajah for his insightful comments that helped improve the paper. This work was funded by NSF grants CCF-1908504, CCF-1919197 and CCF-2216978.

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00100

perform non-rasterization tasks, a natural question to ask is whether RT cores can be leveraged to accelerate non-raytracing algorithms. Recent work has suggested the answer might be "yes." Wald *et. al.* [7] accelerate the task of locating which tetrahedron of a solid a query point lies in by treating the query point as the source of a ray and seeing if it intersects a tetrahedron. While this is perhaps an obvious adaptation of ray-tracing, as it is effectively ray tracing itself, later work has pushed the boundaries further. Zellman *et. al.* [8] and Evangelou *et. al.* [9] showed how to reduce the problem of finding the set of points in a fixed-radius neighborhood of a query point to ray tracing to build force-directed graphs and to do photon mapping, respectively.[1]

These papers use this reduction to write custom ray-tracing kernels that solve these problems. The algorithms essentially use one-shot invocations of ray-tracing hardware to perform the necessary distance computations and solve the problems. However, some distance-based algorithms require integrating repeated distance queries into larger programs, and hence require more careful use of the reduction.

In particular, DBSCAN [10] is a clustering algorithm that groups nearby points in space into clusters based on the distance points within the cluster are from other points in the cluster. Solving this problem requires repeatedly updating cluster definitions by repeatedly identifying nearby points, a more complex task than the "single shot" distance computations of prior RT-acceleration work. In this paper, we investigate whether RT cores can be used to accelerate this algorithm. The contributions of our paper are as follows:

- This paper introduces RT-DBSCAN, the first RT-accelerated clustering algorithm. As DBSCAN uses distance-based queries to identify neighboring points, we are able to leverage the reduction of Zellman *et. al.* [8] and Evangelou *et. al.* [9] to accelerate neighbor searches, which are a major computational bottleneck.
- We create a primitive `RT-FindNeighbor` (Details in Section 2), allowing us to easily negotiate with the ray tracing hardware and its associated programming model (the Optix Wrapper Library (OWL) [11]).
- We use `RT-FindNeighbor` to implement a `UNION-FIND`-based DBSCAN algorithm (Details in Section 3) that minimizes memory consumption. We proceed to show that RT-DBSCAN outperforms current state-of-the-art DBSCAN algorithms that have been optimized to run on GPUs.

II. Background

A. Spatial Trees

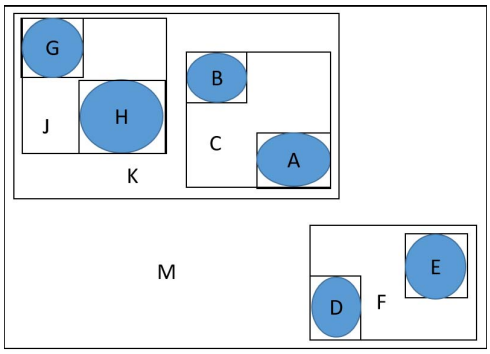
n-body problems encompass all problems where every data point needs to interact with all other *n* data points to calculate some result. Examples of n-body problems include (1) force-directed graph drawing algorithms such as Spring Embedders [12], where repulsive forces between all pairs of vertices are used to find stable graph layouts, (2) cosmology simulations, where the gravitational force exerted by stars in a galaxy is modeled, and (3) k-nearest neighbors [13], where the distance between all points in the dataset is calculated to identify the k-nearest neighbors, to name a few. The naive approach to solving n-body problems is to loop over the *n* data points and compute the effects of interaction with the other $n-1$ data points in a doubly nested loop, leading to an algorithm with $O(n^2)$ time complexity.

Barnes-Hut [14] improved the complexity by introducing a tree representation of input data points called Spatial Index Tree. They built the tree by recursively grouping nearby points using spatial subdivision. The individual points formed the leaves of the tree and internal nodes represented groupings that estimated the effect of the points contained in them. With this representation, instead of computing the effect of each point on *all* other points, we compute the effect of a *group* of points on other *individual* points. Using this spatial tree optimization, the time complexity of n-body problems reduced from $O(n^2)$ to *O(n log n)*.

1) Bounding Volume Hierarchy: The grouping of points can be done by either spatial (R-Trees, Oct-trees) or object-based subdivision of the dataset (Bounding Volume Hierarchies). Ray tracing applications rely on Bounding Volume Hierarchies (BVH) to reduce the number of ray-object intersection tests performed to determine the coloration of a pixel. Analysis of ray tracing execution times shows that about 70% of the total time is spent testing for intersections for simple scenes, and upto 95% for complex scenes [15]. The authors also state that the main reason for these percentages is the *number* of intersection tests that need to be performed.

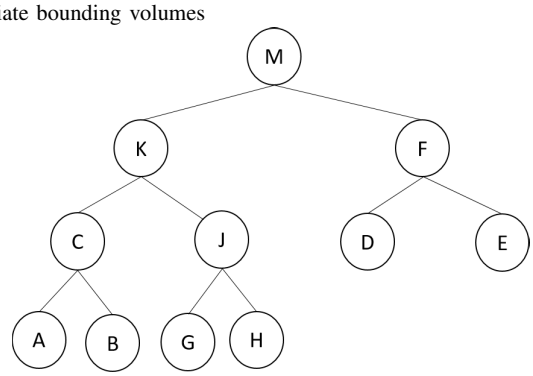
2) BVH Build: In the initial stage of the ray tracing algorithm, all objects in the scene are enclosed in bounding volumes. It is possible for the bounding volumes to overlap if the objects are sufficiently close. A *bounding volume* is a closed volume that completely encompasses one or more objects. We use Axis-Aligned Bounding Boxes (AABBs) as the bounding volumes. The BVH is built bottom-up, starting with the leaves. Bounding volumes containing individual objects in the scene are the leaf nodes of the tree. Fig 1a shows objects *A, B, G, H, D* and *E* enclosed in rectangular (shown in 2D) bounding boxes and Fig 1b shows the corresponding bounding boxes as leaves of the tree. The bounding volumes containing individual objects are then recursively grouped together to create larger bounding volumes, forming internal tree nodes *C, J, F* and *K*. This process continues until a single bounding box, *M*, encloses every intermediate and individual bounding volume. In Fig 1a, intermediate bounding volume *C* encloses bounding volumes *A* and *B*, and *J* encloses *G* and *H*. *C* is combined with bounding volume *J* to create *K*, which is further combined with *F* to create *M*, which encloses the entire scene. The corresponding hierarchical relationship is captured in Fig 1b, with *C* as the parent of *A* and *B*, *J* as the parent of *G* and *H*, *K* as the parent of *C* and *J*, and *M* being the parent of *K* and *F*.

[1]This reduction is described in more detail in Section III.



(a)

(a) 2D rectangular bounding boxes for the objects and intermediate bounding volumes



(b)

(b) Bounding Volume Hierarchy built from the bounding boxes in (a)

Figure 1: Bounding Volume Hierarchy Construction

B. Ray Tracing Cores

The addition of Ray Tracing (RT) cores to GPUs has enabled hardware-accelerated real-time ray tracing in gaming applications. These accelerators co-exist with the traditional Streaming Multi-processors (SMs), and the Optix API (Section II-B2) allows us to write code that can leverage both RT and shader cores of the GPU. The RT cores [6] accelerate Bounding Volume Hierarchy traversal and ray-triangle intersection tests, which are crucial and expensive elements of the ray tracing pipeline.

1) BVH Traversal: Given objects in the scene, the RT cores intelligently[2] build a Bounding Volume Hierarchy, similar to the description in Section II-A. The reduction in the number of intersection tests performed comes from pruning large parts of the search space by performing intersection tests on *bounding volumes* rather than individual objects. During BVH traversal, if a ray does not intersect a bounding volume, it *cannot* intersect any of the volumes contained in it and traversal does not continue down that subtree. In Fig 1, if a ray does not intersect bounding volume *K*, we need not test for intersection against *C*, *J*, *A*, *B*, *G* or *H*.

[2]Details of the actual hardware internals are not publicly available

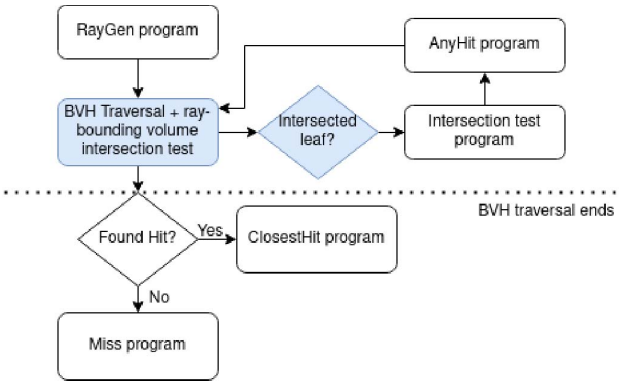


Figure 2: Optix pipeline. The components shaded in blue are performed in hardware and are not programmable. The unshaded components can be defined by the user. In the case where objects are triangles, the Intersection Test is also performed in hardware

2) Optix Programming Model: The Optix API [16] allows users to write custom shader programs in CUDA (processed as a single CUDA kernel), in addition to offloading BVH build and traversal to RT cores. Both the shader and RT cores in GPUs use the same device memory and work can be done in parallel on the two cores. If the GPU does not have an RT core, Optix programs can still be run, with BVH build and traversal being performed in software.

Optix[3] allows users to set up their scene by providing support for triangles, spheres and other user-defined geometries. Once the scene is set up, the user can define a bounding volume program to enclose objects in the scene. For geometries such as spheres, axis-aligned bounding boxes are typically used. The bounding volumes are recursively combined by the RT cores to build the BVH, as explained in Section II-A. With the BVH constructed, we can now create rays and trace their interactions with objects in the scene by traversing the BVH and performing intersection tests.

As ray tracing is an *embarrassingly parallel* problem where the color of each pixel can be computed independently, Optix allows multiple rays to be launched in parallel as separate CUDA threads. The components of the Optix pipeline are shown in Fig 2 and each ray executes the various stages in parallel. The `RayGen` program generates parallel rays with the given origin ($\vec{o}$) and direction ($\vec{d}$). The user also needs to specify the ray interval $[t_{min}, t_{max}]$, which determines the extent of the ray:

$$\vec{r} = \vec{o} + t\vec{d}, t \in [t_{min}, t_{max}]$$

The generated rays both traverse the BVH and test for intersection with bounding volumes in hardware (shown in blue in Fig 2). When the ray reaches a *leaf* bounding volume enclosing an object, the ray-*object* intersection test is performed in software or hardware, depending on the object. If the object is a triangle, the test is performed in hardware.

[3]All Optix kernels are also present in OWL [11]

Otherwise, the user specifies the `Intersection` program to be used for ray-object intersection testing. The user can optionally specify an `AnyHit` program to record information about all the intersected objects and to determine whether to continue or terminate BVH traversal. Once the BVH traversal has completed, the user can optionally call the `ClosestHit` program to identify the object closest to the ray along its path or the `Miss` program to handle cases with no ray-object intersections.

C. Density-Based Clustering of Applications with Noise

Cluster analysis is an unsupervised learning technique used to identify patterns in a dataset. Clustering techniques are widely used to provide targeted advertising to customers with similar purchase histories, identify faulty machines, detect anomalous financial transactions, and so on. *k*-means [17], a popular clustering technique due to its simplicity and scalability, picks *k* centroids and forms clusters based on whether other points in the dataset are within a permissible distance to the centroids. However, it requires that the user specify the number of clusters (*k*) to be formed and performs poorly when the dataset is noisy. Density-Based Clustering of Applications with Noise (DBSCAN) addresses the disadvantages of *k*-means, as it does not require the user to specify the number of clusters to be formed [10]. It has the added advantages of being able to form clusters of varying shapes and being unperturbed by noisy datasets.

The DBSCAN algorithm takes two parameters as inputs: ε and *minPts*, where ε is the maximum permissible distance between any two points in a cluster and *minPts* is the minimum number of points within the ε-neighborhood required to form a cluster. A point is said to be a *Core Point* if it has *minPts* neighbors within ε distance. A *Border Point* is not a core point but is *reachable* from a core point and is a part of a cluster. Reachability comes in two forms: (1) *directly reachable*, where point y is within a distance ε from core point x, and (2) *reachable*, where y is connected to core point x through one or more core points. A *Noise Point* is neither a core nor a border point.

Algorithm 1 shows the original DBSCAN algorithm. Initially, all points are considered UNASSIGNED as they have not been assigned to a cluster yet. The *FindNeighbors(p)* function in Line 2 identifies all points within ε distance of *p*. In lines 3-6, we classify the point as a Core point or a Noise point based on whether the ε neighborhood of *p* has *minPts* points. If the point is a Core point, we examine each neighbor and assign it the same Cluster_ID as the core point, as shown in Lines 9-11. If the neighbor has already been assigned a Cluster_ID, we ignore the point and move on to the next neighbor. In lines 13-16, we call the *FindNeighbors* function on the *neighbors* of point *p*. If the neighbor is a Core point, we add it to the set of neighbors and repeat the process until all points have either been assigned to a cluster or classified as noise.

Algorithm 1: Original DBSCAN

```
1  for UNASSIGNED point p do
2      Neighbors ← FindNeighbors(p)
3      if Neighbors.length < minPts then
4          p ← NOISE
5      else
6          p ← CLUSTER_ID
7          NeighborSet ← Neighbors − {p}
8          for neighbor ∈ NeighborSet do
9              if neighbor == UNASSIGNED ||
10             neighbor == NOISE then
11                 neighbor ← CLUSTER_ID
                   NewNeighbors ←
                   FindNeighbors(neighbor)
12                 if NewNeighbors.length ≥ minPts
                   then
13                     NeighborSet ←
                       NeighborSet ⋃ NewNeighbors
14                 end
15             end
16         end
17     end
18 end
```

III. Design

A. Neighbor Search

In Section II-C, we introduced an algorithm that performed density-based clustering. Algorithm 1 includes two references to a *FindNeighbors()* function that identifies all points within ε distance of a point. We generalize the query as follows:

Definition III.1. *findNeighborhood*(p, S, ε): Given a dataset S, point p and distance ε, find all points $\{x \in S \mid distance(p, x) \leq \varepsilon\}$

The $distance(x, y)$ function calculates the Euclidean distance between points x and y. The answer to this question is used to establish the ε-neighborhood to detect core points in DBSCAN. In Section II-B, we discussed how RT cores are used to determine the color of a pixel by answering the ray-object intersection query:

Definition III.2. $intersect(\vec{r}, S)$ Given a set of objects S in a scene and ray, $\vec{r}$, find all objects $\{o \in S \mid \vec{r}$ intersects $o\}$

The key question, then, is to find a way to implement *findNeighborhood* in terms of $intersect$. If we can do this, then algorithms such as DBSCAN can be readily written in terms of $intersect$ and can use the RT cores to accelerate their execution. We describe this process next, adapting a reduction proposed by prior work [8], [9].

B. Input Transformation

The key insight to the transformation is that points within a distance ε of a query point p (within p's ε-neighborhood) are the *same* points that would be contained inside a sphere

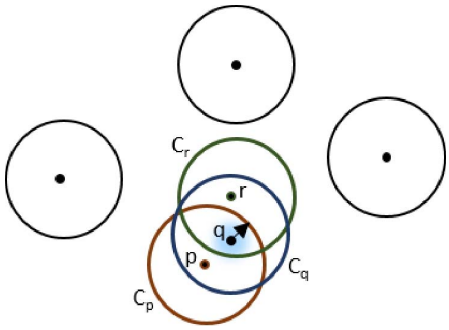


Figure 3: We expand spheres of radius (ε) around all points. We launch an infinitesimally small ray originating at query point q and see that it intersects C_p, C_q and C_r, since the origin is contained within all 3 spheres. Hence, *p, q* and *r* are q's neighbors

with origin p and radius ε. This intuitively makes sense for 3D datasets, since expanding a sphere over the query point would span across the three dimensions and accumulate *all* points within a particular query radius. However, Zellmann *et. al.* [8] propose an alternate approach where, instead of expanding a sphere only around the query point p, they expand spheres of radius ε around *all* points in the dataset. Fig 3 shows how spheres C_p, C_q and C_r are expanded over points p, q and r. We notice that the spheres overlap if their centers are in each other's ε neighborhood.

C. Putting it all together: RT-accelerated *findNeighborhood*

Now that we have a way of representing our points as spheres in a scene, we need to formulate our ray tracing query such that we can identify all neighbors of a point. Algorithm 2 presents a high-level overview of the process.

The algorithm takes a query point q, query radius ε and the dataset D as inputs. In lines 1-3, for each point p_i, we add a sphere primitive with origin p_i and radius ε to the scene. This is the input transformation process described in Section III-B and shown in Fig 3.

Using the user-specified axis-aligned bounding box program, a Bounding Volume Hierarchy is constructed in hardware to create the scene. In Line 4, we launch an infinitesimally small ray $\vec{r}$ with origin $\vec{q}$, direction $\vec{d}$ and $[t_{min}, t_{max}]$ as $[0, 1e^{-16}]$ to find all the spheres that are intersected by the ray. For example, from Fig 3, we see that such a ray launched from origin $\vec{q}$ intersects C_p, C_q and C_r. Recall that these are solid 3D spheres and a ray of infinitesimal length is sufficient to find intersections with overlapping spheres. The overlap of spheres in Fig 3 indicates that the centers of the spheres C_p and C_r are within ε distance of $\vec{q}$.

In Lines 5-9, we record all the spheres intersected by the ray and pass it through a filter to remove self-intersections (C_q in this case). When the ray traverses the BVH in hardware, it returns all the intersected *bounding volumes* in the BVH tree. As the hardware tests for intersection with *bounding volumes* and not *objects*, it is possible that the intersection test results are incorrect. Though bounding volumes completely enclose objects, they are not an *exact* fit. It is possible for the ray to intersect the bounding volume but completely miss the object contained in it. In Line 6, we perform an additional check to confirm that we intersect the *object*. Since it is also possible for bounding volumes to overlap if the dataset is dense, this filter removes any erroneous bounding volume intersections. The filtered list of intersected spheres (*NeighborList*) contains the nearest neighbors of point q.

Algorithm 2: RT-FindNeighborhood

```
   Input  : Query point q, Query radius ε, Dataset D
   Output: NeighborList
1  for p ∈ D do
2  |  S ← S ⋃ createSphere(p, ε)
3  end
4  traceRay(q⃗, d⃗, [t_min, t_max])
5  if Intersect(q, s ∈ S) then
6  |  if (dist(q, s) ≤ ε) ∧ (q ≠ s) then
7  |  |  NeighborList ← NeighborList ⋃ s
8  |  end
9  end
```

Algorithm 3: RT-DBSCAN

```
1  for point p do
2  |  neighborCount ← RT-
   |    FindNeighbors(p).length
3  |  if neighborCount ≥ minPts then
4  |  |  p ← CORE_POINT
5  |  end
6  end
7  for point p do
8  |  for n: RT-FindNeighbors(p) do
9  |  |  if n == CORE_POINT then
10 |  |  |  Union(p,n)
11 |  |  else
12 |  |  |  if n == UNCLASSIFIED then
13 |  |  |  |  critical section:
14 |  |  |  |  UNION(p,n)
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

D. RT-DBSCAN

We base our `RT-DBSCAN` algorithm on the parallel Union-Find FDBSCAN algorithm proposed by Prokopenko *et. al.* [18]. Algorithm 3 has two stages: (1) identifying Core points, and (2) updating cluster information using `union-find`. For the first stage, we use Algorithm 2 to identify each point's *neighbors*. For each point in the dataset, we launch a ray tracing query to check if the ray intersects more than *minPts* spheres. If the ray *does* intersect more than *minPts* spheres, the query point is marked as a Core point as shown in Lines 3-5.

In the second stage, we begin to form the clusters. As we did not save information about the neighbors of each point, we re-compute the Euclidean distance between points in the dataset using Algorithm 2. Though this computation is redundant and may seem inefficient, the hardware-accelerated BVH traversal prevents performance degradation. In fact, Prokopenko *et. al.* show that performance does not suffer even without hardware acceleration [18]. Additionally, this approach scales well to larger datasets as we do not run out of memory.

In Lines 7-9, we check whether the point and its neighbor are core points. If both are core points, they can be merged to form a larger cluster using the `UNION` operation. If the neighbor is not a core point, it must be a border point and we merge the border point into the core point's cluster. It is necessary to perform Line 14 atomically as border points can belong to more than one cluster. If not, the border point could be incorrectly assigned to two clusters, causing the erroneous merging of two different clusters. We use a DisjointSet [19] structure to store the rank and parent of the point. At the end of the second stage, all points that share the same parent are a part of the same cluster and all other points are noise.

IV. Implementation

We implemented RT-DBSCAN using the Optix Wrapper Library (OWL), which is built on top of Optix 7. The tests were run on an NVIDIA GeForce RTX 2060 GPU (with RT cores) with 6 GB device memory, CUDA version 10.1 and Optix 7.1.

OWL has separate programs for different components of the ray tracing pipeline: bounding box construction, intersection test, closest hit and any hit. We implemented the fixed-radius nearest neighbors search and the DBSCAN clustering phases within the Intersection program, saving the cost of calling the `AnyHit` or `ClosestHit` program.

As Optix only accepts 3D inputs, we set the z-dimension to 0 for 2D datasets and set the z-dimension of the ray direction as 1. We also explicitly disabled the `AnyHit` and `ClosestHit` programs to avoid overhead costs.

V. Evaluation

A. Datasets

We use four real-world datasets to evaluate RT-DBSCAN. As RT cores can only handle datasets with at most 3 dimensions, we chose these 2D and 3D datasets that have been widely used to evaluate DBSCAN performance([18], [20], [21]).

3DRoad The 3DRoad dataset was constructed using the road network information of North Jutland, Denmark [22]. The dataset consists of *435K* points and we use it as a 2D dataset, considering only the latitude and longitude coordinates.

NGSIM The Next Generation Simulation (NGSIM) Vehicle Trajectories dataset provides precise vehicle locations along three US highways [23]. The dataset has more than *11M* points and we use the local coordinates to construct a 2D dataset.

Porto The Taxi Service Trajectory-Prediction Challenge dataset collected trajectory data of 442 taxis in the city of Porto, Portugal [24]. The dataset has just over *1M* points and we use the 2D GPS coordinates to identify clusters.

3DIono The 3D Ionosphere dataset describes the behavior of weather in the ionosphere [25]. The dataset has just over *1M* points and we construct the 3D dataset using latitude, longitude and total electron count parameters.

B. Performance Evaluation

We compare RT-DBSCAN against three GPU-based DBSCAN implementations (though none of these use RT cores).

FDBSCAN FDBSCAN uses a parallel DisjointSet algorithm to compute clusters [18]. It has minimal memory footprint and uses Bounding Volume Hierarchies to minimize the number of distance computations.

G-DBSCAN G-DBSCAN stores ε-neighborhood information for all points in a graph and uses BFS to find connected components [26].

CUDA-DClust+ CUDA-DClust+ [27] uses the idea of incrementally growing clusters in parallel using chains from CUDA-DClust [20] but reduces memory footprint and index structure build time. As CUDA-DClust+ is strictly better than CUDA-DClust, we only evaluate the former.

For all cases, we used the authors' original source code, with the only modifications being those necessary to get the code to run on our GPU system and to handle our inputs (with the exception of FDBSCAN, which uses an early traversal termination optimization to improve execution time for single runs. In this work, we focus on typical DBSCAN use cases where the user is expected to run DBSCAN multiple times with different parameter values. We go into more detail in Section VI-B).

We do not compare against Densebox approaches such as FDBSCAN-Densebox [18], HDBSCAN-Densebox algorithms [28], [29], as they are specialized to improve performance in datasets with very high density regions. In the absence of such regions, performance remains the same or is worse. We also do not compare our performance with Mr.Scan [30] and BPS-HDBSCAN [28] as they are designed to cluster very large datasets (billion-point scale) and the incurred overhead is not amortized for smaller (thousand/million-point scale) datasets. We also do not report results against cuML's DBSCAN implementation as we were more than 3 orders of magnitude faster in all cases.

We vary the ε parameter (defined in Section II-C) and dataset size such that we include a wide range of clusters: a few large clusters, and many small clusters. We also look at a case where no clusters are formed in a dense dataset in Section V-C. We do not report results from varying *minPts* (defined in Section II-C) as it did not provide any new insights. We report execution times averaged over 10 runs.

Overall, we find that RT-DBSCAN is consistently faster in almost all cases. In particular, RT-DBSCAN is more than 2.5x faster on larger datasets. For smaller dataset sizes (Section V-B1), the performance difference between RT-DBSCAN

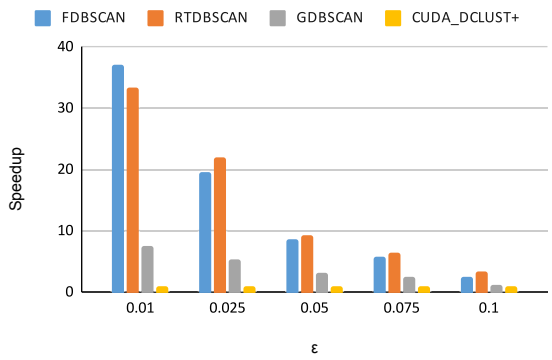


Figure 4: Speedup over CUDA-DClust+ on varying search radius (ε) for *16K* 3DRoad points

and FDBSCAN is not as pronounced due to the non-negligible BVH build time of RT-DBSCAN. We elaborate on the impact of BVH build time in Section V-D.

1) RT-DBSCAN Performance on Small Dataset Sizes: This section evaluates the four DBSCAN implementations on a small dataset (*16K* points). We found that both G-DBSCAN and CUDA-DClust+ ran out of memory on our GPU for more than *100K* points. For this reason, subsequent sections will only compare against FDBSCAN.

Overall, we found that RT-DBSCAN outperformed other approaches in most cases for *16K* points. As we kept decreasing the number of points in the dataset, we found that RT-DBSCAN was between 1.5x and 2x slower than FDBSCAN when the dataset size was less than 500.

We used *16K* points from the 3DRoad dataset and set *minPts* as 100. In Fig 4, we compare the speedup of different approaches over CUDA-DClust+. It is evident that though RT-DBSCAN is faster in most cases, speedup is minimal compared to FDBSCAN, as the overhead of setting up the ray tracing framework was not amortized by the computations. We found that the poor performance of G-DBSCAN and CUDA-DClust+[4] was due to the time taken to traverse the adjacency list and the time needed to build and traverse the index structure, respectively.

2) Impact of ε: We now turn to larger datasets, on which only FDBSCAN and RT-DBSCAN can run. We investigate clustering performance for different ε values. We vary ε while fixing *minPts* as 100 and dataset size as *1M*. We chose the first *1M* points in the datasets for clustering and averaged our results over 10 runs.

We observe from Fig 5 that RT-DBSCAN outperforms FDBSCAN in all cases. We attribute the speedup entirely to our ability to leverage hardware acceleration of BVH traversal, as FDBSCAN is also a BVH-based DBSCAN implementation, though it does not utilize RT cores.

We see a maximum speedup of 1.5x on the 3DRoad dataset as shown in Fig 5a. As we will see in Section V-D, DBSCAN execution time for small dataset sizes and small search radii is dominated by BVH build time.

In the cases of Porto and 3DIono, BVH build time of RT-DBSCAN was only 2.5x slower than FDBSCAN, allowing us to leverage the speedup in BVH traversal for fast clustering. For the Porto dataset in Fig 5b, we see a maximum speedup of 2.3x and our speedup tended to increase with increasing ε values.

For the 3DIono dataset in Fig 5c, we achieve a maximum speedup of 3.6x. As the neighborhood search radius ε becomes larger, the number of BVH traversals and intersection tests performed also increases, allowing us to realize the full potential of RT acceleration.

3) Impact of Dataset Size: Fig 6 shows how performance of RT-DBSCAN and FDBSCAN varies with the size of the input dataset. We fix the (ε,*minPts*) values as (0.05,100), (0.5,10) and (0.5,1000) for the 3DRoad, 3DIono and Porto datasets respectively. For different dataset sizes (n), we choose the first n points for clustering.

We find that RT-DBSCAN outperforms FDBSCAN on all datasets and the performance disparity is especially evident for larger dataset sizes. For the 3DRoad dataset, we see from Fig 6a that our maximum speedup is 1.37x. As 3DRoad is relatively small with a maximum of *400K* points, it is not surprising that we face issues similar to those discussed in Section V-B2, where BVH build time is not amortized by the time taken to complete the two stages of the DBSCAN algorithm.

For the Porto (Fig 6b) and 3DIono (Fig 6c) datasets, we find that we achieve maximum speedups of 2.9x and 4.1x, respectively for the maximum dataset sizes. We report the raw execution time for Porto, the largest dataset we examined, in Table I. We examine the growth rate of the execution times of RT-DBSCAN and FDBSCAN on the 3DIono dataset in Fig 7. We find that the growth rate of RT-DBSCAN's execution time is significantly slower than that of FDBSCAN as we are able to leverage hardware acceleration, showing that our approach is scalable. In general, increasing the dataset size widens the performance gap between RT-DBSCAN and FDBSCAN, as the RT hardware is designed to handle a large number of rays.

Dataset size	FDBSCAN(s)	RT-DBSCAN(s)
500K	539.85	200.82
1M	2868.1	1347.2
2M	14859.02	6264.6
4M	65935.14	23486.15
8M	282047.12	96333.7

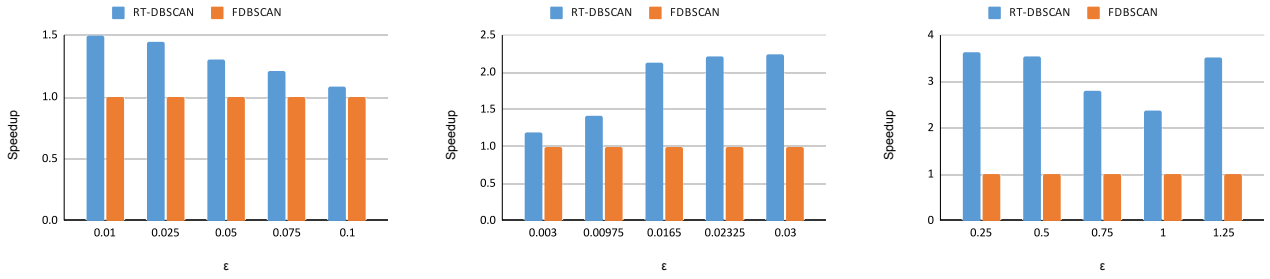
Table I: Execution time (in seconds) for Porto dataset on varying dataset size

C. RT-DBSCAN Performance on a Dense Dataset

Finally, we evaluated RT-DBSCAN on NGSIM, a very dense dataset where the number of clusters formed is 0, using the same criteria as in Sections V-B2 and V-B3.

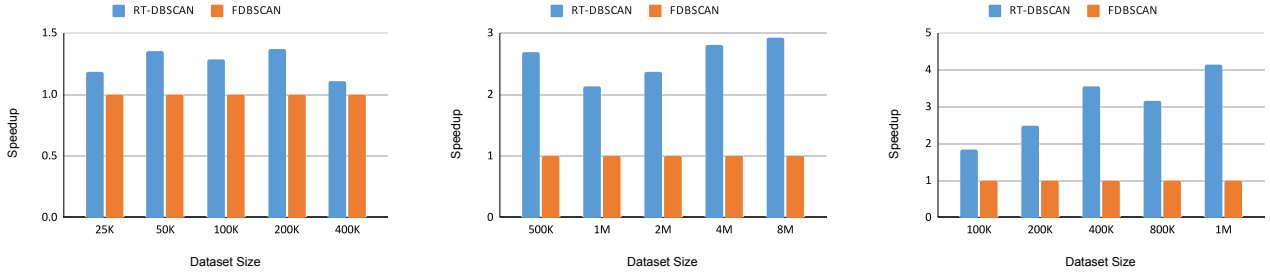
When we varied the dataset size, we found that RT-DBSCAN outperformed FDBSCAN by large margins, with

[4]We found that CUDA-DClust+ ran into memory issues on our 6GB GPU and also showed variability in clustering results between runs



(a) 3DRoad (b) Porto (c) 3DIono

Figure 5: Speedup over FDBSCAN on varying search radius (ε)



(a) 3DRoad (b) Porto (c) 3DIono

Figure 6: Speedup over FDBSCAN on varying dataset size

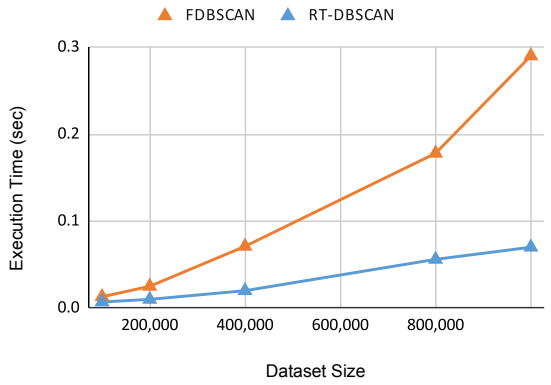


Figure 7: Scalability of execution time for 3DIono dataset

a maximum of 5500x, as shown in Fig 8b. Table III shows the raw execution times.

Search radius (ε)	FDBSCAN(s)	RT-DBSCAN(s)
0.0001	64.72	0.0257
0.00025	64.77	0.0259
0.0005	64.74	0.0259
0.00075	64.71	0.026
0.001	64.74	0.0259

Table II: Execution time (in seconds) for NGSIM dataset on varying search radius (ε)

On varying ε with *minPts* as 100 and dataset size as *1M*, we found that RT-DBSCAN was nearly 2500x faster than FDBSCAN, as shown in Fig 8a. Table II shows raw execution times for different ε values. The execution times of both FDBSCAN and RT-DBSCAN did not significantly change for different ε as the dataset was still dense for these different radii.

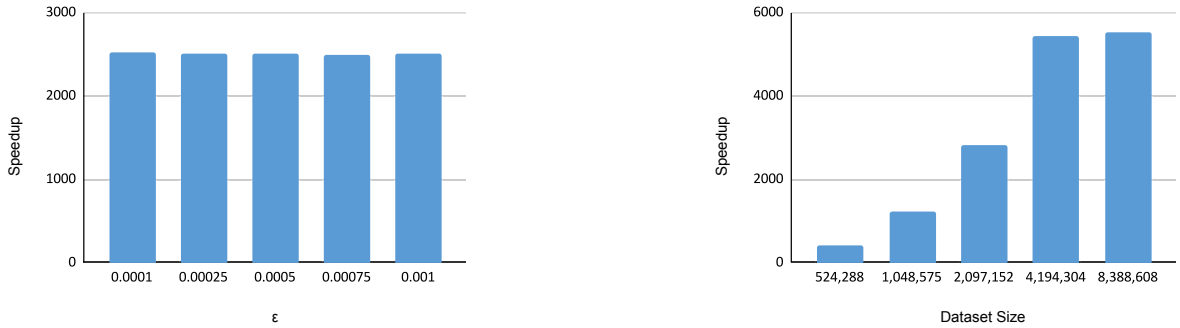
Dataset size	FDBSCAN(s)	RT-DBSCAN(s)
500K	12.7	0.03
1M	72.8	0.06
2M	364.6	0.13
4M	1631.4	0.3
8M	6964.1	1.26

Table III: Execution time (in seconds) for NGSIM dataset on varying dataset size

On analyzing the output, we found that the RT hardware made relatively few calls to the intersection program. As the specifics of BVH construction and traversal in RT hardware are unclear, we speculate that the hardware was able to construct the BVH such that we were able to prune large parts of the search space and minimize the number of intersection tests required. As we will discuss in Section VI-C, having access to the workings of the hardware internals would help explain our results a lot better.

D. Runtime Analysis of RT-DBSCAN

In Section III-B, we discussed how data points are converted to spheres so that RT cores can build and traverse the BVH in hardware. Though this helps attain our objective of converting the nearest neighbor problem to a ray tracing query, it comes at a cost. Building a BVH from spheres for a ray tracing



(a) Speedup on varying search radius (ε)

(b) Speedup on varying dataset size

Figure 8: Speedup over FDBSCAN on varying ε and dataset size for NGSIM

application is much more complex and time-consuming than building a spatial tree for data points. The Optix builder performs memory compaction, invokes bounding box routines and other ray-tracing-specific operations that add to the BVH build time.

The general trend we observed was that BVH build time dominated the total execution time for smaller datasets and cases where ε was small, as fewer BVH traversals and intersection tests needed to be performed. For example, we consider the first 1 million points from 3DIono dataset with ε = 0.25, and *minPts* = 100, similar to Section V-B2. We found that RT-DBSCAN was 3.6x faster than FDBSCAN overall. Breaking down the execution time, we found that the time taken by RT-DBSCAN to perform clustering operations *after* BVH build was 6.4 ms for the Core point identification phase and 6.6 ms for the cluster formation phase. In total, RT-DBSCAN only spent 48% of total execution time) on actual clustering operations. On the other hand, FDBSCAN spent 0.118 seconds (94% of total execution time) on clustering operations. This shows us that, on average, RT-DBSCAN is more than 9x faster than FDBSCAN in performing the actual clustering operations!

VI. Discussion

A. Hardware Limitations

A major disadvantage of using RT cores to accelerate distance computations is that the dimensionality of the dataset can be at most three. Indeed, the RT cores themselves expect the dataset to be *exactly* three dimensions. Despite this limitation, we note that there are many important real-world 2D and 3D datasets such as Geospatial data, point clouds, and object geometries, and important distance algorithms such as DBSCAN, computing normals, and filtering point cloud noise that use distance searches over these datasets. Indeed, we note that most of the prior DBSCAN works evaluate their approaches on 2D geospatial datasets [18], [20], [21]. It is possible to reduce the number of dimensions in the dataset using dimensionality reduction techniques such Principal Component Analysis, though this introduces approximation.

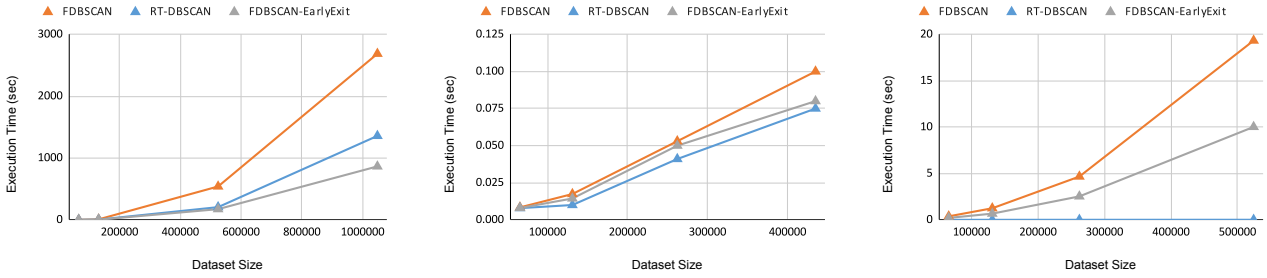
B. Impact of early traversal termination

FDBSCAN uses an optimization where it stops BVH traversal when *minPts* neighbors are found in the `FindNeighbors` function [18]. However, the Optix API, based on which OWL is built, does not allow BVH traversal termination unless the `Intersection` kernel makes an additional call to the `AnyHit` kernel. As this can incur significant overhead, RT-DBSCAN does not attempt to perform early traversal termination.

Though the early exit optimization works very well for cases where the user is only expected to run DBSCAN *once*, it is, in practice, more useful to record the number of neighbors of every point by allowing the BVH traversal to run its course. By saving the number of neighbors of each point, we do not have to re-run core point identification phase (Stage-1 of Algorithm 3) for any subsequent DBSCAN runs where the user changes the *minPts* parameter.

In Figs 9, we compare RT-DBSCAN to FDBSCAN *with* early termination, as well as FDBSCAN *without* while varying dataset size for fixed (*minPts*, ε) values. As expected, using early termination guarantees better performance as it often performs orders of magnitude fewer distance computations. This is especially true when *minPts* is very small and BVH traversal can stop very early. From our experiments, it is evident from Fig 9a that for the Porto dataset, using early exit improves performance of FDBSCAN-EarlyExit by 3x compared to FDBSCAN *without* and by 1.5x compared to RT-DBSCAN for larger dataset sizes.

However, in other cases, even the early-exit optimization is not enough to overcome RT-DBSCAN's superior performance. For example, note that RT-DBSCAN outperforms FDBSCAN-EarlyExit on the 3DRoad dataset (Fig 9b) and vastly outperforms it on the NGSIM dataset (Fig 9c). We explain our findings by noting that the cost of additional intersection tests is hidden by the acceleration from RT cores. We especially note that even though the early exit optimization is able to leverage the density of the NGSIM dataset to improve performance substantially, RT-DBSCAN's ability to massively prune the search space is even more useful.



(a) 3DRoad (b) Porto (c) NGSIM

Figure 9: Impact of early traversal termination on execution time

C. Extensions to the Ray Tracing API

We believe that it would be advantageous to have a certain degree of control over the hardware BVH traversal. For applications such as Barnes-Hut [14], it is necessary to access and update the intermediate nodes of the BVH to estimate the effect of the enclosed volumes and this is not possible with the current setup of the Optix API. We leave the Barnes Hut implementation as future work.

In this work, we leveraged the acceleration from the hardware BVH build and traversal. However, from Section II-B, we know that the RT cores can also accelerate ray-triangle intersection tests. Indeed, Wald *et. al.* find that using hardware-accelerated intersection tests *in addition* to hardware-accelerated BVH traversal can produce substantial performance gains [7].

In our nearest neighbors algorithm (Algorithm 2), we expand spheres around the points in the dataset and designate all points that fall within the sphere as neighbors of that point. We performed some experiments to see if we could approximate the spheres using triangles to leverage the hardware acceleration. As the ray-triangle intersection tests were done in hardware, we had to call the `AnyHit` kernel to collect the intersected points. We found that using triangles resulted in 2x to 5x performance degradation due to the overhead cost associated with calls to the `AnyHit` kernel. If there was an extension to the hardware such that the intersected points can be returned without using the costly `AnyHit` kernel, there is massive potential for performance improvement from leveraging hardware-accelerated ray-triangle intersection testing.

VII. Related Work

a) Using RT cores for non-Ray-Tracing Applications: Wald *et. al.* first used RT cores to accelerate non-ray-tracing programs [7]. They formulated the problem of identifying a point's location in a tetrahedral mesh as a ray tracing problem by declaring the meshes as 3D objects in a scene and tracing rays originating at the query point. They show how leveraging both hardware BVH traversal and ray-triangle intersections resulted in upto 6.5x speedup over other CUDA implementations. Morrical *et. al.* used RT cores to successfully accelerate the unstructured mesh point location problem [31]. Zellmann *et. al.* proposed a mapping of the fixed-radius nearest neighbor query to a ray tracing query for the Spring Embedders force-directed graph drawing algorithm [8]. Evangelou *et. al.* used the nearest neighbor mapping to solve the k-nearest neighbors problem [9]. Zhu proposed query re-ordering and partitioning algorithms to improve ray coherence and minimize the number of intersection tests performed [32]. We note that adding these optimizations to RT-DBSCAN would further improve performance.

b) DBSCAN: Ester *et. al.* introduced the DBSCAN algorithm in 1996 to identify clusters of arbitrary shapes based on dense regions in the dataset [10]. Although DBSCAN is an inherently sequential algorithm (Algorithm1), researchers have exploited GPU parallelism to accelerate DBSCAN. Thapa *et. al.* exploited parallelism by having multiple CPU threads perform ε-neighbor distance computations in parallel [33]. Andrade *et. al.* proposed G-DBSCAN, where they built a graph over the dataset and performed parallel Breadth First Searches to mark reachable points as belonging to the same cluster [26]. However, the memory required to store and maintain the graph structure affects the scalability of G-DBSCAN. Böhm *et. al.* introduced CUDA-DClust, which used a spatial index structure to incrementally grow clusters in parallel [20]. Poudel *et. al.* proposed CUDA-DClust+ which improved on CUDA-DClust by building the index structure on the GPU instead of CPU and reducing communication overhead [27]. However, CUDA-DClust+ requires a significant amount of time for index construction and suffers when the size of GPU memory is small. Prokopenko *et. al.* proposed FDBSCAN and FDBSCAN-DenseBox that use Bounding Volume Hierarchy with UNION-FIND for clustering [18]. Both FDBSCAN and FDBSCAN-DenseBox avoid memory issues as they do not store any neighbor information. FDBSCAN-DenseBox, similar to [28], [30], superimposes a Cartesian grid-based indexing to identify dense regions and reduce the number of distance computations in dense boxes.

VIII. Conclusion

In this work, we implemented RT-DBSCAN, where we accelerated the nearest neighbor searches in DBSCAN using Ray Tracing cores. We found that the hardware acceleration led to performance improvements by as much as 4.5x over current state-of-the-art GPU-based DBSCAN implementations. Future work entails removing the fixed-radius constraint for neighbor

searches to accelerate a wider range of applications. It would also be interesting to see if RT cores can be used to accelerate more general tree traversal algorithms.

REFERENCES

[1] NVIDIA, "Cuda," 2007. [Online]. Available: https://developer.nvidia.com/cuda-zone

[2] K. Group, "Opencl," 2009. [Online]. Available: https://www.khronos.org/opencl/

[3] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, no. 6, p. 343–349, jun 1980.

[4] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.

[5] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for gpu execution of tree traversals," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013.

[6] NVIDIA, "Nvidia turing architecture whitepaper," 2021. [Online]. Available: https://gpltech.com/wp-content/uploads/2018/11/NVIDIA-Turing-Architecture-Whitepaper.pdf

[7] 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 *High-Performance Graphics - Short Papers*. The Eurographics Association, 2019.

[8] S. Zellmann, M. Weier, and I. Wald, "Accelerating force-directed graph drawing with rt cores," in *2020 IEEE Visualization Conference (VIS)*, 2020, pp. 96–100.

[9] I. Evangelou, G. Papaioannou, K. Vardis, and A. A. Vasilakis, "Fast radius search exploiting ray tracing frameworks," *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, February 2021.

[10] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." AAAI Press, 1996, pp. 226–231.

[11] I. Wald, N. Morrical, and H. E, "Owl-the optix 7 wrapper library," 2020.

[12] P. Eades, "A heuristic for graph drawing," 1984.

[13] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, pp. 175–185, 1992.

[14] J. E. Barnes and P. Hut, "A hierarchical O(n-log-n) force calculation algorithm," *Nature*, vol. 324, p. 446, 1986.

[15] A. Fujimoto, T. Tanaka, and K. Iwata, "Arts: Accelerated ray-tracing system," *IEEE Computer Graphics and Applications*, vol. 6, no. 4, pp. 16–26, 1986.

[16] NVIDIA, "Nvidia optix 7.4 programming guide," 2021.

[17] Y. Li and H. Wu, "A clustering method based on k-means algorithm," *Physics Procedia*, vol. 25, pp. 1104–1109, 2012, international Conference on Solid State Devices and Materials Science, April 1-2, 2012, Macao.

[18] A. Prokopenko, D. Lebrun-Grandié, and D. Arndt, "Fast tree-based algorithms for DBSCAN on gpus," *CoRR*, vol. abs/2103.05162, 2021. [Online]. Available: https://arxiv.org/abs/2103.05162

[19] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM J. Comput.*, vol. 2, pp. 294–303, 1973.

[20] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther, "Density-based clustering using graphics processors," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, ser. CIKM '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 661–670.

[21] H. Mustafa, E. Leal, and L. Gruenwald, "An experimental comparison of gpu techniques for dbscan clustering," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 3701–3710.

[22] M. Kaul, B. Yang, and C. S. Jensen, "Building accurate 3d spatial networks to enable next generation intelligent transportation systems," in *2013 IEEE 14th International Conference on Mobile Data Management*, vol. 1, 2013, pp. 137–146.

[23] transport.gov, "Next generation simulation (ngsim) vehicle trajectories and supporting data," 2021. [Online]. Available: https://www.opendatanetwork.com/dataset/data.transportation.gov/8ect-6jqj

[24] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas, "Predicting taxi–passenger demand using streaming data," *IEEE Transactions on Intelligent Transportation Systems*, vol. 14, no. 3, pp. 1393–1402, 2013.

[25] V. Pankratius, A. Coster, J. Vierinen, P. Erickson, and B. Rideout, "Gps data processing for scientific studies of the earth's atmosphere and near-space environment," pp. 1–12, 01 2015.

[26] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-dbscan: A gpu accelerated algorithm for density-based clustering," *Procedia Computer Science*, vol. 18, pp. 369–378, 2013, 2013 International Conference on Computational Science.

[27] M. Poudel and M. Gowanlock, "Cuda-dclust+: Revisiting early gpu-accelerated dbscan clustering designs," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 354–363.

[28] M. Gowanlock, "Hybrid cpu/gpu clustering in shared memory on the billion point scale," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 35–45.

[29] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius, "A hybrid approach for optimizing parallel clustering throughput using the gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 766–777, 2019.

[30] B. Welton, E. Samanas, and B. P. Miller, "Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–11.

[31] N. Morrical, W. Usher, I. Wald, and V. Pascucci, "Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing," *2019 IEEE Visualization Conference (VIS)*, pp. 256–260, 2019.

[32] Y. Zhu, "Rtnn: Accelerating neighbor search using hardware ray tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 76–89.

[33] R. J. Thapa, C. Trefftz, and G. Wolffe, "Memory-efficient implementation of a graphics processor-based cluster detection algorithm for large spatial databases," in *2010 IEEE International Conference on Electro/Information Technology*, 2010, pp. 1–5.