Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration

Aaron Barnes, Fangjia Shen, Timothy G. Rogers
Elmore Family School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{barnes88, shen449, timrogers}@purdue.edu

***Abstract*—Specialized ray-tracing acceleration units have become a common feature in GPU hardware, enabling real-time ray-tracing of complex scenes for the first time. The ray-tracing unit accelerates the traversal of a hierarchical tree data structure called a bounding volume hierarchy to determine whether rays have intersected triangle primitives. Hierarchical search algorithms are a fundamental software pattern common in many important domains, such as recommendation systems and point cloud registration, but are difficult for GPUs to accelerate because they are characterized by extensive branching and recursion. The ray-tracing unit overcomes these limitations with specialized hardware to traverse hierarchical data structures efficiently, but is mired by a highly specialized graphics API, which is not readily adaptable to general-purpose computation. We present the Hierarchical Search Unit (HSU), a flexible datapath to accelerate a more general class of hierarchical search algorithms, of which ray-tracing is one. We synthesize a baseline ray-intersection datapath and maximize functional unit reuse while extending the ray-tracing unit to support additional computations and a more general set of instructions. We demonstrate that the unit can improve the performance of three hierarchical search data structures in approximate nearest neighbors search algorithms and a B-tree key-value store index. For a minimal extension to the existing unit, our HSU improves the state-of-the-art GPU approximate nearest neighbor implementation by an average of 24.8% using the GPU's general computing interface.**

***Index Terms*—GPU, Ray-Tracing, BVH, Nearest Neighbors**

I. Introduction

Ray-traced (RT) rendering generates photo-realistic images by casting millions of rays into a scene and generating pixel colors based on the objects they intersect with [23], [64]. While such techniques often generate computer graphics of a higher quality than traditional rasterization techniques, the large number of ray-triangle intersection computations required had prevented their adoption in applications with real-time rendering requirements until the introduction of dedicated RT hardware accelerators [44]. To reduce the number of intersection tests required for a given ray, scene primitives (commonly triangles) are organized in data structures such as Bounding Volume Hierarchies (BVH), which allow the search space to be culled by hierarchically eliminating candidate primitives. RT units accelerate the traversal of BVH trees, and the ray-triangle intersection tests computed upon reaching a leaf node.

There are two primary benefits the RT unit provides over traditional GPU SIMT hardware: it improves throughput in highly thread-divergent workloads (which is a common characteristic of RT workloads [8], [11]), and it provides fixed function datapaths to perform complete ray-box and ray-triangle intersection tests with a single instruction [1]. These RT accelerators implement CISC instructions that fetch data from memory and perform several calculations before returning the result. The ray-tracing problem requires searching a large set of primitives and testing for intersections with a given ray. In this work, we capitalize upon the observation that this search-and-compute software pattern is not limited to ray tracing applications. The dedicated hardware to accelerate ray tracing can be easily extended to benefit more general GPU search algorithms that may use different hierarchical data structures.

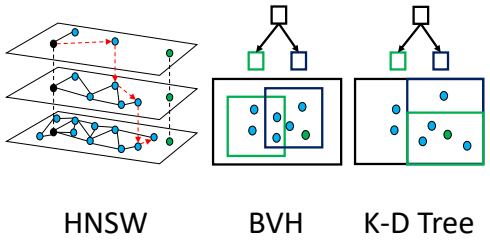


Fig. 1: Hierarchical Navigable Small World (HNSW) Graphs, Bounding Volume Hierarchy, and K-D tree hierarchical data structures are commonly used to accelerate search on GPUs and have similar branching characteristics which make them difficult to execute on SIMT execution units. Current GPU ray tracing units only accelerate BVH traversal, which has limited effectiveness at higher dimensions and with non ray-tracing workloads.

Presently RT unit resources on Nvidia and AMD GPUs are only accessible via graphics library APIs such as Optix, Vulkan, or DirectX. This reduces RT unit accessibility for scientific application developers who may be familiar with CUDA but not graphics pipelines. However, several works have successfully adapted the use of RT units to accelerate non-graphics workloads [6], [15], [21], [38], [40], [63], [71], [74]. While this requires significant programming effort, adapting the problem to execute on the GPU RT unit can unlock substantial performance gains. For example, Zhu [74] reformulates the nearest neighbors problem to use RT units

for acceleration and achieves speedups over several GPU nearest neighbor implementations. However, the design does not support datasets with more than three dimensions, and non-ideal reformulations are made, such as representing a point as a ray with a fixed direction and small extent. Further, the RT unit graphics APIs provide little visibility and control over the creation and layout of the bounding volume hierarchy to the programmer. Optimal partitioning of data while creating the BVH can vary for different datasets and applications and significantly impacts performance during traversal. The opaque hierarchical data structures built by RT library APIs may be optimal for graphics applications but sub-optimal for other application domains.

The current state of ray tracing unit programming is similar to the early stages of general-purpose GPU programming before the creation of programming models such as CUDA, ROCm, or OpenCL. In those days, GPU programmers were forced to reformulate their problems to fit a graphics rendering model which restricted their capability and ease of use. For example, Larsen and McAllister [31] represented matrix elements as colors and computed a matrix product by rendering an image of a cube and reading the results from screen memory. Thompson, Hahn, and Oskin [58] developed an early C++ programming framework which made it easier to accelerate vector computations using graphics hardware. However, they enumerated several limitations of the hardware, which was designed without general-purpose computing in mind, such as a lack of operating system management of GPU memory and the inability to preserve state across vertex shader invocations. Similar to early GPU programming, applications using the ray tracing unit for acceleration must adhere to a rigid programming structure with a fixed set of graphics shaders and data structures (see section III-A). Applications that successfully adapt their algorithm to fit within these constraints are still limited by restrictive hardware that was designed only with ray-tracing acceleration in mind. Current GPU RT Units only support three-dimensional data and the hierarchical data structure is restricted to an opaque BVH whose type and format are fixed. Figure 1 shows three common hierarchical data structures used in GPU nearest neighbor applications: hierarchical graphs, BVH, and k-d trees [41]. Graph-based hierarchies are currently used by the state-of-the-art ANN implementations for high-dimension features [18], [22], [47], [72], but cannot easily be accelerated by existing GPU ray tracing units.

Given the limitations of existing GPU ray tracing units, we propose a new Hierarchical Search Unit (HSU) that can accelerate a broad class of hierarchical search data structures and is programmable via a small, simple set of GPU instructions. We examine the functional units required to implement a baseline RT unit and determine that angular/euclidean distance calculations and B-tree key comparisons can be performed in the RT unit with minimal additions to the hardware. We evaluate the performance of HSU on Approximate Nearest Neighbor (ANN) search, which is a critical kernel in several important domains, including recommendation systems, autonomous driving systems, point cloud registration, and motion estimation. Our unit can accelerate searches in high-dimension datasets by reducing and aggregating distance results across multiple HSU instructions. We demonstrate its effectiveness by accelerating three different GPU ANN algorithms using distinct hierarchical data structures: a hierarchical navigable small world graph [18], a k-d tree [41], and a bounding volume hierarchy [74]. Additionally, we show how the unit can be applied to a b-tree key-value store for general 1-dimension search applications [9]. In summary, we make the following contributions:

- We propose a new programming interface for building and traversing hierarchical search structures on GPGPUs
- We design HSU, a set of Ray Tracing Unit compute extensions capable of performing euclidean/angular distance calculations with arbitrarily high dimensions, and key comparisons common in 1-dimension search algorithms.
- We simulate the performance benefit of HSU across a broad range of datasets using four representative hierarchical search algorithms.
- We synthesize RTL designs for our baseline ray tracing datapath and enhanced HSU datapath to estimate their area and dynamic power cost.

II. Background

A. Ray tracing algorithms

The runtime of ray tracing computation is dominated by two key operations: tree traversal and ray-primitive intersection tests. To reduce the overhead of ray intersection computations, scene primitives are typically organized into hierarchical tree data structures known as Acceleration Structures (AS). These AS are used to prune the search space while determining which objects a ray intersects with. Optimal AS creation, traversal, and scene modification is a rich area of research with a large body of prior work [10], [17], [32], [50], [62], [65], [73]. Bounding volume hierarchy (BVH) has become the most common type of AS used on GPUs. Nvidia and AMD GPUs use BVH for their acceleration structure [1], [45] because of its bounded memory footprint and flexibility to adapt to changing scene geometry [25]. Programmable GPUs have a large memory bandwidth and massively parallel compute units, thus they are an obvious choice to target ray tracing applications. However, ray tracing on GPUs has suffered poor SIMT efficiency and hardware utilization due to the branching and recursion inherent to the algorithm [8], [11], [52].

B. Bounding Volume Hierarchies

A graphics scene can be composed of hundreds of millions of triangle primitives. BVH partitions primitives into subsets contained in bounding volumes, which are typically axis aligned and referred to as Axis Aligned Bounding Boxes (AABBs). Rather than test each primitive individually for ray intersection, a ray can be tested against the larger bounding volume. If the ray does not intersect with the AABB, it is guaranteed not to intersect with any of the smaller AABBs or

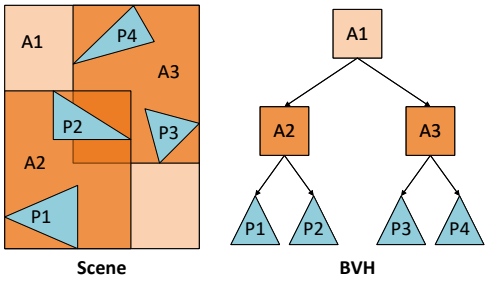


Fig. 2: Example scene with Bounding Volume Hierarchy

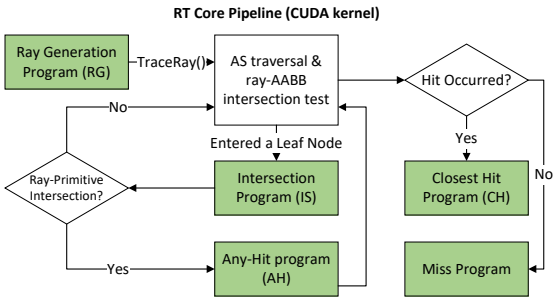


Fig. 3: Ray Tracing Unit Graphics Pipeline

primitives contained within. However if the ray does intersect with the volume, the search must continue down the tree.

Figure 2 shows an example scene with four primitives and 3 AABBs. BVH structures can be constructed top-down or bottom-up using heuristics such as scene surface area to determine bounding volume divisions [32], [62]. Ray search begins by testing if the ray intersects AABB A1. After a hit is confirmed search continues by testing AABB A2. If the ray does not intersect with A2, search can continue with A3 without testing intersection for primitives P1 and P2. In BVH structures volumes may overlap, however each primitive belongs to exactly one leaf node. For example A2 and A3 overlap in the scene, but P2 exists only in the A2 volume in the BVH. This fundamentally differs from spatial subdivision techniques like grids or k-d trees, which partition space into non-overlapping regions.

III. The Hierarchical Search Unit Programming Interface

A. Baseline Ray Tracing Unit Programming Interface

The baseline ray tracing unit is accessible only via graphics APIs such as DirectX, Vulkan, or Optix [45]. These APIs define a graphics pipeline that follows the fixed structure shown in Figure 3 but is adaptable via user defined shaders or Optix programs. The RT unit follows a single instruction multiple ray execution model wherein each ray is directly mapped to a corresponding CUDA thread. The user-defined Optix programs are shaded in green while operations performed by the RT unit hardware or graphics driver software are in white. These programs are essentially CUDA callback functions that are triggered by the RT unit during various stages of tree traversal. When a program is triggered, control is transferred from the RT unit or graphics driver software to a user-defined shader which executes before returning control back to the fixed function logic. The ray generation program (RG) serves as the entry point to the pipeline and typically is used to define the ray origin and direction. The intersection program (IS) is called each time a leaf node is encountered while traversing the AS. The intended purpose of the IS program is to allow programmers to define custom ray-primitive intersection tests. The IS program is optional; if triangles are used for primitives, the RT unit can execute ray-triangle intersection tests in hardware. The any-hit (AH) program is also optional, and is triggered any time an intersection is found. AH can be used to terminate ray traversal or reject intersections based on customized conditions. After ray traversal completes the closest-hit (CH) program is invoked if an intersection was found. If no hit occurred the miss program is called.

B. HSU Programming Interface

The HSU extends the baseline RT Unit ISA, and is therefore fully compatible with existing graphics ray tracing interfaces. In contrast to a standard baseline RT Unit, HSU also works in conjunction with a CUDA library that provides a set of configurable hierarchical data structures which are accessible from within device kernels. HSU natively supports N-dimensional data points as well as standard ray tracing primitives such as triangles and AABBs. This allows the HSU library to express a larger class of data in a more intuitive and memory-efficient manner than past approaches such as [21], [74] which resorted representing point data as a ray with a small extent or the center of a triangle. The library provides a set of optimized algorithms and data structures that make use of the HSU hardware, similar to existing GPU search libraries [22], [49]. The library provides support for creating and traversing the following structures:

- B-Trees for key-value stores [3], [21]
- Bounding Volume Hierarchies for ray tracing and 3-D applications
- K-D trees for N-Dimension point searches and Approximate Nearest Neighbors [41]
- Hierarchical Graphs for N-Dimension Approximate Nearest Neighbors and vector search [18], [47]

Additionally, the HSU's basic operations, such as ray-box intersection tests and N-dimension distance operations, are directly available to CUDA programmers for use in device code, which allows programmers the flexibility to create their own application-specific data structures. The function for performing Euclidean distance calculation, *__euclid_dist(a, b, N)*, has three input arguments and returns a single 32 bit float value. The first two arguments are pointers to an array of floats which represent two N-dimensional points. The return value of the function is the squared euclidean distance between the two points. The angular distance function, *__angular_dist(a, b, N)*, behaves in the same manner except the angular distance between the two points is returned. The compiler inserts the appropriate number of HSU instructions according to the dimension of the data points, as described in Section IV-F.

IV. The Hierarchical Search Unit Hardware Design

A. Baseline Ray Tracing Unit Hardware

We design our baseline ray tracing unit to closely match the operation described by the *IMAGE_INTERSECT_RAY* instructions defined in AMD's RDNA3 ISA [1]. Ray data and BVH node data pointers are passed to the RT unit through the register file. The unit then fetches the requisite node data from the L1 data cache for each active thread. Each active thread in the instruction performs either one ray-triangle intersection test, or up to four ray-box intersection tests depending on the type of node data retrieved from memory. The results of the tests are then written to the instruction's result registers. Our baseline RT unit builds upon the GPU Ray Tracing model established in prior work [34], [51]. However rather than maintain a per-ray traversal stack in hardware, we return the results of each intersection test through the register file [1] and allow software to manage the traversal stack. There are several advantages of this approach. First, it allows for flexibility in adapting to different traversal algorithms [7], [19] and BVH widths [14], [33], [60]. Second, it enables software to manage the traversal stack in low-latency shared memory [70] rather than each thread's local memory space or choose to implement a stackless or short stack traversal algorithm. Lastly, it reduces the area and power overheads of storing per-thread traversal stack information within the RT unit. Figure 4 shows how our ray tracing unit integrates with the rest of the SM.

Each SM includes one RT unit shared across all four schedulers in a similar manner to the load store unit [5]. A warp is dispatched to the RT unit during the execution stage of the SIMT pipeline and a round robin arbiter is used to select between warps from different sub-cores if there are multiple RT warp instructions dispatched at the same time. Since each RT instruction must load its triangle or box node operand data from memory, the unit buffers multiple warp instructions in a *Warp Buffer* to enable memory-level parallelism similar to [51]. Upon dispatch, the active mask of the warp instruction is written into an empty warp buffer entry along with each thread's ray data and an address pointer to its node data. The active threads in a warp instruction that enters the warp buffer push their node data memory requests into the FIFO memory access queue. The FIFO memory access queue then accesses the L1 cache once per cycle. Upon a miss the request is forwarded to the interconnect, and eventually returned via the response FIFO. The valid mask bits are used to track which threads have gathered their node data or are still waiting to submit a memory access request to the memory access FIFO. When the active mask is equal to the valid mask, all of the warp instruction's data has been gathered and it is ready to be scheduled to the datapath.

B. Baseline Unified Single-Lane Datapath Pipeline

Our baseline design includes a pipelined unified ray-box and ray-triangle intersection test datapath similar to RayCore [42] to take advantage of the overlap in functional units necessary for the two computations. Ray tracing workloads tend to exhibit large amounts of thread-divergence leading to poor SIMT efficiency while performing ray-box and ray-triangle tests [11], [29], [51], [52], [70] and GPU hierarchical search applications often suffer from the same divergence performance issues. In fact, the RTNN algorithm [74] performs a pre-processing step to find spatially local queries and group them together in rays to reduce the effects of divergence. Rather than attempting to improve SIMT efficiency through warp repacking, we accept that poor SIMT efficiency is inherent to these workloads and provision a single-lane pipeline, similar to the approach described in an AMD patent [52]. The datapath scheduler issues a single thread's data into the pipeline per cycle according to the active mask, while ignoring lanes within the warp buffer that have an active mask bit of 0. After all of the active threads within the warp buffer entry have been issued to the datapath pipeline the warp buffer entry is cleared. In this manner, high throughput can be achieved even with warp instructions with a sparse active mask. The *Result Buffer* stores the per-lane results and then writes the result to the register file once all of the active threads in the warp instruction have completed (similar to the corner turn buffer described in Saleh et al. [52]).

The single-lane pipeline is shown in Figure 5. A single CUDA thread can be initiated into the pipeline each cycle, and the pipeline has multiple functional units (FUs) in each stage to capitalize on the intra-thread parallelism that exists in each of the different instruction types. Control logic enables or disables FUs at each stage and multiplexes the results to different pipeline register positions depending on the opcode for the instruction, enabling the reuse of FUs for different operations. The pipeline has a depth of 9 stages, and can support a mix of operating modes simultaneously, i.e. a thread executing a ray-box test can be scheduled the cycle after a thread executing a ray-triangle test without waiting for the first instruction to exit the pipeline. The baseline pipeline has two operating modes: ray-triangle and ray-box intersection testing. We base our ray-triangle intersection hardware on the watertight triangle intersection algorithm proposed in [67]. We remove the fallback to double precision from the algorithm for tie-breaking when the edge equation is calculated to zero, motivated by a recent Nvidia patent which describes a method without such a fallback [30]. Our ray-box intersection test hardware is based upon the commonly used slab intersection method [26], [27], [36], [48]. An overview of the steps of the ray-box and ray-triangle computations are shown in Figure 4 on the right. Our hardware supports testing a ray against four AABBs simultaneously as well as sorting any hit results in order of the closest hit values. Processing four nodes is useful while traversing a BVH4 acceleration structure where each node can have up to four children.

C. Hierarchical Search Unit Hardware Extensions

We augment the unit to additionally support euclidean and angular (cosine similarity) distance calculations and key comparisons with minimal hardware cost. Figure 6 shows the

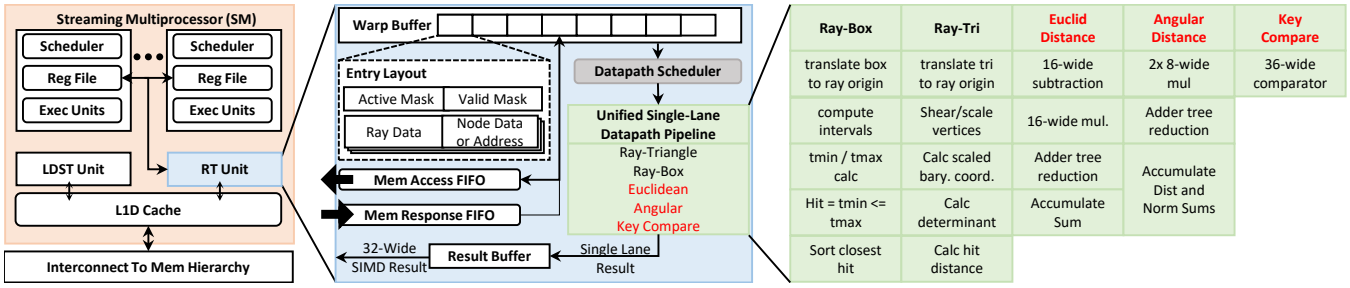


Fig. 4: GPU RT Unit Block Diagram. Items shown in red indicate functionality HSU adds to the baseline design

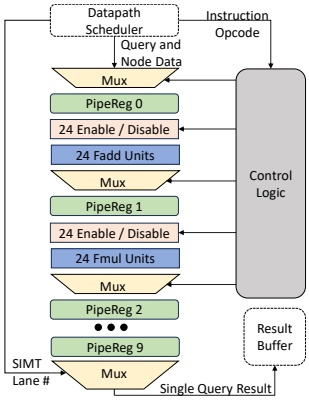


Fig. 5: Unified Single-Lane Datapath Pipeline (first two stages depicted). FUs (in blue) are shared across different instructions types and enabled / disabled according to Figure 6. HSU maintains the same unified structure but adds a few new FUs in some stages.

Stage	Ray-Box	Ray-Triangle	Euclidean	Angular	Compare
1	**24 Adds**	9 Adds	16 Adds		
2	**24 Muls**	9 Muls	16 Muls	16 Muls	
3	**36 Compares**	6 Adds	**8 Adds**	8 Adds	36 Compares
4	**4 Compares**	**6 Muls**			
5		3 Adds	**4 Adds**	4 Adds	
6		**3 Muls**			
7		**2 Adds**	2 Adds	2 Adds	
8		2 Adds	1 Add	**2 Adds (accum)**	
9	**2 QuadSort**	**5 Compares**	**1 Add (accum)**		

Fig. 6: HSU unified datapath pipeline functional units required for each operating mode. The total number of Functional Units (FUs) for each stage is indicated in bold (the maximum across all columns)

functional unit requirements for each stage of our unified datapath pipeline for each of its five operating modes. There is a significant overlap in the functional units necessary for the euclidean and angular modes with the baseline ray-triangle and ray-box calculations. The key-compare mode is implemented using the ray-box comparators in stage 3, and requires no additional functional units. Only two additional adders are required in stage 3, and one in stages 5, 8 and 9 to support the additional instructions. A 16-wide euclidean distance and 8-wide angular distance pipeline was selected for our design because it allows for the most reuse in functional units from the baseline RT unit. However HSU supports distance calculations for dimensions beyond these widths by accumulating partial results across multiple cycles (see section IV-F).

D. Baseline RT Unit instructions

Our baseline RT unit design has a single *RAY_INTERSECT* instruction similar to [1], which performs either a ray-triangle intersection test or four ray-box intersection tests depending on the type of BVH node retrieved from memory. The operands for the instruction are the same for both ray-triangle and ray-box tests: a pointer to the node data to retrieve and information about the ray to test. We pre-compute the inverse ray direction as well as the shear and k constants in the same way as [67]. These values are constant for each ray and can be reused for each intersection test performed by the ray. Four registers are used per thread to return the result of the *RAY_INTERSECT* instruction, but their contents vary depending on whether a ray-box or ray-triangle test was performed. If the node pointer provided as a source operand pointed to a box node, a ray-box intersection test is performed on each of that node's children. The results of the four tests are sorted by the RT unit and pointers to the four children nodes are returned in order of closet hit. If the ray did not intersect one of the child nodes a null pointer is returned.

When the input node pointer is to a triangle node a single ray-triangle intersection test is performed on the triangle primitive within that node. The hit status indicates whether the ray intersected or missed the triangle, and the triangle id is used to identify which triangle was tested. Additionally the unit returns the t parameter which indicates the distance/time of the ray origin to the hit point on the triangle. Rather than perform the expensive division operation necessary to calculate this value within the RT unit, we return the ratio as *t_num* and *t_denom* in the same way as [1].

E. HSU instructions

We augment our RT unit with three new instructions, *POINT_EUCLID*, *POINT_ANGULAR*, and *KEY_COMPARE* to support distance calculations and key comparisons within the HSU. The function of the first two of these instructions is to compute the distance between two n-dimensional points, a query point q and a candidate point c, to accelerate the critical computation of nearest neighbor search algorithms.

TABLE I: HSU instructions

Instruction	Description
RAY_INTERSECT	The baseline instruction used to perform one ray-triangle test or four ray-box intersection tests. Operands to this instruction include the ray data and a pointer to a BVH node. The type of test performed is determined by whether the node fetched from memory is a triangle node or a box node. Two different formats of data are returned in four registers depending on the type of test performed.
POINT_EUCLID	Performs a 16-wide squared euclidean distance (eq. 1) test between a query point and a candidate point. Higher dimensions are supported by aggregating the result across multiple instructions (section IV-F) Reduces the result to a single scalar value across all 16 lanes.
POINT_ANGULAR	Computes the 8-wide dot product between query and candidate point as well as the 8-wide squared norm of the candidate point. Higher dimensions are supported by aggregating the result across multiple instructions. Reduces the results to two scalar values. These quantities can be used to compute the angular distance (eq. 2), but the scalar division and square root operations must be performed outside the HSU.
KEY_COMPARE	Fetches a node containing up to 36 separator values. Compares the key to the values and returns a bit vector set to 0 if the key is less than the separator value and 1 otherwise. Used for traversing b-tree internal nodes.

The *POINT_EUCLID* instruction returns the squared euclidean distance between the query and candidate points, performing the calculation shown in equation 1.

$$d^2(q,c) = \sum_{i=1}^{n} (q_i - c_i)^2 \tag{1}$$

The query point data is passed to the HSU directly through the register file similar to the ray data for ray instructions. If the query and candidate points have $n \geq 16$ for euclidean or $n \geq 8$ for angular, multiple instructions must be generated with the accumulate operand bit set to 1. Section IV-F discusses the operation of multi-beat instructions in more detail. The angular distance between a candidate and query point is defined by equation 2.

$$cos\theta = \frac{q \cdot c}{\sqrt{\sum_{i=1}^{n} q_i^2}\sqrt{\sum_{i=1}^{n} c_i^2}} \tag{2}$$

The magnitude of the query point will remain constant for all candidate points it is tested against, so its value is pre-computed before the nearest neighbor search begins. Rather than perform the scalar division and square root computations shown in equation 2 within the HSU hardware, we instead only compute the dot product between q and c (equation 3) and the squared norm of the candidate point (equation 4).

$$dot\_sum = \sum_{i=1}^{n} c_i * q_i \tag{3}$$

$$norm\_sum = \sum_{i=1}^{n} c_i * c_i \tag{4}$$

These two quantities are returned by the instruction through the register file as *dot_sum* and *norm_sum* respectively.

The third new instruction, *KEY_COMPARE* is used while traversing internal nodes of a b-tree. An internal node of a B-tree with N child nodes has $N-1$ separator values in sorted (typically non-decreasing) order. The child node to traverse to next is decided by comparing the query key against these separator values to determine which values the key lies between. The *KEY_COMPARE* instruction performs this operation by computing the $key\_value < seperator\_value$ operation for up to 36 separator values at a time. The instruction has three arguments: the query key_value, a pointer to the internal node of separator values, and the number of separator values within the node. The instruction returns a bit vector with the result for each separator value.

F. Multi-beat distance instructions

The *accumulate* operand field is used to aggregate distance results when the data has width larger than the width of the pipeline (16 for euclidean and 8 for angular). The HSU will aggregate the results across multiple instructions rather than writing into the result buffer when this field is set. The compiler generates multiple HSU instructions in a row according to the number of beats necessary to process the distance for a point of a given dimension. For example, 9 instructions would be generated for an angular distance test on a point with a dimension of 65 because $\lceil 65/8 \rceil = 9$. The first 8 instructions would have the accumulate bit set, and the last instruction would have it cleared. When the last instruction is executed, the accumulated result will be written to the output buffer as normal. To ensure correctness, no instructions from a different warp can enter the datapath after the first accumulate instruction is executed. The accumulate bit is checked by the sub-core round robin arbiter to ensure this ordering. If the bit is set, the arbiter stops using round robin scheduling and instead only schedules instructions from the sub-core that issued the last instruction. We use a greedy then oldest warp scheduler, which ensures the same warp issues all of its accumulate instructions before scheduling a different warp. However, our design is also compatible with other warp scheduling algorithms. Non-greedy warp scheduling algorithms would simply need to check the accumulate field in the warp scheduler as well as the sub-core RT unit arbiter.

V. Methodology

A. Algorithm Selection

Three nearest neighbors implementations were selected for this study as well as a B-tree traversal algorithm:

- **GGNN** [18] is the current state of the art approximate nearest neighbors GPU implementation for high dimen-

sional data. It uses a hierarchical graph search structure similar to [37] and [72] and a parallel cache in shared memory for maintaining a priority queue of nodes to visit and the current closest K neighbors. A key aspect of their design is to assign an entire threadblock to each query, rather than mapping each query to a single thread, to exploit intra-query parallelism.

- **FLANN** [41] is a library for performing approximate nearest neighbor searches in high dimensional spaces. It provides support for several different acceleration structures on the CPU, but is limited to 3 dimensions and a k-d tree search index for its CUDA GPU implementation.
- **BVH-NN** is a CUDA GPU nearest neighbor search implementation we created based on the BVH ray tracing search algorithm presented in RTNN [74] and [15]. Our implementation does not use the Optix library [45] because it hides the BVH construction and traversal details. We construct our leaf AABB widths at two times the search radius with each data point in the center as described in [74]. The points are then sorted based on their Morton codes and a BVH is constructed using the the algorithm described in [24]. We implement a stack-based traversal which our kernel maintains per thread in shared memory. Our hierarchical data structure is a single monolthic bounding volume hierarchy and we do not pre-process the queries to maximize ray coherence.
- **B-Tree** [9] We evaluate a B-tree implementation from the Rodinia benchmark suite. The application has a maximum of 255 separation values per internal node, so the tree has a maximum branch factor of 256.

B. Datasets

The datasets we evaluate in this study are listed in Table II. They include a range of large dimension data relevant to computer vision problems taken from [2]. The column on the right indicates whether Angular or Euclidean distance is used. Additionally, we evaluate 3D point cloud datasets including traditional graphics models from [56] and a cosmological n-body simulation dataset from [16]. Random10k is random point cloud generated from a continuous uniform distribution. Two 1-dimensional datasets are used to evaluate the B-Tree implementation from [9]. Although we use the same 3D datasets to evaluate the FLANN and BVH-NN algorithms, we distinguish between them using "F" or "B" prefixes in all figures because they use fundamentally different hierarchical data structures and have unique performance characteristics.

C. Simulation infrastructure

We use Accel-Sim [28] with GPGPUSim 4.0 and the Volta V100 configuration for our performance evaluation. Table III shows the simulator parameters we used for our performance model. Our HSU performance model is based upon prior work integrating a ray tracing unit with GPGPUSim [34], [51], however we modify the unit to support per-instruction intersection tests rather than manage a traversal stack within the unit. To integrate our unit with Accel-Sim, we developed a trace post-processor that can replace sequences of SASS instructions with our HSU instructions described in Table I.

TABLE II: Evaluation Datasets

Dataset	Abbr.	Dimensions	#Points	Dist
deep1b [2]	D1B	96	9.9M	A
fashion-mnist [2]	FMNT	784	60K	E
mnist [2]	MNT	784	60K	E
gist [2]	GST	960	1M	E
glove [2]	GLV	200	1.18M	A
last-fm [2]	LFM	65	292K	A
nytimes [2]	NYT	256	290K	A
sift1m [2]	S1M	128	1M	E
sift10k [2]	S10K	128	10K	E
random10k	R10K	3	10K	E
bunny [56]	BUN	3	35.9K	E
dragon [56]	DRG	3	437K	E
buddha [56]	BUD	3	543K	E
cosmos [16]	COS	3	100K	E
B-Tree 1M [9]	B+1M	1	1M	N/A
B-Tree 10k [9]	B+10K	1	10K	N/A

TABLE III: Simulator Configuration

# SMs	80
Sub-cores / SM	4
Register file / Sub-core	64 KB
Warp Scheduler Policy	GTO
Max Warps / SM	64
RT Units / SM	1
Warp Buffer Size	8
Device Memory	32 GB HBM
L1D / Shared Memory Cache	128 KB
L1 Instruction Cache	128 KB
L2 Cache	24-way 6MB

VI. Experimental Results

A. Execution time spent on HSU operations

The performance of our design is limited by the share of execution in the application that can be executed on HSU. Figure 7 shows the simulated proportion of cycles in a baseline V100 (i.e. without an RT Unit) spent on operations that could be executed on the HSU. These values provide a theoretical maximum for the performance HSU could achieve, and include cycles spent on loading HSU operand data from memory as well as arithmetic computations.

B. Roofline Analysis

The roofline analysis [66] shown in Figure 8 provides insight into the compute and memory characteristics of the evaluated workloads. We define the performance of the HSU as the number of instructions completed (meaning the instruction exited the last stage of the datapath pipeline) by the unit each cycle. This definition provides a straightforward maximum compute bound of 1 intersection op per cycle per HSU, and is consistent regardless of the type of instruction processed by the unit. We define operational intensity as the number of intersection operations completed per cache line accessed from the L2, with a maximum memory bandwidth of one cache line per cycle.

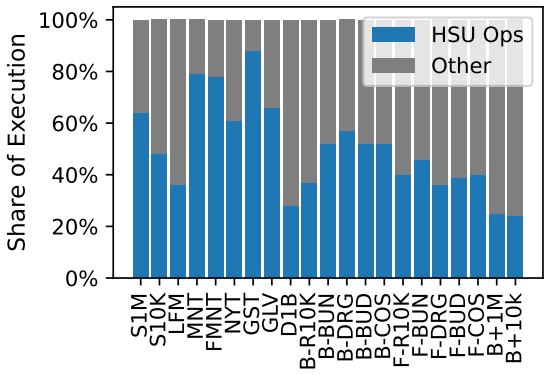


Fig. 7: Proportion of non-RT V100 GPU applications that can be accelerated with HSU operations

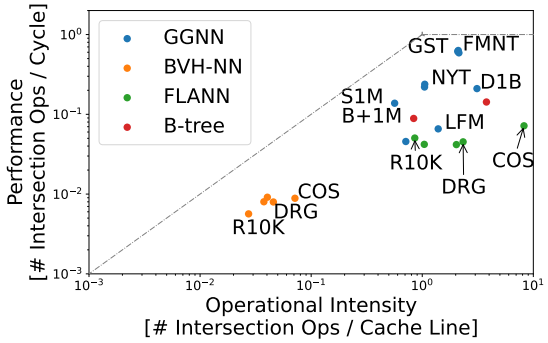


Fig. 8: Roofline Analysis of the HSU

The figure shows that none of the applications evaluated achieve full utilization of the HSU. Gist, mnist, and fashion-mnist are closest to the compute bound limit. Software changes to GGNN to better overlap instructions spent maintaining the priority queue of nodes to visit with HSU distance instructions could improve the utilization of the unit. The V100 GPU model used for the roofline analysis has a cache line size of 128 bytes. A euclidean distance instruction requires 64 bytes to be retrieved from memory, while an angular distance instruction requires 32 bytes. Therefore an operational intensity greater than 4 for a euclidean application or 8 for angular is indicative of data reuse between instructions. All of the BVH-NN applications were under the memory-bound portion of the plot, while not reaching the bottleneck line, indicating that further optimization could improve their utilization of the HSU but would ultimately remain limited by memory bandwidth.

C. Summary Performance

Figure 9 shows the speedup across all four search algorithms when using the HSU instructions. On average GGNN, FLANN, BVH-NN, and B+ improved 24.8%, 16.4% 33.9%, and 13.5% respectively. The BVH-NN implementation benefited the most from the HSU because the costly ray-box intersection tests were able to be accelerated in hardware. The BVH-NN kernel also processes the result of the ray-box intersection tests to maintain a per-thread traversal stack and determine which box nodes to process next. These operations are not accelerated within the RT unit and instead execute on the traditional SIMT SM. Each leaf node contains exactly one point in BVH-NN, so there are very few distance tests calculated in this algorithm. The BVH is able to cull the search space effectively and reduce the total number of euclidean distance tests to less than 200 for each query across all of the 3-D datasets.

GGNN maintains a parallel cache to track candidate points and a priority queue of the K-nearest neighbors [18]. The percentage of cycles spent on maintaining this cache (see Figure 7) is highly dependent upon application search parameters such as the number of layers in the graph, number of neighbors per point in the graph, and the desired amount of approximation in the result. As our unit does not accelerate the priory queue management operations, the largest speedups are seen in applications with a limited number of queue management operations per query. The results of the Euclidean and Angular distance tests are used to determine which candidate nodes in the graph to visit next, so our HSU is effectively accelerating the traversal of the hierarchical data structure in GGNN.

The FLANN kernel uses a k-d tree to determine which points should tested as potential candidates for the nearest neighbor of the query. Selecting a kd-tree child node during traversal is much less computationally expensive than a ray-box intersection test. K-d trees partition n-dimensional space along a single dimension for each level of the tree. The query point is simply compared against a split plane along a single dimension which requires only a single scalar subtraction and comparison. Due to its poor computational density we chose not to accelerate this portion of the traversal inside the HSU, and instead target the Euclidean and Angular distance computations and reductions that take place upon reaching a leaf node in the tree.

B+ Tree the smallest proportion of the algorithm that can be offloaded to the HSU of any of the applications tested (Figure 7). Further, the key compare operation performed by the unit is the simplest of the HSU operations, thus the instruction reduction that comes from using the HSU over the traditional SM compute resources is smaller than the other applications tested.

D. Graph Traversal Acceleration and High Dimension Datasets (GGNN)

GGNN is the current state of the art GPU implementation for approximate nearest neighbor search in datasets with a large quantity of features. In recommendation systems the current trend is towards training with increasingly larger datasets with a growing number of features. Our HSU provided an average speedup of 24.8% across all of the benchmarks tested using GGNN. The largest performance benefit was seen in Glove200 and NyTimes, both datasets which use angular distance during the search. These datasets both have a relatively large number of dimensions and the angular distance mode has a higher arithmetic intensity than the euclidean distance operating mode. However DEEP1B had the lowest performance benefit from the HSU at a speedup of 7.8%.

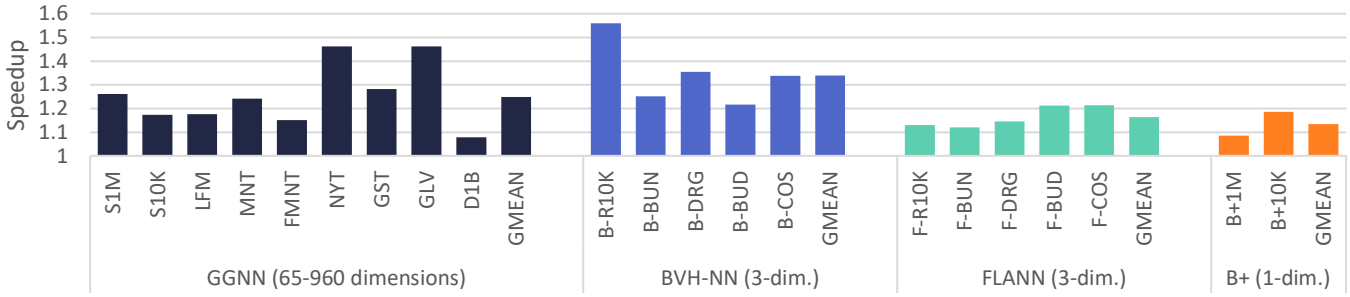


Fig. 9: Simulated speedup with HSU over a baseline without raytracing hardware support. The unit performs distance tests in GGNN and FLANN, box-intersection tests in BVH-NN, and key-compare tests in B+tree.

DEEP1B was the largest dataset evaluated and quickly became bottle-necked on the memory system after the distance calculations were offloaded to the HSU.

E. BVH-NN

Our BVH Nearest neighbors implementation achieved an average speedup of 33.9% when using the HSU to accelerate ray box intersection tests. Although our unit is designed to support four ray-box intersection tests per thread at a time, the BVH-NN implementation used a binary BVH tree. Thus only two child node boxes were traversed per thread at a time, and the application did not fully utilize the ray-box test hardware. A BVH4 tree would likely have better performance in our unit for this reason. Our BVH-NN implementation used a BVH construction algorithm known for its fast construction time but not for its quality [24]. This allows the BVH to be created for large datasets in a reasonable amount of time, but does not guarantee the fastest traversal. A more optimized BVH that uses surface area heuristic to determine partitioning would further improve performance. Additionally, unlike [74] our BVH-NN implementation did not perform any pre-processing steps to reduce ray-incoherence or select an optimized search radius. Although the HSU skips lanes with an inactive mask when scheduling threads into the single lane pipelined datapath, ray-incoherence causes non-coalesced memory accesses that reduce overall performance of the application.

F. FLANN and K-D Tree Traversal acceleration

The HSU achieved an average speedup of 16.4% over the non-rt accelerated k-d tree implementation, smaller than both the graph-based and BVH-based data structures. This is because the k-d tree structure uses a very simple test during traversal to determine whether to visit the left or right child node. K-d trees are split upon a single axis at each level, so only a single scalar value needs to be tested against the query during traversal. The HSU provides some benefit by accelerating the distance tests performed at when reaching the leaf nodes of the k-d tree, but there is little benefit of offloading the scalar value traversal test to a dedicated hardware unit.

G. Comparison with Software RT Unit Related Work

Previous works [6], [15], [21], [40], [63], [71], [74] have adapted their algorithm to use RT Unit hardware acceleration through the Optix API [45]. The HSU maintains ISA compatibility with the baseline RT Unit, thus it is fully compatible with any existing RT Unit application. RTNN [74] can execute directly on the HSU without changes and have equivalent performance to an RT Unit. The HSU has no high-dimensional nearest neighbor RT Unit software implementations to compare against because the baseline hardware and Optix API is restricted to three dimensions and cannot run GGNN. RTIndeX [21] is a state of the art GPU database indexing application that expresses keys as triangle primitives and uses ray casting for lookups. The authors identify several disadvantages of their approach due to the restrictive programming interface and hardware. First, Optix only supports single precision float values but the authors need to index 32-bit or 64-bit integer keys. RTIndeX circumvents this issue by mapping an integer value to 3 single precision float values. A triangle primitive consisting of 9 floats is then created with the 3d key value as its center. This method is both memory inefficient, because it maps a 32 or 64 bit value to a 288-bit triangle primitive, and messy because the 3D mapping no longer aligns adjacent keys in a direct line in space. HSU avoids these inefficiencies because it supports point primitive data types natively. Second, the authors note that the ray tracing mechanism and structure of the BVH is proprietary information and not made visible to the public [21], [46], therefore it is difficult to anticipate performance of different applications and workload conditions.

Proprietary GPU BVH implementations make it difficult to directly compare the performance of HSU and state of the art software approaches such as RTIndeX [21]. GPU simulators must create and traverse an open source BVH implementation which is likely substantially different from those used by proprietary GPU hardware vendors [51]. To overcome this obstacle we re-implemented RTIndeX in CUDA without Optix driver calls, instead using the same LBVH [32] used to evaluate the HSU. We then evaluated the performance of RTIndeX on our baseline RT Unit hardware using triangle primitives, and compared against an HSU version which represents key values natively. The HSU RTIndeX implementation using point values for keys had a 36.6% speedup over the RT Unit baseline while performing 163,840 lookup queries with 32-bit keys. As previously discussed, HSU has a 9:1 memory advantage for the key store because the baseline RT

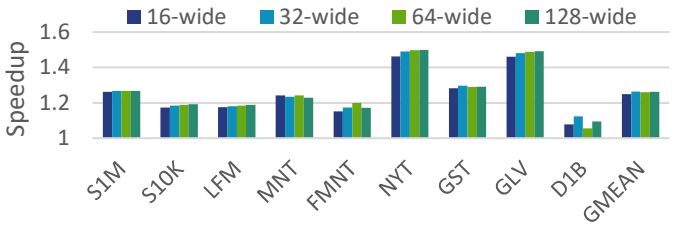


Fig. 10: Speedup with HSU at different datapath widths over a non-RT unit baseline. The datawidths in the legend refer to the width of the Euclidean operating mode, the width of the Angular operating mode is half this value.

unit hardware must represent 32-bit keys as 288-bit triangle primitives. This memory reduction allows leaf nodes to be processed faster during traversal because only a single point needs to be retrieved from memory rather than an entire triangle. Additionally, the HSU benefits from a smaller amount of input data transferred from host to device over PCIE, which scales linearly with the number of keys in the database.

H. Datapath Width Sensitivity

The width of the datapath determines how many *POINT_EUCLID* or *POINT_ANGULAR* instructions are needed to compute the distance between two points of a given dimension. A pipeline width of 16 was selected for the euclidean distance operating mode because it allowed for the most reuse of existing functional units with the ray intersection operating mode. The angular operating mode requires computing two values, the dot product between the candidate and query as well as the candidate point norm, so its effective width is half of the euclidean width to properly re-use functional units. Figure 10 shows the effect of datapath width on GGNN with high dimensional data. The dimensions of each dataset is listed in Table II. A larger datapath width results in fewer HSU instructions needed to compute and aggregate the distance. In general a larger width corresponds to a lower latency for distance computations which improves overall performance. This trend is seen in datasets such as sift10k, last-fm, and glove200. These improvements tend to be marginal as the increase in width has diminishing returns, but add a high cost to the total datapath area and power. In some cases increasing the datapath width can lead to decreased performance. This is because the larger width is also effectively increasing the size of the warp buffer because each entry can store a larger width of data. As will be further discussed in section VI-I, the HSU time shares access to the L1D cache with the load-store unit. Euclidean and angular distance instructions tend to miss in the L1 cache, thus a large number of outstanding HSU instruction memory requests can reduce the overall hit rate of the L1 cache for non-RT instructions.

I. Warp Buffer Size Sensitivity

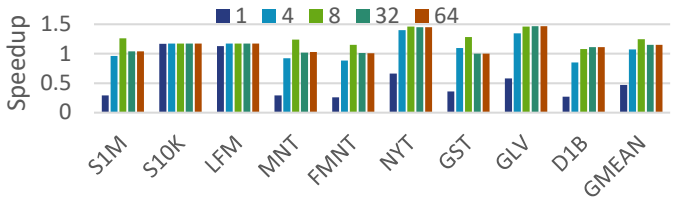
Figure 11 shows the effect of warp buffer sizes (number of entries) on the performance of three different hierarchical nearest neighbor search algorithms: graph based (GGNN), bounding volume hierarchy (BVH-NN), and k-d tree (FLANN). A single entry warp buffer is much too restrictive as it allows only one instruction from one sub-core to fetch its operands from the memory system at a time, without capitalizing on any memory level parallelism. Thus a single entry warp buffer is less effective than the baseline design because the load store unit can generate multiple accesses under a miss as long as there is available space in the miss status holding registers (MSHR). A warp buffer size of 4 is victim to a similar effect in applications that are memory bound after the distance computation is offloaded to the HSU such as MNIST, fashion-MNIST, and DEEP1B. We found a warp buffer size of 8 to provide the best amount of memory level parallelism for the least power and area cost across all applications. It performed the best in GGNN and BVH-NN and only marginally worse in FLANN. Previous work has also determined a warp buffer size of 8 to be the sweet spot in terms of performance and area savings [51]. While the authors of [51] suggested maintaining the warp buffer inside the ray-tracing unit, we note that these entries could alternatively be maintained in a reserved space within the register file to reduce the area cost of the baseline ray tracing unit.

An interesting effect occurs wherein larger warp buffer sizes result in poorer performance for some high-dimension datasets such as sift-1m, fashion-MNIST, and MNIST. In these datasets, a large number of memory requests from the HSU can reduce the performance of non-rt instructions. HSU instructions tend to miss in the cache, as they are accessing candidate points that have not been previously visited. At high warp-buffer sizes these cache accesses can lead to the MSHR to fill which restricts the memory-level parallelism of non-rt instructions. To avoid this contention, a private cache dedicated to the RT unit could be used, or a method of bypassing the L1 data cache for accesses generated from the ray tracing unit could be employed.

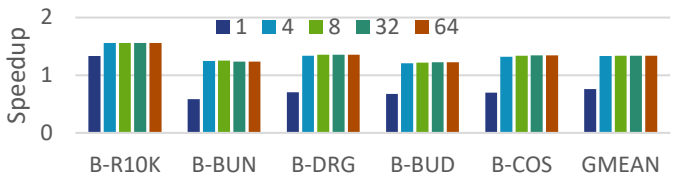
J. Memory System Effects

HSU can reduce the number of L1D cache accesses by coalescing spatially local accesses in sequential instructions present in the baseline into a single access in a HSU CISC instruction. Figure 12 shows the number of L1D accesses of the HSU applications normalized to a non-RT baseline. The BVH-NN applications most prominently display this effect, as several adjacent global load instructions are coalesced into a single access. This improves the efficiency of the MSHR by reducing the number of instructions that access the cache and hit on a pending miss request.

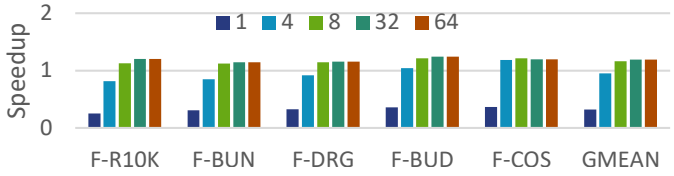
The high dimension applications in GGNN exhibit high L1D and L2 cache miss rates, whereas the lower dimension applications make better use of the caches (see Figure 13). Note that accesses that hit in the MSHR are considered hits, therefore by reducing the number of accesses in some applications the miss rate has increased (most notably in the BVH-NN applications). Figure 14 shows the average memory row access locality using a First Row-First Come First Served (FR-



(a) GGNN



(b) BVH-NN



(c) FLANN

Fig. 11: Speedup with HSU at different warp buffer sizes over a non-RT unit baseline. Evaluated for three different hierarchical data structures: graph (a), BVH (b), and k-d tree (c)

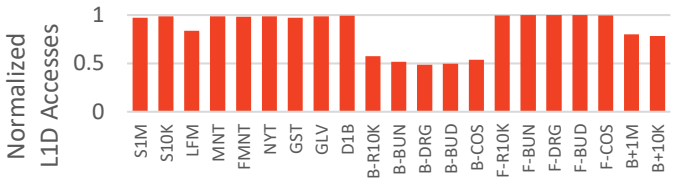


Fig. 12: HSU L1D Cache Accesses normalized to non-RT Baseline

FCFS) memory controller scheduler, which prioritizes queued accesses for the currently open row before oldest requests. HSU CISC instructions change the order of memory accesses seen at the memory controller, causing slight differences in available locality. This does not result in a large material difference since most of the locality is captured by coalescing and in the MSHRs.

K. Area and Power Results

We implemented and made publicly available [54] the unified single lane datapath pipeline (Figure 5) in RTL to assess the power and area impact of augmenting a baseline ray tracing unit with our new instructions. We generated two designs: a baseline datapath that supports only ray-triangle and ray-box intersection tests, and a HSU design that adds support for the additional instructions. Our new HSU instructions re-use the functional units used in the baseline ray-intersection

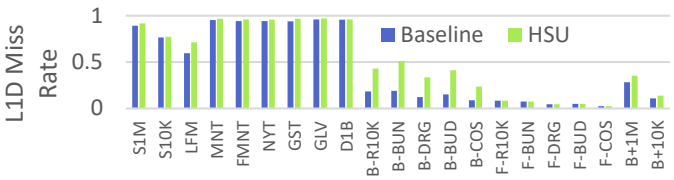


Fig. 13: L1 Data Cache Miss Rate

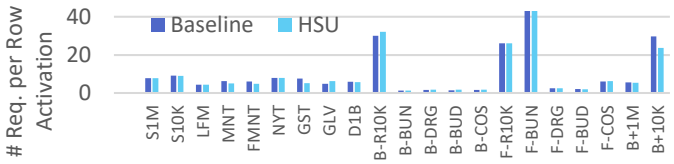


Fig. 14: Average Memory Row Access Locality

tests wherever possible, thus additional functional units are only required in stages 3,5,8, and 9 (Figure 6).

We implemented our design using Chisel [4] due to its flexibility creating parameterized designs and rapid verification capabilities. We synthesized our design with a 1 GHz clock using a 15 nm process design kit [39] in Cadence Genus. FUs for floating point arithmetic were sourced from Berkeley Hardfloat [20], [59]. We verified the correctness of our RTL design with test cases covering all ray-box, ray-triangle, Euclidean, Angular, and mixed modes. Figure 15 shows the area cost of HSU in several classes of resources normalized to the baseline design. The total area increase was 37% of the baseline datapath. Note that this represents an increase in area only for the datapath portion of the ray tracing unit, which is a small portion of the overall area of the entire ray tracing unit. Furthermore, this area overhead does not represent a production-ready design; to allow for flexible prototyping and design space exploration, we modeled our design without aggressive area optimizations. For example, we (1) adopted a fixed-latency pipeline architecture, (2) used individual registers at every stage for each operating mode and (3) performed costly floating-point number rounding at the end of each stage. Future optimizations could reduce the area overhead by (1) adopting a reservation table-based pipeline with flexible latency [57], (2) multiplexing pipeline stage registers for different operating modes, and (3) using a more efficient rounding scheme for intermediate results.

Figure 16 shows the dynamic power of each operating mode for our baseline and HSU. We calculated dynamic power for each operating mode by generating a random series of input stimulus. HSU increases the baseline ray-box and ray-triangle modes by 10 and 8 mW respectively. The power requirements for the Euclidean and Angular operating modes are 79 and 67 mW respectively. This is only 5 mW more than the baseline ray-box mode power cost.

VII. Related Work

To our knowledge no other work has studied the impact of adjusting the GPU ray tracing unit hardware to make it more amenable to general purpose hierarchical search.

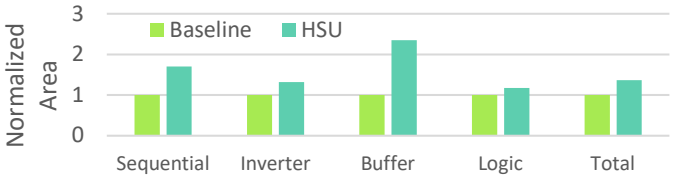


Fig. 15: Area of HSU datapath normalized to a baseline RT unit that can only perform ray-box and ray-triangle intersection tests. Note that the RT datapath is a small portion of the entire RT Unit, which is a small portion of the entire GPU SM.

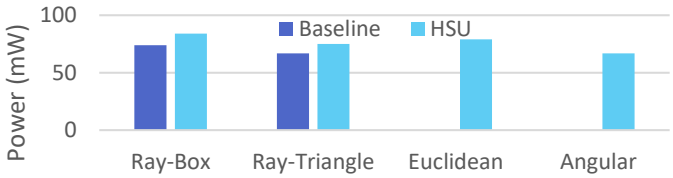


Fig. 16: Power of HSU datapath and a baseline RT unit that can only perform ray-box and ray-triangle intersection tests

However, there have been several works that have reformulated their search algorithm as a ray-tracing problem to fit into the graphics pipeline and make use of ray-tracing hardware acceleration. In [6] the authors optimize the problem of performing dependence analysis on sequences of operations. They cleverly recognize the equivalence of identifying dependencies between subsets of data and the visibility problem in computer graphics, which allows them to reduce the problem to a ray-casting problem and make use of RT-Unit acceleration. In [74] and [15] the authors present a nearest neighbors radius search algorithm formulated as a ray tracing problem and using the Optix library [45]. These software approaches are limited in the functionality they can provide and are not easily adaptable to a broad range of hierarchical search algorithms.

There are other studies that have looked at using custom hardware to improve the performance of nearest neighbor search. In [13] the authors propose hardware and a series of instructions to accelerate compression and decompression of 3-d point cloud data within the context of autonomous vehicle image recognition. The instructions proposed solely focus on the compression and decompression of floating point data to reduce pressure on the memory system. They do not provide custom instructions for performing distance tests in hardware or support for non 3-D datasets. Further, this study focuses on vector instructions in CPU implementations targeting 3D data used for autonomous driving systems. Large datasets such as those used for recommendation systems typically have a high number of features and benefit greatly from the increased memory bandwidth and parallelism available on the GPU.

Several works have proposed hardware accelerators for tree construction algorithms. In [12], [35], [61] the authors show the benefit of customized hardware for k-d and BVH tree construction in ray tracing applications. In contrast to these works, HSU targets the traversal phase of hierarchical search algorithms and would be compatible with dedicated hardware accelerators used during the construction phase. Other works have developed programmable traversal and intersection hardware accelerators, but they are typically closely coupled to a specific data structure. Early designs, such as [53] and [68], used SIMD packet tracing for k-d tree traversal and had dedicated traversal, intersection, and ray generation units. In [43] and [55] the authors propose a single-ray architecture for k-d tree traversal, similar to the HSU, because they recognized the poor performance per area of SIMD packet tracing caused by frequent ray incoherence. The work in [42] proposes a unified pipeline for traversal and intersection tests to improve functional unit reuse and reduce load imbalance problems. HSU builds upon this approach to support additional operations and a broader class of hierarchical data structures and primitives while reusing existing hardware in a baseline ray tracing unit. The architecture proposed in [69] accelerates k-d tree traversal for 3-D point cloud registration. They co-design a novel two-level k-d tree data structure and an accelerator to traverse and search the tree. Their design is limited to three dimensions and includes hardware recursion units that are tightly coupled to the layout of their data structure. Unlike specialized fixed-function accelerators, HSU provides the flexibility to adapt to different hierarchical data structures while reusing many of the hardware resources that already exist in GPU ray tracing units and would otherwise be unused.

VIII. Conclusion

In this work we examined the potential to expand the programmability of the ray tracing unit to hierarchical search data structures beyond bounding volume hierarchies. We presented a novel unified datapath capable of reusing the functional units necessary to perform ray-box and ray-triangle intersection tests to also compute high dimension distances and perform key comparisons. We showed how this expanded functionality can be used to accelerate three different nearest neighbor algorithms and a B-tree key-value store. We discussed the limitations of the current ray tracing units in terms of both performance and programmability. The expanded flexibility of our design allows general purpose GPU programmers to develop hardware accelerated search algorithms without reformulating their problem or using unfamiliar graphics APIs. We provided a power and area analysis to show the minimal overhead to the datapath in order to support these additional features. A dedicated GPU hierarchical search unit accessible through a standard compute programming interface has the potential to benefit a much broader class of applications than a traditional ray tracing unit.

Acknowledgment

We thank our anonymous reviewers for their feedback. This work was supported, in part, by NSF CCF #1943379 (CAREER), and the Applications Driving Architectures Research Center, a JUMP Center cosponsored by SRC and DARPA.

REFERENCES

[1] "RDNA3 Instruction Set Architecture Reference Guide," Advanced Micro Devices, 2023. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf

[2] M. Aumüller, E. Bernhardsson, and A. Faithfull, "ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms," *Information Systems*, vol. 87, p. 101374, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437918303685

[3] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance GPU B-Tree," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Association for Computing Machinery, 2019, p. 145–157.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proceedings of the Design Automation Conference (DAC)*, 2012, pp. 1212–1221.

[5] A. Barnes, F. Shen, and T. G. Rogers, "Mitigating GPU Core Partitioning Performance Effects," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 530–542.

[6] M. Bauer, E. Slaughter, S. Treichler, W. Lee, M. Garland, and A. Aiken, "Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: Association for Computing Machinery, 2023, p. 218–231.

[7] N. Binder and A. Keller, "Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time," in *Proceedings of High Performance Graphics (HPG)*. Eurographics Association, 2016, p. 41–50.

[8] N. A. Carr, J. D. Hall, and J. C. Hart, "The Ray Engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HHWS)*. Eurographics Association, 2002, p. 37–46.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[10] D. Cohen, *Voxel Traversal along a 3D Line*. USA: Academic Press Professional, Inc., 1994, p. 366–369.

[11] S. Damani, M. Stephenson, R. Rangan, D. Johnson, R. Kulkami, and S. W. Keckler, "GPU Subwarp Interleaving," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1184–1197.

[12] M. J. Doyle, C. Fowler, and M. Manzke, "A Hardware Unit for Fast SAH-optimised BVH Construction," *ACM Transactions on Graphics*, vol. 32, no. 4, 2013.

[13] P. H. E. Becker, J.-M. Arnau, and A. González, "K-D Bonsai: ISA-Extensions to Compress K-D Trees for Autonomous Driving Tasks," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, 2023.

[14] M. Ernst and G. Greiner, "Multi bounding volume hierarchies," in *IEEE Symposium on Interactive Ray Tracing*, 2008, pp. 35–40.

[15] 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, no. 1, pp. 25–48, February 2021. [Online]. Available: http://jcgt.org/published/0010/01/02/

[16] L. H. Garrison, D. J. Eisenstein, D. Ferrer, J. L. Tinker, P. A. Pinto, and D. H. Weinberg, "The Abacus Cosmos: A Suite of Cosmological N-body Simulations," *The Astrophysical Journal Supplement Series*, vol. 236, no. 2, 2018. [Online]. Available: http://dx.doi.org/10.3847/1538-4365/aabfd3

[17] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics and Applications*, vol. 4, no. 10, pp. 15–24, 1984.

[18] F. Groh, L. Ruppert, P. Wieschollek, and H. P. A. Lensch, "GGNN: Graph-Based GPU Nearest Neighbor Search," *IEEE Transactions on Big Data*, vol. 9, no. 1, pp. 267–279, 2023.

[19] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, "Efficient Stack-Less BVH Traversal for Ray Tracing," in *Proceedings of the 27th Spring Conference on Computer Graphics (SCCG)*. Association for Computing Machinery, 2011, p. 7–12.

[20] J. Hauser, "Berkeley Hardfloat," 2022. [Online]. Available: http://www.jhauser.us/arithmetic/HardFloat.html

[21] J. Henneberg and F. Schuhknecht, "RTIndeX: Exploiting Hardware-Accelerated GPU Raytracing for Database Indexing," *Proceedings of the VLDB Endowment*, vol. 16, no. 13, p. 4268–4281, 2023. [Online]. Available: https://doi.org/10.14778/3625054.3625063

[22] J. Johnson, M. Douze, and H. Jégou, "Billion-Scale Similarity Search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2021.

[23] J. T. Kajiya, "The Rendering Equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery, 1986, p. 143–150.

[24] T. Karras, "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees," in *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*. Eurographics Association, 2012.

[25] T. Karras and T. Aila, "Fast Parallel Construction of High-Quality Bounding Volume Hierarchies," in *Proceedings of the 5th High-Performance Graphics Conference (HPG)*. Association for Computing Machinery, 2013, p. 89–99.

[26] T. T. Karras, T. O. Alia, S. M. Laine, and J. E. Lindholm, "Beam Tracing," U.S. Patent US10242485B2 Mar. 26, 2019.

[27] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery, 1986, p. 269–278.

[28] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[29] S. Laine, T. Karras, and T. Aila, "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference (HPG)*. Association for Computing Machinery, 2013, p. 137–143.

[30] S. Laine, T. Karras, T. Alia, R. Ohannessian, W. P. Newhall Jr, G. Muthler, I. Kwong, P. Nelson, and J. Burgess, "Watertight Ray Triangle Intersection ," U.S. Patent US20220230380A1 Jul. 21, 2022.

[31] E. Larsen and D. McAllister, "Fast Matrix Multiplies Using Graphics Hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2001, pp. 43–43.

[32] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH Construction on GPUs," *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009.

[33] A. Lier, M. Stamminger, and K. Selgrad, "CPU-Style SIMD Ray Traversal on GPUs," in *Proceedings of the Conference on High-Performance Graphics (HPG)*. Association for Computing Machinery, 2018.

[34] L. Liu, W. Chang, F. Demoullin, Y. H. Chou, M. Saed, D. Pankratz, T. Nowicki, and T. M. Aamodt, "Intersection Prediction for Accelerated GPU Ray Tracing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. Association for Computing Machinery, 2021, p. 709–723.

[35] X. Liu, Y. Deng, Y. Ni, and Z. Li, "FastTree: a Hardware KD-tree Construction Acceleration Engine for Real-time Ray Tracing," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. EDA Consortium, 2015, p. 1595–1598.

[36] A. Majercik, C. Crassin, P. Shirley, and M. McGuire, "A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering," *Journal of Computer Graphics Techniques (JCGT)*, vol. 7, no. 3, pp. 66–81, 2018.

[37] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, 2020.

[38] D. K. Mandarapu, V. Nagarajan, A. Pelenitsyn, and M. Kulkarni, "Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing," in *Proceedings of the 38th ACM International Conference on Supercomputing*. Association for Computing Machinery, 2024, p. 14–25.

[39] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open Cell Library in 15nm FreePDK Technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD)*. Association for Computing Machinery, 2015, p. 171–178.

[40] N. Morrical, I. Wald, W. Usher, and V. Pascucci, "Accelerating Unstructured Mesh Point Location With RT Cores," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 8, pp. 2852–2866, 2022.

[41] M. Muja and D. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration." *Proceedings of the 4th International Conference on Computer Vision Theory and Applications (VISAPP)*, vol. 1, pp. 331–340, 01 2009.

[42] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, "RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices," *ACM Transactions on Graphics*, vol. 33, no. 5, 2014. [Online]. Available: https://doi.org/10.1145/2629634

[43] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T & I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing," *ACM Transactions on Graphics*, vol. 30, no. 6, p. 1–10, 2011.

[44] "Nvidia Turing GPU Architecture," Nvidia Corporation. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[45] "Nvidia OptiX 7.5 Programming Guide," Nvidia Corporation, 2022. [Online]. Available: https://raytracing-docs.nvidia.com/optix7/guide/index.html

[46] "Statement from NVIDIA about inspecting the BVH," Nvidia Corporation, 2023. [Online]. Available: https://forums.developer.nvidia.com/t/visualize-optix-generated-bvh-in-nsight-compute/253837/2

[47] H. Ootomo, A. Naruse, C. Nolet, R. Wang, T. Feher, and Y. Wang, "Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus," in *IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2024, pp. 4236–4247. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDE60146.2024.00323

[48] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016.

[49] Rapidsai, "Rapidsai/cuVS: Vector Search and Clustering on the GPU." 2024. [Online]. Available: https://github.com/rapidsai/cuvs

[50] S. M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," in *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*. Association for Computing Machinery, 1980, p. 110–116.

[51] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2022, pp. 263–281.

[52] S. J. Saleh, M. B. Kazakov, and V. Goel, "Texture Processor Based Ray Tracing Accleration Method and System," U.S. Patent 20190197761A1 Jun. 27, 2019.

[53] J. Schmittler, S. Woop, D. Wagner, W. J. Paul, and P. Slusallek, "Real-time ray tracing of dynamic scenes on an FPGA chip," in *Proceedings of the Conference on Graphics Hardware (HWWS)*. Association for Computing Machinery, 2004, p. 95–106.

[54] F. Shen, A. Barnes, and T. G. Rogers, "A Chisel Implementation of the HSU Datapath," https://github.com/purdue-aalp/raytracer, 2024.

[55] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, "TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1802–1815, 2009.

[56] "The Stanford 3D Scanning Repository," Stanford Computer Graphics Laboratory. [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/

[57] H. S. Stone, *High-Performance Computer Architecture*. Addison-Wesley Longman Publishing Co., Inc., 1987, pp. 127–150.

[58] C. J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society Press, 2002, p. 306–317.

[59] "Berkeley Hardfloat," https://github.com/ucb-bar/berkeley-hardfloat, UC Berkeley Architecture Research, 2023.

[60] K. Vaidyanathan, S. Woop, and C. Benthin, "Wide BVH Traversal with a Short Stack," in *Proceedings of the Conference on High-Performance Graphics (HPG)*. Goslar, DEU: Eurographics Association, 2022, p. 15–19.

[61] T. Viitanen, M. Koskela, P. Jääskeläinen, H. Kultala, and J. Takala, "MergeTree: A Fast Hardware HLBVH Constructor for Animated Ray Tracing," *ACM Transactions on Graphics*, vol. 36, no. 5, 2017.

[62] I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics*, vol. 26, no. 1, 2007.

[63] B. Wang, I. Wald, N. Morrical, W. Usher, L. Mu, K. Thompson, and R. Hughes, "An GPU-accelerated particle tracking method for Eulerian–Lagrangian simulations using hardware ray tracing cores," *Computer Physics Communications*, vol. 271, p. 108221, 2022.

[64] T. Whitted, "An Improved Illumination Model for Shaded Display," *Communications of the ACM*, vol. 23, no. 6, p. 343–349, June 1980. [Online]. Available: https://doi.org/10.1145/358876.358882

[65] S. Widmer, D. Pajak, A. Schulz, K. Pulli, J. Kautz, M. Goesele, and D. Luebke, "An Adaptive Acceleration Structure for Screen-Space Ray Tracing," in *Proceedings of the 7th Conference on High-Performance Graphics (HPG)*. Association for Computing Machinery, 2015, p. 67–76.

[66] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, p. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[67] S. Woop, C. Benthin, and I. Wald, "Watertight Ray/Triangle Intersection," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, pp. 65–82, 2013.

[68] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a Programmable Ray Processing Unit for Realtime Ray Tracing," in *ACM Transactions on Graphics (SIGGRAPH)*. New York, NY, USA: Association for Computing Machinery, 2005, p. 434–444. [Online]. Available: https://doi.org/10.1145/1186822.1073211

[69] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and Algorithms for 3D Perception in Point Clouds," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2019, p. 629–642. [Online]. Available: https://doi.org/10.1145/3352460.3358259

[70] H. Ylitie, T. Karras, and S. Laine, "Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs," in *Proceedings of High Performance Graphics (HPG)*. Association for Computing Machinery, 2017.

[71] 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.

[72] W. Zhao, S. Tan, and P. Li, "SONG: Approximate Nearest Neighbor Search on GPU," in *IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1033–1044.

[73] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-Time KD-Tree Construction on Graphics Hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, 2008.

[74] Y. Zhu, "RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: Association for Computing Machinery, 2022, p. 76–89. [Online]. Available: https://doi.org/10.1145/3503221.3508409