RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing

Yuhao Zhu
yzhu@rochester.edu
University of Rochester
Rochester, New York, USA

Abstract

Neighbor search is of fundamental importance to many engineering and science fields such as physics simulation and computer graphics. This paper proposes to formulate neighbor search as a ray tracing problem and leverage the dedicated ray tracing hardware in recent GPUs for acceleration. We show that a naive mapping under-exploits the ray tracing hardware. We propose two performance optimizations, query scheduling and query partitioning, to tame the inefficiencies. Experimental results show 2.2× – 65.0× speedups over existing neighbor search libraries on GPUs. The code is available at https://github.com/horizon-research/rtnn.

CCS Concepts: • **Computing methodologies** → **Ray tracing**; **Graphics processors**; • **Information systems** → **Nearest-neighbor search**; • **Theory of computation** → **Nearest neighbor algorithms**.

Keywords: neighbor search, ray tracing, OptiX, BVH

1 Introduction

3D Neighbor search is a building block widely used in many application domains such as computer vision, graphics, and scientific computing. Due to its fundamental importance, fast neighbor search has long been a subject of much research, including many CPU [17, 19, 26] and GPU libraries [15, 16, 45] as well as hardware accelerators [32, 44].

A fundamental trade-off neighbor search algorithms make is one between work efficiency and hardware efficiency. On one hand, grid-based algorithms are work-inefficient, as they perform (limited) exhaustive search over a grid, but exhaustive searches are hardware friendly and can be easily parallelized. On the other hand, tree-based algorithms (e.g., Octree, k-d tree) are work-efficient by hierarchically pruning the search space, but tree traversal is hardware-inefficient, introducing irregular control flows and memory accesses.

This paper argues that neighbor search can be made both work-efficient and hardware-friendly — by using Bounding Volume Hierarchy (BVH) tree as the basic data structure. This design decision allows us to formulate neighbor search as a ray tracing problem (Section 3.1), which, critically, has dedicated hardware support (for BVH traversal) in recent GPUs such as Nvidia's Turing (and later) architecture.

Unfortunately, a naive mapping from neighbor search to ray tracing does not effectively exploit the ray tracing hardware, and is in fact work- and hardware-inefficient (Section 3.2). We quantitatively show two performance-limiting factors : 1) unmanaged query-to-ray mapping, which leads to control-flow divergences, and 2) excessive tree traversals stemming from the monolithic BVH construction. These characterizations motivate us to propose two optimizations.

First, we propose a query scheduling strategy to tame the control-flow divergence by mapping spatially-close queries to nearby rays (Section 4). We show that this scheduling algorithm, in itself, can be formulated as a truncated ray tracing problem and, thus, is extremely efficient to execute.

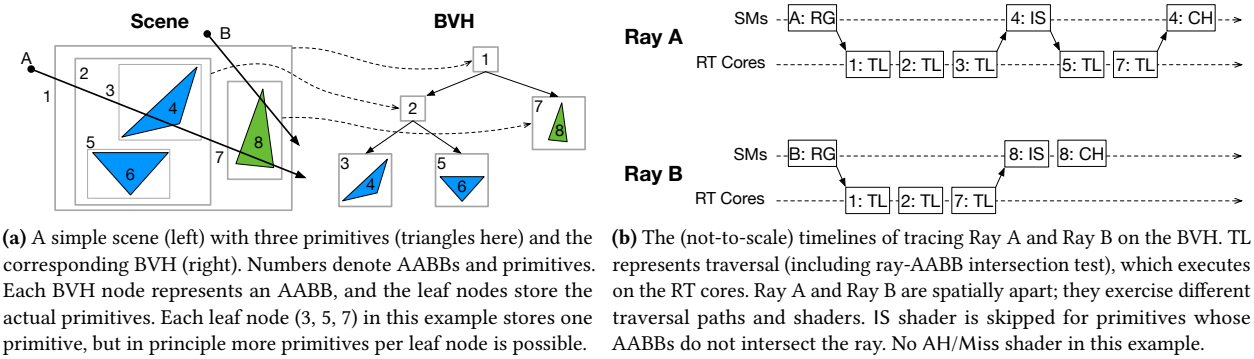
Second, we propose a lightweight query partitioning algorithm to aggressively suppress tree traversals (Section 5.1). Instead of using a single BVH for all the queries/rays, we partition queries such that each partition has a unique BVH that minimizes tree traversals for that partition. Query partitioning, however, comes with the overhead of extra BVH constructions. We propose an algorithm that bundles the partitions to minimize the execution time (Section 5.2).

On the RTX 2080 GPU, we show 2.2× to 44.0× speedups compared to optimized CUDA neighbor search and a 65.0× speedup over unoptimized ray tracing-accelerated neighbor search. The contributions of the paper are the following:

- We describe a systematic way to map neighbor search to ray tracing, and quantitatively demonstrate two key performance bottlenecks of such a mapping.
- We introduce two optimizations, query scheduling and query partitioning, that mitigate the bottlenecks and effectively exploit the ray tracing hardware.
- We provide an open-source implementation of our algorithm, which achieves 2.2× – 65.0× speedup over existing GPU neighbor search algorithms.

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508409



(a) A simple scene (left) with three primitives (triangles here) and the corresponding BVH (right). Numbers denote AABBs and primitives. Each BVH node represents an AABB, and the leaf nodes store the actual primitives. Each leaf node (3, 5, 7) in this example stores one primitive, but in principle more primitives per leaf node is possible.

(b) The (not-to-scale) timelines of tracing Ray A and Ray B on the BVH. TL represents traversal (including ray-AABB intersection test), which executes on the RT cores. Ray A and Ray B are spatially apart; they exercise different traversal paths and shaders. IS shader is skipped for primitives whose AABBs do not intersect the ray. No AH/Miss shader in this example.

Fig. 1. A simple BVH example and the (not-to-scale) execution timelines of tracing two rays on the BVH. Abbreviations: RG (Ray Generation), TL (Traversal), IS (Intersection), CH (Closest-Hit), AH (Any-Hit); see Figure 3.

2 Background

We first define the scope of neighbor search that is considered in this paper (Section 2.1). We then briefly overview the ray tracing algorithm (Section 2.2) pertaining to this paper, and introduce the programming and hardware support for ray tracing in Nvidia's recent GPUs (Section 2.3).

2.1 Scope of Neighbor Search This Paper Targets

Dimensionality in Neighbor Search Different applications require neighbor searches in different dimensions. Due to the curse of dimensionality, it is well-known that search algorithms used for low dimensions (three or lower) are different from that for high-dimensional searches [5, 41, 42].

We target neighbor search in low-dimensional (three or lower) space, which is prevalent in engineering and science fields (e.g., computational fluid dynamics, graphics, vision), because they deal with physical data such as particles and surface samples that inherently reside in the 2D/3D space.

Neighbor Search Variants Two types of neighbor search exist: fixed-radius search (a.k.a., range search) and K nearest neighbor search. RTNN optimizes for both types.

Fixed-radius search concerns with returning all the neighbors within a fixed radius r. In practice, the maximum amount of returned neighbors is bounded in order to bound the memory consumption and to interface with downstream tasks, which usually expect a fixed amount of neighbors.

KNN search concerns with returning the nearest K neighbors of a query. In practice, the returned neighbors are bounded by a search radius, beyond which the neighbors are discarded. This is because the significance of a neighbor (e.g., the force that a particle exerts on another) is minimal and of little interest when it is too far away.

Therefore, for both types of search we assume a search interface that provides a search radius r and a maximum neighbor count K, consistent with the interface of existing neighbor search libraries. We can easily emulate an unbounded KNN search by providing a very large r and emulate an unbounded range search by providing a very large K.

2.2 Ray Tracing Algorithm and Data Structure

Graphics rendering algorithms are moving toward ray tracing. We briefly review algorithmic components relevant to our paper, and refer interested readers to Pharr et al. [30] and Glassner [13] for a more comprehensive treatment.

Intersection Test The crux of ray tracing is to calculate the closest intersection of a ray and the scene, which is usually represented by a set of geometric primitives such as triangles and spheres. The intersection test dominates the rendering time [39], and is the prime target for optimization.

The intersection test is done by partitioning the primitives in the scene. In particular, primitives are represented by their bounding volumes, which are usually Axis-Aligned Bounding Boxes (AABBs). The AABBs are then hierarchically organized as a tree, which is called the Bounding Volume Hierarchy (BVH). Figure 1a shows the BVH of a simple three-primitive (triangles here) scene. The leaf nodes in the BVH are the AABBs that store the actual scene primitives, and the interior nodes are the AABBs that enclose other AABBs.

With the BVH, finding the closest hit for a ray becomes a tree traversal problem. At every node, we test whether the ray intersects with the AABB of that node. If the ray does not intersect the node's AABB, the entire subtree beneath that node can be skipped, because all the primitives enclosed by that AABB are guaranteed to be not intersected by the ray. For instance, Ray A in Figure 1a does not intersect AABB 5, so primitive 6 can be skipped. Otherwise, we further test the ray against all the AABBs enclosed by the node. Note that AABB 7 and primitive 8 will also be skipped after hit test with primitive 4, which provides a closer hit than AABB 7.

When the ray reaches a leaf node, we test the ray against all the enclosed primitives and record the closet hit point (so far). This process continues until we have traversed the entire tree, at which point the closet hit is reported.

Intersection Conditions It is vital to understand the conditions under which an AABB is considered to be intersected by a ray, which our algorithm relies on. Formally, a ray is a line $P(t)$ parameterized by two parameters: the

origin O and the direction vector $\mathbf{d}$ [36]:

$$P(t) = O + t\mathbf{d}. \quad (1)$$

While in theory t can take any value, providing a full line, in practice we are often interested in only a segment of the ray, which is described by bounding $t \in [t_{min}, t_{max}]$. A ray intersects an AABB if one of the following two conditions is met. Figure 2 illustrates the two conditions.

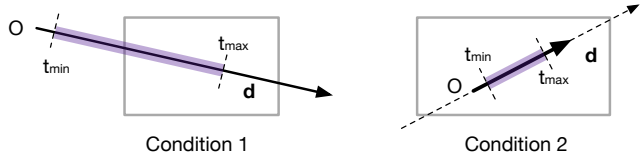


Fig. 2. Two conditions for ray-AABB intersection.

1. when a ray hits the bounds of the AABB (the six faces) and the t value of hit point is within $[t_{min}, t_{max}]$;
2. when the origin of the ray is within the AABB, *even if* the intersected t value is beyond $[t_{min}, t_{max}]$.

As we will show later, we rely on Condition 2 to implement neighbor search. Condition 2 might initially seem odd. It is necessary because when a ray originates from within an AABB, it is possible that the ray *might* intersect children AABBs that are enclosed in the current AABB. Therefore, we must treat that ray as intersecting such that the ray is allowed to further test against the enclosed (children) AABBs.

2.3 Hardware Support and Programming Model for Ray Tracing on Nvidia GPUs

While using BVH to prune the search space is work-efficient, tree traversal is irregular, exhibiting frequent control-flow and memory divergences (e.g., per-thread stack management). Nvidia's recent Turing (and later) GPU architecture is equipped with dedicated hardware, i.e., the RT cores, to accelerate BVH tree traversal [6]. We briefly review the architectural and programming details that are relevant to developing neighbor search algorithms.

Hardware The RT cores are essentially tightly-coupled accelerators sitting alongside the conventional Stream Multiprocessors (SMs). The RT cores and the SMs share the same device memory — an important feature that allows us to, in one program, use SMs for regular parallel computations and use the RT cores for ray tracing.

Programming Model To leverage the RT cores, we use the OptiX programming model [28] from Nvidia as it natively supports the RT cores, but the same principles apply to other graphics APIs as well (e.g., Vulkan and DirectX).

In OptiX, ray tracing starts by building the BVH; this stage executes on the SMs and is non-programmable. Once the BVH is built, the ray tracing pipeline is launched. OptiX presents to programmers a fixed pipeline organization, but exposes interfaces for user-defined programs (a.k.a., *shaders*

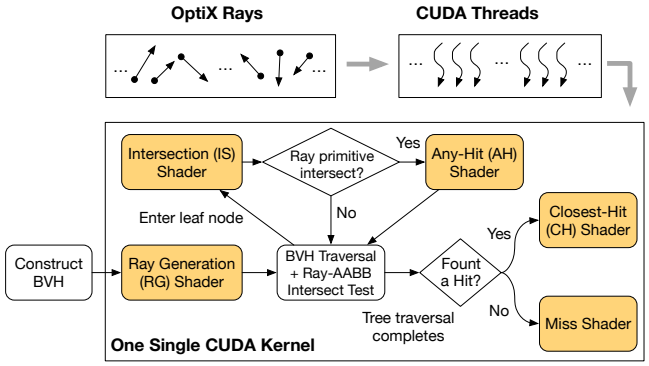


Fig. 3. The simplified programming model of OptiX. Shaded components are the programmable shaders. AH/CH/Miss shaders are optional. All the shaders are compiled into one single CUDA kernel, which executes on the SMs. Each ray is mapped to a CUDA thread. BVH traversals, including the ray-AABB intersection tests, are accelerated on the RT cores.

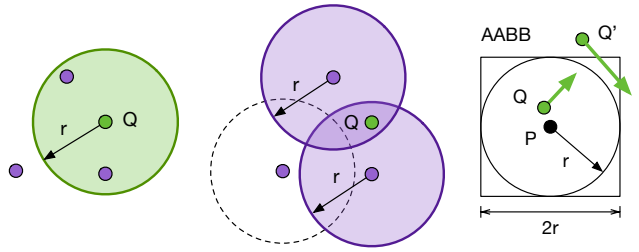
in the graphics parlance) to control different stages in the pipeline. It is these programmable shaders that provide the opportunity for implementing algorithms beyond rendering. Figure 3 shows a simplified view of the pipeline, where the shaded components are the programmable shaders.

OptiX shaders are essentially callback functions triggered at different phases during BVH traversal. The Ray Generation (RG) shader, the entry to the pipeline, generates rays by specifying the ray origins and directions. During traversal, whenever leaf nodes of the BVH are encountered the Intersection (IS) shader is called, which performs the ray-primitive intersection test. If an intersection is found, the Any-Hit (AH) shader can be called to process the hit information or to terminate the traversal. When the entire traversal finishes, either the Closest-Hit (CH) shader or the Miss shader could be called depending on whether a hit is found.

Execution Model OptiX provides a "Single Instruction Multiple Rays" execution model: every shader in the pipeline (Figure 3) is executed by every single ray. Under the hood, all the shaders are compiled into one single CUDA kernel executing on the SMs; each ray is mapped to a CUDA thread.

Figure 1b illustrates the execution timelines of tracing the two rays in the BVH in Figure 1a. Each ray starts from the RG shader on the SMs. The control then transfers to the RT cores for the BVH traversal, during which if a shader is triggered the hardware traversal is interrupted and the control is transferred back to the SMs.

Terminology Clarification In graphics parlance, identifying the intersection of a ray and the scene is called *ray casting*; ray tracing refers to *recursive* ray casting; the recursion is necessary for realistic shading [18, 43]. In Nvidia's post-Turing GPUs, it is ray casting that is being accelerated in hardware. How (and whether) to implement recursion is left to the OptiX programmers (usually in the IS shader). In this sense, our paper maps neighbor search to a ray casting



(a) Original neighbor search for query Q. **(b)** Reversed neighbor search. **(c)** Testing if Q is in P's r-sphere.

Fig. 4. Formulating neighbor search as ray tracing. (a): Searching points that are within r from the query point Q; (b) Testing whether Q is within r from all other points; (c) Testing whether Q is in P's r-sphere can be done by tracing a very short ray from Q. Using long rays would lead to false positives in ray-AABB tests (e.g., Q').

problem rather than a ray tracing problem. That is, a query will not recursively spawn new queries.

3 Neighbor Search as Ray Tracing: Basic Idea and Performance Characterizations

We describe how to formulate neighbor search as ray tracing (Section 3.1). We then quantitatively demonstrate two sources that dictate the performance of our algorithm (Section 3.2), i.e., ray coherence and the AABB size, which motivate our optimizations that follow.

3.1 The Basic Idea

Distance measure in Euclidean spaces is commutative: testing whether a point P is within a distance r from a query Q is equivalent to testing when Q is within the same distance r from P. Leveraging this property, we can turn the neighbor search problem around: instead of finding all the points within a distance r from a query point Q (shown in Figure 4a), we test whether Q is within r from all other points. This inverse test can be done by generating spheres with a radius r around all the points, and returning points whose spheres enclose Q (shown in Figure 4b).

Identifying r-radius spheres that enclose Q is done in two steps. Figure 4c illustrates the idea.

- Step 1 (AABB test): For each sphere, we first generate an AABB that circumscribes the sphere (i.e., the tightest AABB that just encloses the sphere), and test whether Q resides in the AABB. All the spheres that fail this test can be skipped.
- Step 2 (Sphere test): Otherwise, we further test whether Q resides in the sphere, by comparing the distance between Q and the sphere center P with r. If successful, we can record P as a neighbor of Q under range search or operate a priority queue under KNN search.

Critically, the algorithm above can be mapped as a ray tracing problem. This is done by building a BVH from the AABBs of all the points and casting a ray originated from Q. Step 1 essentially uses the ray to traverse the BVH and discards points whose AABBs do not intersect the ray. Step 2 is a ray-primitive intersection test, where the primitives are spheres. Step 1 is completely accelerated by the RT cores, and Step 2 can be implemented as an IS shader in OptiX.

Casting the Ray In theory, the ray from Q can be of an arbitrary length. However, using a ray with a long length could lead to false positives in Step 1. Figure 4c illustrates this scenario using query Q', whose ray intersects with P's AABB (i.e., Step 1 test passes and Step 2 test is performed), but does not reside in P's sphere.

While this false positive does not affect the correctness, as Step 2 will eventually reject P as a neighbor of Q', it does lead to redundant computation in Step 2, which is much more expensive than Step 1 — an order of magnitude slower in our experiments: Step 2 requires floating point multiplications and potentially manipulates a priority queue, whereas Step 1 requires only bounds comparison.

Therefore, we generate very short rays from the queries by setting t_{min} to be 0 and t_{max} to be a small number (e.g., 1e-16 in our implementation). With this, only rays whose origins reside in an AABB will trigger Step 2. Note that the ray-AABB intersection tests now will mostly rely on Condition 2 (Figure 2) since the rays are very short. With short rays, the ray direction can be arbitrary. We set all ray directions to [1, 0, 0] in our implementation.

Benefits Our algorithm is work-efficient, because it prunes the search space by omitting points whose AABBs do not contain the query point. It is similar, in spirit, to other tree-based algorithms using k-d trees [25, 48] and Octrees [3, 29]. Using BVH, however, let us leverage the ray tracing hardware in recent GPUs to accelerate the irregular tree traversals, which would not be available if other data structures were used.

Summary We show the pseudo-code of the range search algorithm in Listing 1. KNN search is similar except the IS shader would operate a priority queue.

Listing 1. Pseudo-code of range search as ray tracing.

```
input: points, queries, radius, K;

buildBVH(points, radius) {
  foreach point in points
    create an AABB {center=point; width=2*radius};
  return (generate BVH from all AABBs);
}

/* code on the host */
bvh ← buildBVH(points, radius);
//launch pipeline, starting from the RG shader
traceRays(queries, K, radius, bvh);

/* shader code */
```

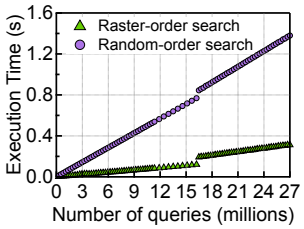


Fig. 5. Searches with ordered (coherent) rays (raster-scan order here) are 4× faster than searches with incoherent, randomly-ordered rays.

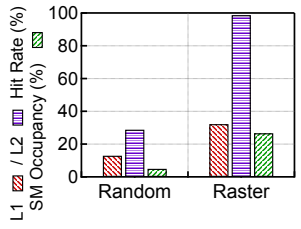


Fig. 6. Ordered searches are faster because coherent queries/rays lead to higher L1/L2 cache hit rate and higher SM occupancy.

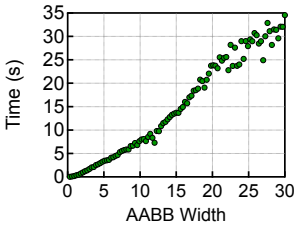


Fig. 7. Search time correlates with the AABB size, which dictates the work on both the SMs and the RT cores.

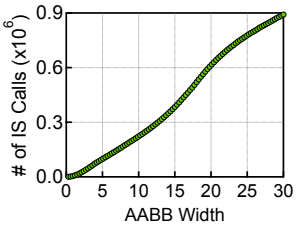


Fig. 8. The number of IS calls increases with the width of the AABBs in the BVH. Rays intersect more AABBs when AABBs are larger.

```
15 RG_Shader() {
16   //OptiX API to get current ray ID;
17   rayId ← optixGetLaunchIndex().x
18   ray {origin=queries[rayId], direction=[1,0,0],
         tmin=0, tmax=1.e-16f};
19   //trigger BVH traversal with the ray
20   castRay(bvh, ray, count=0);
21 }
22
23 IS_Shader() {
24   //called every time a ray intersects a leaf AABB
25   ray_origin ← get the ray origin;
26   curPoint ← get the AABB center;
27
28   if (distance(ray_origin, curPoint) < radius^2) {
29     record curPoint as a neighbor;
30     if ((count + 1) == K) calls the AH shader;
31     else count++;
32   }
33 }
34
35 AH_Shader() {
36   terminate the current ray;
37 }
```

Lines 3-6 show that we generate an AABB for each point and build the BVH. All the AABBs have the same width (twice the search radius). Lines 17-18 show that each query is mapped to a ray. When the maximum neighbor count K is met, the IS shader calls the AH shader to terminate the ray.

3.2 Understanding the Performance

We characterize key aspects that impact the performance of our algorithm. The performance characterizations point out sources of potential inefficiency and motivate optimizations that we propose in sections that follow. The performance results are obtained from a RTX 2080 Ti GPU. Input data used in this section are from the popular KITTI dataset [11, 12]. See Section 6.1 for a complete experimental setup.

3.2.1 Ray Coherence Tree traversal is control-flow intensive. Rays that are spatially distant ("incoherent" rays in graphics parlance [1, 31]) will diverge when traversing the BVH. For instance, Ray A and Ray B in Figure 1b exercise different traversal paths and execute different shaders.

OptiX groups every 32 adjacent rays generated in the RG shader into a warp. This means adjacent rays, if representing spatially-distant queries, will lead to control-flow divergence. Even worse, for tree traversal-based algorithms, control-flow divergences translate to lower memory access efficiency. This is because incoherent rays access different tree nodes as they exercise different traversal paths, increasing the working set size and reducing chances for memory coalescing.

To demonstrate the impact of incoherent rays on neighbor search, we perform a simple experiment, where we assign queries uniformly to the cells in a 3D grid and compare two different query-to-ray mappings: 1) queries are mapped to rays according to the raster-scan order of the grid cells such that adjacent rays represent spatially-close queries, and 2) queries are randomly mapped to rays.

Figure 5 shows the results. To draw general conclusions, x-axis varies the number of queries from 0.27 to 27 millions. Searching with arbitrarily-ordered rays is consistently 5 times slower compared to searching with coherent rays. The performance difference is corroborated by the micro-architectural behaviors. Figure 6 shows that the search with ordered queries/rays has significantly higher L1/L2 cache hit rate and SM occupancy compared to the incoherent search.

It is worth noting that in the current OptiX implementation, a ray could, at run time (and out of a programmer's hands), be moved to a different thread, warp, or an SM to improve the ray coherency [27]. Our results show that *even with* the run-time coherence optimization by OptiX, performance remains sensitive to the initial query to ray mapping.

Observation 1: *Search performance is sensitive to ray coherence, which could be improved by mapping queries to rays such that adjacent rays represent spatially-close queries.*

3.2.2 AABB Size. The search time strongly correlates with the AABB size in the BVH. To demonstrate the impact of the AABB size, we fix the amount of queries and vary the AABB width in the BVH from 0.3 to 30. Figure 7 shows that the search time increases as the AABB width increases.

The reason that the search time correlates with the AABB size is that a larger AABB size means a query is enclosed by more AABBs; thus, the corresponding ray intersects more AABBs. More ray-AABB intersections translate to more BVH traversals on the RT cores (step 1 in the algorithm), which in turn leads to more IS calls on the SMs (step 2 in the algorithm). Figure 8 confirms that the number of IS shader calls increases with the AABB width[1]. Interestingly, the number of IS shader calls grows super-linearly. This is because the AABB volume grows *cubicly* w.r.t. its width, so the number of AABBs that a query resides in (i.e., ray intersections) grows cubicly too.

Observation 2: *Search time is strongly correlated with the AABB size, which dictates the work on both the SMs and the RT cores. Reducing AABB size reduces the search time.*

4 Spatially-Ordered Query Scheduling

Each query is mapped to a ray; a direct mapping, shown at the top of Figure 9, maps queries to rays in the order that the queries appear in the input, which could be arbitrary, leading to incoherent rays. This section introduces a query scheduling technique that tames the ray incoherence and reduces the control-flow divergence in the algorithm.

Our intuition is to group spatially close queries such that adjacent rays follow similar BVH traversal paths. We propose a lightweight grouping algorithm using a simple heuristic: *queries that reside in the same leaf AABB are spatially close* and should be grouped together. In practice, a query is usually enclosed by many leaf AABBs; any such enclosing AABB would provide a useful hint for the query's spatial proximity. That is, we are not interested in a particular enclosing AABB for a query as long as we associate an AABB with each query.

This loose definition of spatial proximity allows us to group queries very efficiently — as another ray tracing problem! In particular, finding an enclosing AABB for a query can be done by casting a ray for the query and immediately terminating the ray once the *first* IS shader is called, essentially returning the first intersecting leaf AABB of each query. This ray tracing is very efficient because it invokes the IS shader only once for each ray without traversing the entire BVH.

Listing 2. Pseudo-code of neighbor search with ray reordering, which is done through an initial ray tracing that terminates when the first leaf AABB is found for each ray.

```
1 bvh←buildBVH(points, radius);
2 // initial search with K = 1
3 FirstHitAABBs←traceRays(queries, 1, radius, bvh);
4 reorderQueries(queries, FirstHitAABBs);
5 // second search with the actual K
6 traceRays(queries, K, radius, bvh);
```

[1]Statistics about the number of traversals are hidden by OptiX. The fact that Figure 7 and Figure 8 have the same trend means that the time per IS execution is roughly constant, which we experimentally confirm.

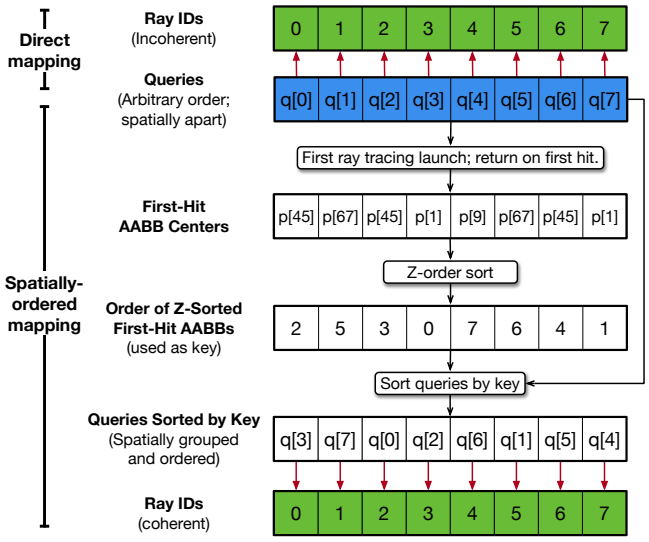


Fig. 9. Comparison between direct query-to-ray mapping (top) and spatially-order mapping (bottom). **p[]** denotes the array of points, which are also the AABB centers; **q[]** denotes the array of queries, which are also the ray origins.

Listing 2 shows the pseudo-code of the search algorithm with query grouping. Essentially we perform ray tracing twice: the first time is with a maximum neighbor count $K = 1$ (Line 3) and the second time is with the actual search K (Line 6). After the first search, all the queries have a first-hit AABB ID; queries with the same ID are then grouped together.

One issue remains: how should different groups be ordered? Recall that each leaf AABB represents a point in the search space. Thus, the order of the first-hit AABBs is essentially the order in which their corresponding search points appear in the input, which could be arbitrary.

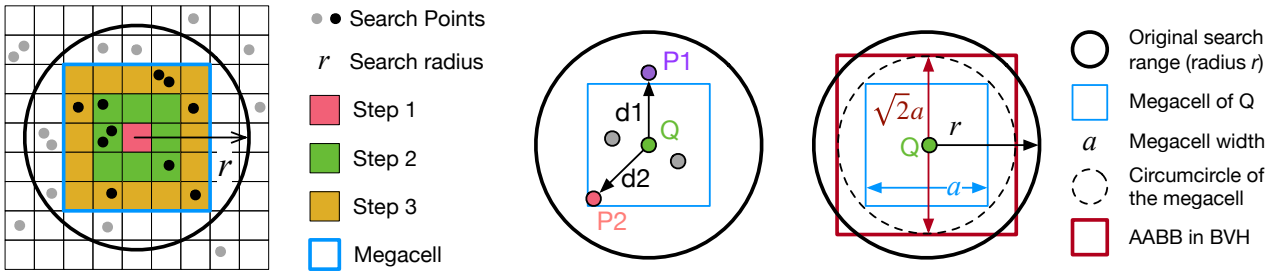
To introduce order into the first-hit AABBs, we simply sort their corresponding points (i.e., AABB centers) in a Morton (Z) order. This is done by the function `reorderQueries()` in Listing 2, and is implemented in a CUDA kernel, which operates on the first-hit AABB data produced by the shaders directly in the device memory without extra memory copies. Figure 9 compares the the spatially-ordered query-to-ray mapping (bottom) with the direct mapping (top).

5 Query Partitioning and Bundling

This section introduces a technique to suppress BVH traversals. The idea is to partition queries and build a specialized BVH for each partition with the smallest possible AABB size without violating correctness. We first describe the basic idea (Section 5.1), followed by an algorithm to determine the (near-)optimal partitioning (Section 5.2).

5.1 Query Partitioning

In our baseline algorithm, all the queries share the same BVH. We observe, in Section 3.2.2, that the search time is strongly correlated with the AABB size in the BVH, because larger



(a) Calculating the megacell of a query by incrementally growing from the cell that contains the query. The growth stops when either the sphere boundary is reached or at least K neighbors are found.

(b) $P1$ is outside of Q's megacell that contains 3 points, but is among the 3 nearest neighbors of Q (as $d1 < d2$).

(c) For KNN search, the AABB must circumscribe the sphere that circumscribes the megacell. The AABB width is $\sqrt{2}a$ for 2D search (illustrated here) and $\sqrt{3}a$ for 3D search (as the AABB is 3D and the circumcircle becomes the circumsphere), where a is the megacell width.

Fig. 10. Determining the megacell and AABB size used for neighbor search.

AABBs result in more traversals and IS shader calls, which increases the search time.

Our idea is to, for each query, identify an AABB size that is *just large enough* to ensure correctness. In this way, instead of using one monolithic BVH for all queries, queries are partitioned into different partitions, each with a unique BVH that minimizes the amount of search work for that partition.

In theory, the AABB width must be twice as the search radius r provided by users (see Figure 4c). However, it is possible to use smaller AABBs for a query if its K neighbors can be found within a smaller radius. We use an iterative method over a uniform grid to identify the proper AABB size for a query. This is illustrated in Figure 10a.

We first create a uniform grid over the scene containing all the search points. For each query, we then calculate the least amount of grid cells that contain K neighbors. This calculation is done by starting from the cell that contains the query, and iteratively growing the cells along all six directions (or four in the case of 2D search). The growth stops just before the r-radius sphere boundary is reached or at least K neighbors are found. We call the final collection of cells a *megacell*. Naturally, the largest possible megacell is the cube (square) that is inscribed by the sphere (circle).

Determining AABB Size Given the megacell of a query, the next step is to decide the AABB size used to build the BVH, which differs between range search and KNN search.

In range search, the AABB size can be safely set to the megacell size. The actual search would simply return the K neighbors from the megacell, which are guaranteed to be within the distance r from the query Q. An additional benefit now is that the IS shader does not have to perform the sphere test anymore (Step 2 in Section 3.1), since any query that is enclosed by the AABB is guaranteed to be enclosed by the sphere. This leads to significant performance gains.

The situation is slightly more complicated for KNN search. Even if a megacell contains K neighbors of Q, it does not mean the *nearest* K neighbors of Q are in the megacell. Figure 10b shows a counter-example where $P1$, which is just outside of the megacell, is part of the 3 nearest neighbors of Q even though the megacell contains 3 neighbors.

We *can*, however, guarantee that the circumscribed sphere of the megacell will contain the K nearest neighbors of Q. This is illustrated in Figure 10c. To guarantee correctness, a conservative estimation would be to set the AABB width $w = \sqrt{2}a$ for 2D search (as illustrated in Figure 10c) and $w = \sqrt{3}a$ for 3D search, where a is the megacell width.

We use a simple heuristic to reduce w: assuming that the point density is locally uniform within and around a megacell, a sphere with the same volume as the megacell should contain K neighbors of Q. Thus, we use a $w = 2\sqrt[3]{3/(4\pi)}a$ (in 3D search)[2]. We find this heuristics to be sufficient (for correctness) from the datasets we evaluate (Section 6.1). One could further relax w if an application is amenable to approximate neighbor search, which we discuss in Section 8.

Algorithm Summary Listing 3 shows the pseudo-code of the search algorithm with query partitioning. Lines 1-5 calculate the megacell size for each query. In the end, the queries are naturally split into different partitions, each with a unique AABB size. Lines 7-10 generate a BVH for each partition; each partition is then searched separately using the corresponding BVH. Queries in each partition could be further spatially-ordered as described in Section 4.

Listing 3. Pseudo-code of neighbor search with query partitioning. Each partition has a different BVH.

```
1 grid ← generate grid from search points;
2 foreach query in queries
3   megaCellWidth ← gen megacell for query in grid;
4   AABBSize ← megaCellWidth√2; // √3 for 3D search
5   partitions[AABBSize].add(query);
6
```

[2]The megacell volume is a^3, and the sphere volume is $\frac{4}{3}\pi(\frac{w}{2})^3$, where a is the megacell width and w is the sphere diameter (i.e., AABB size) to be solved for. Thus, w is $2\sqrt[3]{3/(4\pi)}a$ to ensure equi-volume.

```
foreach p in partitions
  bvh ← buildBVH(points, p.AABBSize/2);
  queries = partitions[p];
  traceRays(queries, K, radius, bvh);
```

We implement the megacell calculation in CUDA. An important parameter is the grid resolution. Intuitively, small grid cells lead to a more accurate megacell estimation because the stride of each growth step is smaller, but also increase the memory consumption. In our implementation, we use the smallest cell size allowed by the GPU memory capacity.

5.2 Bundling the Partitions

By default, each partition is launched separately with its corresponding BVH (as shown in Listing 3). However, this strategy might be sub-optimal. This is because each partition requires constructing a unique BVH. In cases where the search time saving is smaller than the BVH construction overhead (e.g., when the partitions are small), generating many partitions degrades performance.

We propose an algorithm that optimally bundles partitions together to minimize the overall search time. The idea is to first generate as many partitions as described before, and then decide how to combine partitions together by analytically modeling the cost of bundling partitions. The overall ideas for KNN search and range search are the same, but differ in details. We focus on KNN search here, and leave the details of range search to Supplementary Material A.

Cost Model As a first-order approximation, the total search cost T is the sum of the cost of each of the P partitions, which in turn is the sum of the BVH construction cost (T_{build}) and the actual search cost (T_{search}):

$$T = \sum_{i=0}^{P} (T^i_{build} + T^i_{search}). \tag{2}$$

While Nvidia discloses little detail about their BVH construction algorithm and implementation, we empirically find that the BVH construction time is linearly correlated with the number of AABBs in the BVH (see details in Supplementary Material B). We model the BVH construction time as linearly scaling with the number of AABBs M:

$$T_{build} = k_1 M. \tag{3}$$

The search cost for a partition is dictated by the number of IS shader calls, which is a product of the number of queries (N) and the number of IS calls per query. The number of IS calls a query makes is equivalent to the number of leaf AABBs that the query resides in, which in turn is the product of the AABB volume and the point density (i.e., average number of points per unit volume). Therefore, the search time T_{search} is modeled as:

$$T_{search} = k_2 N \rho S^3, \tag{4}$$

where S is the AABB width and ρ is the point density. Since each partition's megacell, by construction, contains just about K points, ρ can be estimated by K/C^3, where C is the megacell width of the partition.

While there are two unknown coefficients k_1 and k_2 in our modeling, knowing their ratio is sufficient to compare the *relative* costs of partitioning strategies. This ratio can be obtained offline through profiling the BVH construction time per AABB and the IS shader execution time per call. On RTX 2080, this ratio is about 1:15000. Absent the offline profiling, we fall back to the default strategy (Listing 3).

Bundling Algorithm Given the cost modeling, let us explain why there exists an optimal bundling. The crux is that bundling partitions increases the total search cost but reduces the BVH construction cost.

Specifically, when we bundle a partition P_i (with N_i queries, an AABB width of S_i, a point density ρ_i) with another partition P_j (with N_j queries, an AABB width of S_j, a point density ρ_j), we eliminate one unit of BVH construction cost; meanwhile, the combined partition will have an AABB width of $max(S_i, S_j)$. Thus, the search cost of this new partition is greater than the individual search cost of P_i and P_j combined (assuming the point density does not change abruptly):

$$k_2(N_i\rho_i + N_j\rho_j)[max(S_i, S_j)]^3 > k_2(N_i\rho_i S_i^3 + N_j\rho_j S_j^3). \tag{5}$$

The goal of our bundling algorithm is to determine how to optimally bundle the available partitions to minimize the total cost. In theory, this is a combinatorial optimization, which in general is intractable. Fortunately, we empirically observe that this problem has a special structure that lends itself to be solved efficiently at run time. We leave the proof to Supplementary Material C, and state the conclusion below.

To find the optimal bundling, we first sort all partitions in the ascending order of their query counts; we then start from the last partition and linearly scan toward the first partition. At each stop, we bundle all the partitions that have been scanned, leave the rest unbundled, and calculate the total cost. The bundling strategy that has the lowest cost T wins.

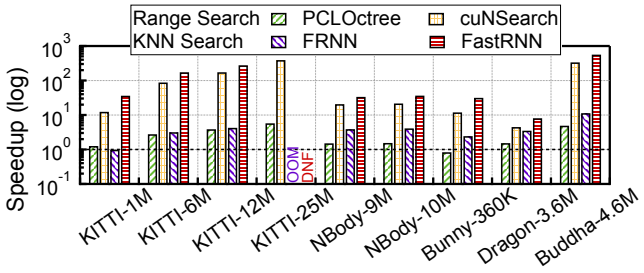
6 Evaluation

After describing our evaluation methodology (Section 6.1), we show the overall speedup of RTNN (Section 6.2), followed by teasing apart the contributions of different optimizations (Section 6.3). Finally, we study the sensitivity of RTNN with respect to search configurations (Section 6.4).

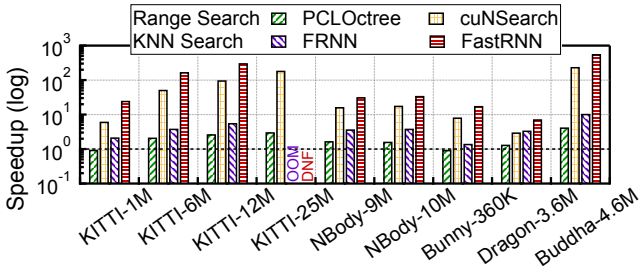
6.1 Evaluation Methodology

Environment We implement our algorithm in OptiX 7.1; the entire program is compiled with nvcc V11.4.48. We evaluate the performance on two Turing GPUs: a RTX 2080Ti (68 RT cores, 4352 CUDA cores, 11 GB GDDR6) and a RTX 2080 (46 RT cores, 2944 CUDA cores, 8 GB GDDR6).

Baselines We compare with four GPU baselines, which are built and evaluated in the same environment as RTNN.



(a) Speedups on RTX 2080.



(b) Speedups on RTX 2080Ti.

Fig. 11. Speedup of RTNN over the baselines (log-scale). OOM denotes that the baseline ran out of memory; DNF denotes that the baseline did not finish within the time that would have given RTNN a 1,000 speedup.

- **cuNSearch** [15, 16] is an optimized CUDA library used in many scientific computing applications such as the widely popular SPlisHSPlasH fluid simulator [4]. cuNSearch has only a range search implementation.
- **FRNN** [45] is a drop-in replacement for (and about 10 times faster than) the KNN search in PyTorch [10]. We instrument the code to measure just the CUDA time without the Python wrapper overhead.
- **PCLOctree** is an octree-based CUDA implementation in Point Cloud Library (PCL) [33, 34], a widely-used library for computational geometry, graphics, and vision. PCLOctree is available for both KNN search and range search (K must be 1 for KNN search).
- **FastRNN** [9] is a recent work that leverages the RT cores for KNN search only and *without* the various optimizations that we propose in this paper.

Why These Baselines? Both cuNSearch and FRNN are grid-based algorithms. The performance gains of RTNN over them highlight the benefits of using a tree structure (i.e., BVH) with hardware acceleration.

Similar to RTNN, PCLOctree also uses a hierarchical data structure, i.e., the Octree, which is a space-partitioning structure rather than an object-partitioning structure like the BVH. Comparing with PCLOctree shows the benefits of hardware-supported object partitioning. Comparing to FastRNN quantifies the optimizations proposed in this paper.

Datasets We use three datasets covering three domains where neighbor search is critical. We first use the LiDAR-generated point clouds from the KITTI self-driving car dataset, which is commonly used in computer vision and robotics research [11, 12]. To evaluate the scalability, we combine point cloud frames to obtain three final frames with a point count of 1M, 12M, and 25M, respectively.

The second dataset consists of three 3D-scanned models from the Stanford 3D Scanning Repository [20]: Bunny (360K points), Asian Dragon (3.6M points), and Buddha [7] (4.6M points), all of which are widely used for graphics research. Finally, we use a cosmological N-body simulation dataset [24, 38] from the Millennium Simulation Project [23]. The dataset has two traces with 9M and 10M particles (galaxies) each.

Apart from covering three representative application domains where neighbor search is critical, these three datasets also allow us to evaluate RTNN under different point distributions. Points in the KITTI self-driving car dataset are mostly distributed in the xy-plane (the ground) and while being confined in a very narrow z-range (height). Points in the other two datasets occupy the entire 3D space, but point distribution in the cosmological simulation is much more *non-uniformly* than that in the 3D scanning dataset. Points in cosmological simulation represent galaxies in the universe; galaxy distribution is naturally non-uniform[3].

6.2 Overall Performance Analysis

On RTX 2080, RTNN provides a (geomean) 2.2× and 44.0× speedup over PCLOctree and cuNSearch, respectively, on range search, and provides a (geomean) 3.5× and 65.0× speedup over FRNN and FastRNN, respectively, on KNN search. Figure 11a shows the per-input speedup.

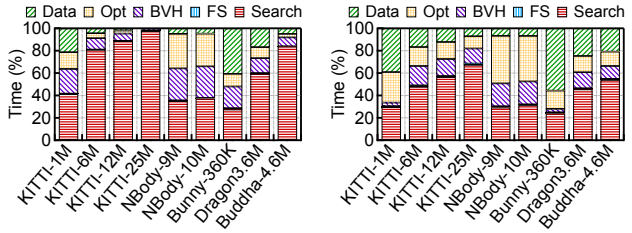
We observe that: 1) the speedup increases when the number of points increases, and 2) the speedup on range search is generally lower than that on KNN search. Let us elaborate.

Across Input Scales On the two smallest inputs (KITTI-1M and Bunny-360K), RTNN has only limited speedup (up to 2×) over the fastest baselines PCLOctree and FRNN. The speedups on larger inputs (e.g., KITTI-12M and Buddha-4.6M) are at least 5–10×. Figure 11b shows the results on RTX 2080Ti; the same trend holds.

To understand why the speedup changes with the scale of the input, Figure 12a breaks down the KNN search time on RTX 2080 into five components: data transfer time (`Data`)[4], the overhead of applying optimizations (`Opt`), including reordering and partitioning queries, time spent on generating the BVHs (`BVH`), the first search to find the first-hit AABBs

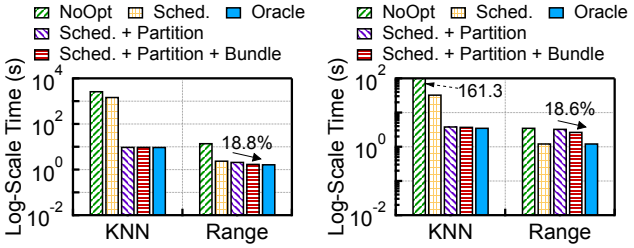
[3]On scales of order 1 to 10 Mpc/h, the galaxy distribution is roughly hierarchical clustering (fractal), where 1 Mpc/h is of order the spacing between galaxies. On scales much larger than 10 Mpc/h the matter distribution very slowly approaches uniformity. The Millennium Simulation dataset runs 500 Mpc/h on a side and, thus, exhibits the non-uniform distribution.

[4]This includes times to copy data to and from the device memory. The former is not hidden, but the latter is almost completely hidden.



(a) KNN search time distribution. **(b)** Range search time distribution.

Fig. 12. Time distribution on RTX 2080.



(a) KITTI-12M. **(b)** NBody-9M.

Fig. 13. Effects of our optimizations on RTX 2080.

for queries (FS), and the second (actual) search (Search). The execution times of smaller inputs (e.g., KITTI-1M and Bunny-360K) are dominated by non-search related tasks, diminishing the gains from accelerating the search.

The two N-body inputs are interesting cases: they have large numbers of points but still spend more than half of the time on non-search related tasks. A close examination shows that the spatial density of their points (i.e., galaxies used in the N-body simulation) varies a lot. Thus, queries have different megacell sizes and fall into different partitions. As a result, RTNN spends much time generating the partitions (Opt) and building the different BVHs (BVH).

KNN vs. Range Search We also observe that the speedup on KNN search is generally higher than that on range search. This is because KNN search spends more than in the actual search than range search due to the need to manipulate a priority queue. This time distribution difference is evident by comparing Figure 12a and Figure 12b, which show the time distribution for KNN search and range search, respectively. For instance, on KITTI-12M RTNN spends 88.5% of the time on the actual search (Search) under KNN search, which decreases to only 63.5% under range search.

6.3 Teasing Apart Optimizations

The optimizations we propose in this paper are critical to the performance benefits of RTNN. Using two representative inputs, Figure 13 compares the performance of five variants of our algorithm on RTX 2080 (KITTI-12M in Figure 13a and NBody-9M in Figure 13b):

- NoOpt: no optimization;
- Sched.: query scheduling only (Section 4);
- Sched. + Partition: query scheduling and query partitioning (Section 5.1);
- Sched. + Partition + Bundle: query scheduling, partitioning, and bundling (Section 5.2);
- Oracle: assuming a priori knowledge of 1) whether to partition, and 2) the best bundling strategy through an offline exhaustive search (infeasible for run time).

Scheduling Comparing to NoOpt, ray scheduling improves the performance by 1.8× and 5.9× on KNN and range search, respectively, for KITTI-12M; the speedups are 4.7× and 2.9× for NBody-9M. The speedups not only come from better ray coherence but also because the overhead of finding the first-hit AABBs is negligible, which is evident in Figure 12, where the FS category is virtually invisible.

Partitioning Query partitioning is much more effective for KNN search. On KITTI-12M, partitioning provides a 154.4× and 1.1× speedup for KNN and range search, respectively, on top of ray scheduling.

Recall that query partitioning improves speed by suppressing tree traversals and IS calls. KNN search needs many more traversals (as it must find the K *nearest* neighbors whereas range search terminates tree traversal whenever K neighbors are found), and the cost of an IS shader call in KNN search is (3–6×) higher than that in range search. Thus, query partitioning is more effective to KNN search.

Interestingly, query partitioning degrades performance on NBody-9M. As discussed early, the points in N-body simulations are non-uniformly distributed, which results in a high partitioning and BVH construction overhead (Figure 12).

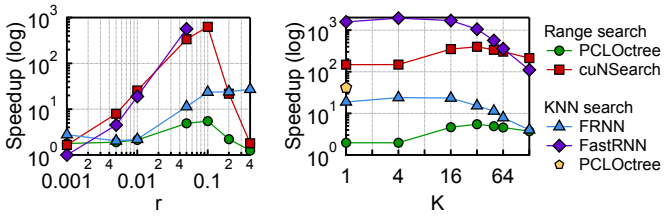
Bundling Bundling provides an additional 18.8% and 18.6% performance gain on range search for the two inputs, but has little impact on KNN search. This is because KNN search, with its hefty costs of IS shader and traversal, typically uses all the partitions anyways (i.e., no bundling), which our bundling algorithm accurately captures.

Overall, our bundling algorithm is effective, leading to a performance that is within 3% of Oracle for KITTI-12M. The Oracle for NBody-9M is achieved when partitioning is disabled, whereas RTNN always assumes partitioning. A future improvement to RTNN is to estimate the points' spatial density before deciding whether to partition.

6.4 Sensitivity Analysis

RTNN offers speedups across a range of r and K values. Using range search on Buddha-4.6M as an example, Figure 14a and Figure 14b show how the speedup on RTX 2080 varies with r and K, respectively.

Sensitivity to r As r increases our speedups increase initially, because a larger r means a larger AABB and, thus, more search work that can be accelerated. For range search, however, the speedups (over PCLOctree and cuNSearch) decrease (still >1) as the search radius r exceeds 0.1. This is because the points in Buddha are bounded in a 1^3 cube;



(a) Sensitivity of speedup to r. **(b)** Sensitivity of speedup to K.

Fig. 14. Range search speedup of Buddha-4.6M on RTX 2080 as r and K change. PCLOctree supports only K=1 for KNN search, so it has only one KNN data point. The missing data points of FastRNN is because it did not finish within the time that would have given RTNN a 1,000 speedup.

the search sphere under, say, $r = 0.4$, covers almost the entire cube. Queries can thus quickly find neighbors under a large r, so the search terminates quickly, in which case the overhead of RTNN (e.g., building BVH, ray scheduling) are more pronounced, leading to lower speedups.

Sensitivity to K As K grows, RTNN's speedup generally increases, because a larger K leads to more search work that can be accelerated by RTNN. The speedup degrades when K becomes too big (e.g., 128). We find that this is because the bundling algorithm tends to be overly aggressive under a larger K. We leave it to future work to investigate a better bundling algorithm under large Ks.

7 Related Work

Section 2.1 provides an overview of neighbor search. Section 6.1 discusses neighbor search algorithms in low-dimensional space. This section focuses on work related to ray tracing.

RTX Beyond Ray Tracing Recent papers have started using (Nvidia) ray tracing hardware to accelerate workloads beyond ray tracing, including both rendering workloads [22, 46, 47] and non-rendering workloads [9, 35, 40].

Among them, Evangelou et al. [9] and Zellmann et al. [47] are the closest to our paper; the former uses RT cores for 3D KNN search and the latter uses RT cores for 2D range search. This paper provides a unified neighbor search algorithm with two generally-applicable optimization that significantly improves the search performance. In addition, we provide a detailed performance characterization of using ray tracing for neighbor search (Section 3.2).

Ray Incoherence Literature is rich with techniques that improve ray tracing performance, much of which is focused on taming incoherent rays [1, 2, 8, 14, 31, 37], which lead to both branch divergences and memory inefficiencies.

All existing techniques target rendering. We propose an efficient reordering technique specialized to neighbor search, which exposes two opportunities. First, all the rays (queries) are known at the beginning, enabling a global reordering; in contrast, incoherent rays in rendering are dynamically generated (e.g., bounces) and, thus, are usually locally reordered. In this sense, our ray scheduling is similar to wavefront ray tracing [21, 37] but with only one wavefront. Second, our rays have the same direction, so reordering involves only the ray origin (3D) whereas reordering for rendering usually involves both ray origin and direction (6D).

8 Conclusion and Future Work

Conventional GPUs, while initially built for rasterization-based graphics, are now widely used as general-purpose accelerators for parallel algorithms. Emerging ray tracing-based GPUs beg the question: can we use the ray tracing hardware for workloads beyond ray tracing? This paper uses neighbor search as a case study and provides a positive answer. We show that effectively exploiting the ray tracing hardware requires carefully mapping work items (i.e., queries in our case) to rays and suppressing excessive tree traversals. The analyses and techniques in this paper provide useful insights in broadening the utility of ray tracing hardware.

Approximate Neighbor Search Many applications do not require exact neighbor search. RTNN is amenable to approximation, sometimes with quantitative error bounds. We discuss two opportunities here. First, in building a BVH for a query (or a query partition) one could use an AABB size smaller than what is strictly required. Using a smaller AABB would reduce the number of neighbors returned but also provide performance gains, since performance is sensitive to AABB size as established in Section 3.2.2.

Second, one could elide Step 2 in the search algorithm (Section 3.1), i.e., treating any query that resides in an AABB as residing in the inscribed sphere. Under this approximation, given a query range r all the returned neighbors are bound to be within a distance $\sqrt{3}r$ of the query. Speedups from this approximation would be significant, given that Step 2 is much more costly than Step 1.

General-Purpose Irregular Processor Given the success of today's GPGPUs for regular algorithms, one would naturally wonder how a ray tracing-based GPU can be used as a general-purpose processor for *irregular* applications. Ray casting is fundamentally a tree traversal problem, which is central to many irregular applications beyond graphics.

Realizing this vision requires us to carefully rethink the architecture, run-time system, and programming model [49]. Hardware-wise, Nvidia's RT cores are specialized for BVH traversal with a specific branching logic (bounded intersection checking), which is not easily extensible to more generic traversals. Meanwhile, the run-time ray scheduler, a performance-critical component, should ideally be customized to a particular algorithm. Our paper shows that the default OptiX scheduler optimized for ray tracing is sub-optimal for neighbor search. Finally, it would be interesting

to explore programming models that free programmers from constantly thinking about rays and geometry.

9 Acknowledgements

The author expresses gratitude to Prof. Kelly Douglass and Prof. Segev BenZvi, both of Department of Physics and Astronomy at University of Rochester, for pointing to the N-body simulation dataset, and to Prof. Michael Vogeley of Department of Physics at Drexel University for explaining the point density distribution in N-body simulation. The author also thanks Tiancheng Xu and Boyuan Tian for providing help in setting up PCL. The work was supported, in part, by NSF under grants #2044963 and #2126642.

Appendix

A Cost Model for Range Search

The BVH construction cost T_{build} of range search is the same as that of KNN search. The search cost T_{search} is different from that in KNN search. In particular, the number of IS shader calls per query in range search is always K, as the search terminates as soon as K neighbors are found. That is:

$$T_{search} = k_3 NK, \tag{6}$$

where S is the AABB width, N is the number of queries in a partition. k_3 is the time of an IS shader in range search.

k_3 depends on the megacell size of a partition. When the megacell size does not touch the search sphere, the IS shader can skip the ray-sphere intersection test, because the corresponding query, being inside the megacell, is guaranteed to reside in the sphere. When the megacell size touches the sphere, the IS shader has to perform the ray-sphere intersection test, because a query residing in the megacell does not guarantee that the query will reside in the sphere. As a result, the ratio of k_1 to k_3 varies. On RTX 2080, the ratio is about 20:1 in the former case and is 2:1 in the latter case.

B Modeling BVH Construction Time

Figure 15 shows how the BVH time varies with the number of AABBs. We regress a linear fit for the correlation with an R^2 of 0.996, indicating a strong linear relationship.

C Derivation of the Optimal Bundling Algorithm

We find that as the AABB size of a partition increases the number of queries in the partition usually decreases. Figure 16 shows a typical query distribution over the AABB size when searching about 6 millions queries in total. This makes statistical sense, because usually only a handful of sparsely located queries need a large AABB, whereas most of queries should be captured by small AABBs.

With this observation, our bundling algorithm is divided into two steps. First, we prove the following theorem:

Theorem: *if* the optimal number of bundles is M_o ($1 \leq M_o \leq M$, where M is the number of available partitions), the optimal bundling strategy is one where the $(M_o - 1)$

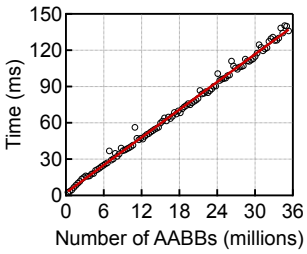


Fig. 15. BVH construction time is linearly correlated with the number of AABBs.

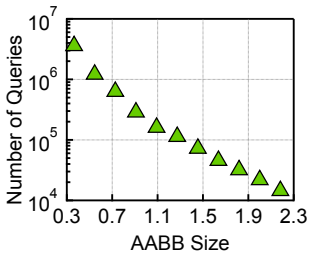


Fig. 16. Number of queries and the AABB size in a partition are inversely correlated.

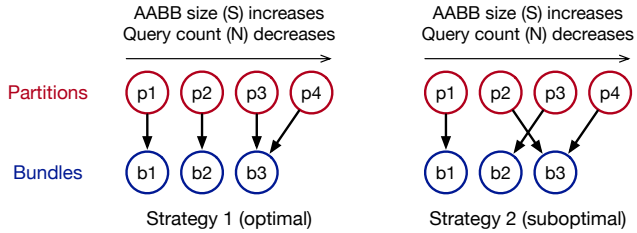


Fig. 17. Partition bundling. Under the empirical observation that the AABB size and the query count of the partitions are inversely correlated, bundling strategy 1 is optimal, whereas strategy 2 is sub-optimal.

partitions that have the most queries are *not* bundled and the remaining partitions are combined into one bundle.

Figure 17 illustrates it using a simple example, which has four partitions that are sorted according to the ascending order of their AABB sizes S, which is also the descending order of their query counts N (based on the empirical observation above). Assuming that the optimal number of bundles is three, the optimal bundling strategy is strategy 1 (left figure), where the first two partitions have their own bundles and the last two partitions are combined. Its search cost is (omitting the constant coefficient k_2):

$$T^{(1)}_{search} = N_1\rho_1 S_1^3 + N_2\rho_2 S_2^3 + (N_3\rho_3 + N_4\rho_4)S_4^3. \tag{7}$$

We can prove this by contradiction. Without losing generality, let us assume that strategy 2 in Figure 17, where $p2$ and $p4$ are combined, is optimal. The resultant search cost is thus (again omitting the coefficient k_2):

$$T^{(2)}_{search} = N_1\rho_1 S_1^3 + N_3\rho_3 S_3^3 + (N_2\rho_2 + N_4\rho_4)S_4^3. \tag{8}$$

Since $S_2 < S_3 < S_4$, $S_2 = \sqrt{3}C_2$, $S_3 = \sqrt{3}C_3$ we have:

$$\rho_2 = K/C_2^3 > K/C_3^3 = \rho_3. \tag{9}$$

Combined with $N_2 > N_3$, we have:

$$T^{(1)}_{search} - T^{(2)}_{search} = N_3\rho_3(S_4^3 - S_3^3) - N_2\rho_2(S_4^3 - S_2^3) < 0. \tag{10}$$

Since both strategies require three BVH constructions (as there are three bundles), strategy 2 has a higher total cost ($T_{build} + T_{search}$) than that of strategy 1, contradicting the proposition that strategy 2 is optimal.

The theorem essentially allows us to find the optimal bundling in constant time *given* an M_o. The problem then is reduced to finding the optimal M_o, which is a linear-time problem: we linearly search all the possible M_o values; for each M_o value ($1 \leq M_o \leq M$), we estimate its cost according to the bundling strategy given by the theorem (constant time) and pick the M_o that has the lowest search cost.

References

[1] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*. 113–122.

[2] Rasmus Barringer and Tomas Akenine-Möller. 2014. Dynamic ray stream traversal. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–9.

[3] Jens Behley, Volker Steinhage, and Armin B Cremers. 2015. Efficient radius neighbor search in three-dimensional point clouds. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3625–3630.

[4] Jan Bender. [n.d.]. SPlisHSPlasH. https://github.com/InteractiveComputerGraphics/SPlisHSPlasH.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1999. When is "nearest neighbor" meaningful?. In *International conference on database theory*. Springer, 217–235.

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

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

[8] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, Vol. 32. Wiley Online Library, 125–132.

[9] I Evangelou, G Papaioannou, K Vardis, and AA Vasilakis. 2021. Fast Radius Search Exploiting Ray-Tracing Frameworks. *Journal of Computer Graphics Techniques Vol* 10, 1 (2021).

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

[11] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. 2021. KITTI Raw Data. http://www.cvlibs.net/datasets/kitti/raw_data.php.

[12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 3354–3361.

[13] Andrew S Glassner. 1989. *An introduction to ray tracing*. Morgan Kaufmann.

[14] Christiaan P Gribble and Karthik Ramani. 2008. Coherent ray tracing via stream filtering. In *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 59–66.

[15] Rama C. Hoetzlein. [n.d.]. cuNSearch. https://github.com/InteractiveComputerGraphics/cuNSearch. commit: b3b708de5a396826aecc74b47b390418f92c8dcb.

[16] Rama C Hoetzlein. 2014. Fast fixed-radius nearest neighbors: interactive million-particle fluids. In *GPU Technology Conference*, Vol. 18. 2.

[17] Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. 2011. A parallel SPH implementation on multi-core CPUs. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 99–112.

[18] James T Kajiya. 1986. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 143–150.

[19] Dan Koschier. [n.d.]. cuNSearch. https://github.com/InteractiveComputerGraphics/CompactNSearch. commit: b8c41fcefd6a8a7896cf3972dcb92aa407969ed7.

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

[21] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*. 137–143.

[22] Nate Morrical, Will Usher, Ingo Wald, and Valerio Pascucci. 2019. Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *2019 IEEE Visualization Conference (VIS)*. IEEE, 256–260.

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

[24] MPA. 2005. Millennium Run semi-analytic galaxy catalogue. https://wwwmpa.mpa-garching.mpg.de/galform/virgo/millennium/.

[25] Marius Muja and David G. Lowe. [n.d.]. FLANN - Fast Library for Approximate Nearest Neighbors. https://github.com/flann-lib/flann. commit: 1d04523268c388dabf1c0865d69e1b638c8c7d9d.

[26] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)* 2, 331-340 (2009), 2.

[27] Nvidia. 2021. NVIDIA OptiX 7.4 – Programming Guide; Overview. https://raytracing-docs.nvidia.com/optix7/guide/index.html#introduction#overview.

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

[29] PCL. [n.d.]. Spatial Partitioning and Search Operations with Octrees. https://pcl.readthedocs.io/projects/tutorials/en/latest/octree.html.

[30] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

[31] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 101–108.

[32] Reid Pinkham, Shuqing Zeng, and Zhengya Zhang. 2020. Quicknn: Memory and performance optimization of kd tree based nearest neighbor search for 3d point clouds. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 180–192.

[33] Radu Bogdan Rusu and Steve Cousins. [n.d.]. Point Cloud Library. https://pointclouds.org/. v1.11.0.

[34] Radu Bogdan Rusu and Steve Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*. Shanghai, China.

[35] Justin Salmon and Simon McIntosh-Smith. 2019. Exploiting Hardware-Accelerated Ray Tracing for Monte Carlo Particle Transport with OpenMC. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 19–29.

[36] Peter Shirley, Ingo Wald, Tomas Akenine-Möller, and Eric Haines. 2019. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs. Chapter 2: What is a Ray?* Apress, Berkeley, CA. 15–19 pages. https://doi.org/10.1007/978-1-4842-4427-2_2

[37] Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual streaming for hardware-accelerated ray tracing. In *Proceedings of High Performance Graphics*. 1–11.

[38] Volker Springel, Simon DM White, Adrian Jenkins, Carlos S Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, et al. 2005. Simulating the joint evolution of quasars, galaxies and their large-scale distribution. *arXiv preprint astro-ph/0504097* (2005).

[39] Elena Vasiou, Konstantin Shkurko, Ian Mallett, Erik Brunvand, and Cem Yuksel. 2018. A detailed study of ray tracing performance: render time and energy cost. *The Visual Computer* 34, 6 (2018), 875–885.

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

[41] Janett Walters-Williams and Yan Li. 2010. Comparative study of distance functions for nearest neighbors. In *Advanced techniques in computing sciences and software engineering*. Springer, 79–84.

[42] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, Vol. 98. 194–205.

[43] Turner Whitted. 1979. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 14.

[44] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. 2019. Tigris: Architecture and Algorithms for 3D Perception in Point Clouds. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 629–642.

[45] Lixin Xue. [n.d.]. Fixed Radius NN Search. https://github.com/lxxue/FRNN. commit: e8017ccb94ccef3afe0a621d5a6f677b45fda2a3.

[46] Stefan Zellmann, Martin Aumüller, Nathan Marshak, Ingo Wald, S Frey, J Huang, and F Sadlo. 2020. High-Quality Rendering of Glyphs Using Hardware-Accelerated Ray Tracing.. In *EGPGV@ Eurographics/EuroVis*. 69–73.

[47] Stefan Zellmann, Martin Weier, and Ingo Wald. 2020. Accelerating force-directed graph drawing with rt cores. In *2020 IEEE Visualization Conference (VIS)*. IEEE, 96–100.

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

[49] Yuhao Zhu. 2021. Graphics Rendering: What's New and What's in it for Architects? (Part I). https://www.sigarch.org/graphics-rendering-whats-new-and-whats-in-it-for-architects-part-i/.