# Generalizing Ray Tracing Accelerators for Tree Traversals on GPUs


Donghoh Ha[1,2]
*Yonsei University*
Seoul, South Korea
dongho.ha@yonsei.ac.kr

Lufei Liu[1]
*University of British Columbia*
Vancouver, Canada
liulufei@student.ubc.ca

Yuan Hsi Chou
*University of British Columbia*
Vancouver, Canada
yuanhsi@ece.ubc.ca

Seokjin Go
*Yonsei University*
Seoul, South Korea
seokjin.go@yonsei.ac.kr

Won Woo Ro
*Yonsei University*
Seoul, South Korea
wro@yonsei.ac.kr

Hung-Wei Tseng
*University of California, Riverside*
Riverside, USA
htseng@ucr.edu

Tor M. Aamodt
*University of British Columbia*
Vancouver, Canada
aamodt@ece.ubc.ca



***Abstract*—Tree traversal is a fundamental operation in many applications, such as database indexing and physics simulations. Although tree traversals feature high parallelism, they are inherently divergent and irregular, leading to inefficient performance on GPUs. Tree traversals are also prevalent in ray tracing, which is executed on dedicated Ray-Tracing Accelerators (RTAs) in modern GPUs to mitigate inefficiencies such as control flow divergence and underutilization of memory bandwidth by irregular memory accesses. In this paper, we propose the Tree Traversal Accelerator (TTA) to replicate the success of RTAs in ray tracing for general tree traversal applications. TTAs extend RTAs to support tree structures and operations beyond those in ray tracing, such as B-Tree search and radius search algorithms, by modifying existing computing units. Despite TTAs' effectiveness, they still rely on fixed-function computations, making it challenging to support other tree-based applications such as N-Body simulation fully. Thus, we introduce TTA+ as an alternative design, which modularizes the RTA computing units and makes them programmable, trading some efficiency for flexibility. With less than 1% increase in RTA area, our proposals can achieve up to 5.4× speedup for B-Tree search, 1.7× for N-Body simulation, and 1.2× for select ray-tracing applications.**

***Index Terms*—GPU, Accelerator, Tree Traversal**


## I. Introduction

Tree data structures are used to efficiently organize and search data across a variety of domains and are increasingly important in our data-driven world. These structures are particularly prevalent in web indexing, databases, data mining, and file systems, where B-Trees [82], B+Trees [17], and R-Trees [27] are used to index data for fast retrieval. Database queries feature high parallelism [55], motivating the Niagara system [68] and making them an excellent candidate for execution on the massively parallel Graphics Processing Unit (GPU). However, tree-based indexing is inherently irregular and divergent, which leads to inefficient performance on modern GPUs [7], [96].

[1]Both authors contributed equally to this work.

[2]The author is working at MangoBoost (dongho.ha@mangoboost.io) now. This paper is done when the author was a visiting scholar at University of California, Riverside

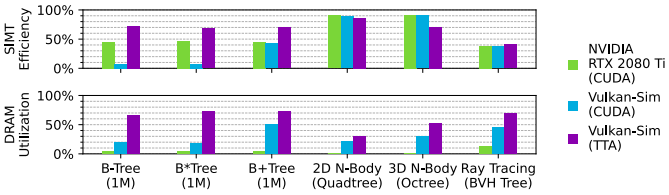



Fig. 1: SIMT efficiency and DRAM bandwidth utilization of tree traversal applications on GPUs with and without TTAs.

GPUs were originally designed to accelerate graphics rendering, but their adaptation of specialized hardware for diverse domains makes them dominant parallel processing devices. For example, GPUs accelerate general-purpose operations using CUDA Cores [50], AI/ML applications using matrix multiplication units [69], ray tracing pipelines using Ray Tracing Accelerators (RTAs) [1], [39], [71], and dynamic programming using DPX instructions [15]. GPUs are a desirable platform for tree-based applications because of their high memory bandwidth and position as a widely available commodity hardware option. As such, our goal is to improve existing GPU architecture to better support tree-based applications.

Graphs are a generalization of trees, and graph-processing accelerators are a popular solution for accelerating similar workloads that are memory-bound and irregular [18], [28], [67], [79], [101], [106]. Although these accelerators would be able to support tree traversals, they are often designed for specific graph processing algorithms and lack the broad support and ecosystem of GPUs. For example, graph accelerators cannot support a ray tracing application even though the underlying algorithm in the application is a tree traversal. Other accelerator cores on the GPU, such as Tensor Cores [42], may be useful for database queries [36], but can only handle regular workloads with predictable structures.

Algorithmic modification makes GPUs successful at accelerating tree traversal applications such as database indexing [7], [8], [84], [102]. However, there are still significant

inefficiencies caused by control flow and memory divergence, which are poorly handled by the Single-Instruction, Multiple-Thread (SIMT) execution model of GPUs. RTAs in GPUs are designed to mitigate these inefficiencies, and RTIndeX [34] and RTNN [105] have shown that RTAs can be adapted to support general-purpose tasks. However, adapting RTAs in software is non-trivial and requires significant performance overhead to manipulate target applications to fit the rigid RTA pipeline.

Figure 1 shows the average SIMT efficiency (percent of active threads per warp due to control flow divergence) and DRAM bandwidth utilization of several tree traversal applications on GPUs, profiled on an NVIDIA RTX 2080 Ti GPU and also measured using Vulkan-Sim [83] with configurations listed in Table II. SIMT efficiency and DRAM bandwidth utilization are low for these applications, indicating that the GPU is not being used efficiently. The difference in SIMT efficiency and DRAM bandwidth utilization between the physical GPU and Vulkan-Sim is due to different hardware configurations, highlighting that these issues persist across different GPUs. Although N-Body exhibits high SIMT efficiency, a single metric cannot directly map to the end-to-end performance. N-Body still suffers from low DRAM bandwidth utilization, implying an opportunity for performance improvement as we demonstrate in our evaluations (Figure 12).

As depicted by the rightmost bars in the figure, RTAs have been successful in addressing these inefficiencies for ray tracing applications but do not support other tree traversal applications. In order to achieve efficient execution of other tree traversal applications on a commodity hardware platform, we propose to augment the already existing RTAs in GPUs by introducing a modicum of programmability to balance between efficiency and flexibility. Studies have shown that domain-specific accelerators are more efficient than general-purpose cores [29] and mixing fixed-function and programmable units can provide the best of both worlds [46]. RTAs also benefit from the captive computer graphics market, which has been a strong driver of their development and adoption.

Based on these observations, we introduce an innovative Tree Traversal Accelerator (TTA) that extends the existing RTAs to support tree traversal applications. TTA allows programmers to handle more diverse tree structures by redesigning the frontend of the RTAs. To support more traversal algorithms beyond ray tracing, TTA exploits the observation that many frequently used traversal algorithms rely on computations that are subsets of the baseline RTA function units. Thus, TTA modifies the existing function units in the ray-tracing pipeline. Moreover, to further broaden the versatility of TTA, we introduce an additional TTA+ design that features modular components. TTA+ decomposes the operations within the intersection test into individual operation units (OP units) connected via an interconnect.

We evaluate both designs for their impact on several representative tree traversal applications including B-Trees, B*Trees, and B+Trees, and N-Body simulation and show that TTAs can achieve a geometric average of 2.4× speedup for B-Tree variants and 1.2× speedup for N-Body simulation with only 0.7% area overhead. We observe that the performance improvement also results in 15-62% energy reduction compared to using GPUs without TTAs. Performance of ray-tracing applications overall degrade by 8% on average with our modified TTAs, but we find individual ray-tracing workloads can achieve up to 1.2× speedup by applying optimizations enabled by our TTA architecture.

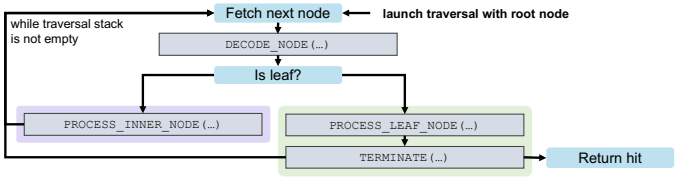



Fig. 2: Abstraction of the tree traversal algorithm

We make the following contributions in this paper:

- We identify the advantages of RTAs and reveal their potential in accelerating tree traversal applications with minimal hardware modifications.
- We propose two potential extensions of RTAs: TTA, which minimizes the hardware overhead, and TTA+, which maximizes the versatility.
- We provide a programming interface for both TTA and TTA+ to simplify access to RTAs by developers.
- We achieve up to 5.4× speedup for tree traversal applications with up to 62% energy reduction using only a 1% increase in total operation unit area.

## II. Background and Motivation

Tree data structures are widely used in applications such as databases, file systems, and computer graphics to organize and search data efficiently. This section provides an overview of tree-based applications, including ray tracing and how it executes on Ray-Tracing Accelerators (RTAs) in modern GPUs.

### A. Tree-Based Applications

Trees, by definition, are a subset of graphs that are acyclic and have exactly one parent per node except for the root node. Figure 2 shows a common flow for traversing through a tree structure using a while-loop and a traversal stack. The traversal starts at the root node and continues until there are no more nodes to visit or it reaches a termination condition. At each node, the traversal algorithm decodes the node contents and performs a predefined test operation based on the node type. Such a hierarchical manner of three structures makes data easy to search, making them a popular choice for many application domains. Note that, after the tree search, applications often post-process the search results into meaningful information, such as generating query responses in index searches, analyzing interactions in scientific applications, or determining shading values in rendering applications. These computations typically consist of element-wise vector operations that are straightforwardly accelerated on GPUs. Hence, this paper focuses on the tree traversal portion of the applications.

**Algorithm 1** Query-Key value comparison for a 9-wide tree

```
NChildren = 9
Input: Query, NodeKeys, ChildAddress
Output: NextChild, Found
function QUERY-KEY VALUE COMPARISON
    for i ← 0 to NChildren-1 do
        if (NodeKeys[i] == Query) then
            Found = true
            return Found, NextChild = NULL
        else if (Query < NodeKeys[i]) then
            NextChild = ChildAddress[i]
            return Found = false, NextChild
        end if
    end for
end function
```

**Algorithm 2** Point-to-Point distance Calculation

```
Input: PointA, PointB, Threshold
Output: result
function POINT-TO-POINT DISTANCE CALCULATION
    dis = PointB - PointA
    dis2 = dot(dis, dis)
    threshold2 = Threshold * Threshold
    result = (dis2 < threshold2)
end function
```

**Algorithm 3** Find primitives intersecting with a given ray

```
Input: root, ray
Output: hits
function TRAVERSEBVH(ROOT, RAY)
    stack.push(root)
    while stack is not empty do
        node = stack.pop()
        if (node is a leaf) then
            if ray intersects node primitive (Ray-Triangle) then
                hits.append(node)
            end if
        else
            if ray intersects node bounding box (Ray-Box) then
                for each child of node do
                    stack.push(child)
                end for
            end if
        end if
    end while
end function
```

B-Trees and its variants are one of the most common indexing structures used in databases to organize and search data efficiently [17], [82], with many works aimed at improving its performance [3], [7], [8], [25], [40], [41], [49], [84], [95], [96]. The B-Tree is a self-balancing tree data structure that maintains sorted data to allow efficient insertion, deletion, and search operations in large databases. Other variations, such as the R-Tree [27], apply the same principles to index multi-dimensional data and have also been widely adopted [5], [20], [21], [26], [77], [81], [97].

Searching a B-Tree involves traversing from the root node down to the leaf nodes, selectively navigating through child nodes based on Query-Key comparisons. Algorithm 1 describes the Query-Key value comparison operation of a single query value against the children key values of a tree node. The operation requires the addresses of the child nodes as inputs and outputs a boolean indicating whether the query value was found in the key values contained by the tree node and the address of the child node to visit next. The algorithm first checks for equality between the key values and the query value. If they do not match but the query value is smaller than the key value, then the query value lies between the previous and the current key values, and the traversal continues to the corresponding child node set as `NextChild`.

Tree structures are also popular in physics simulations for representing spatial data with octrees [12], [14], [19], [51] and k-d trees [22], [30], [35], [76], [80], [104]. In these applications, the tree structure is used to partition the simulation space and efficiently compute interactions between particles or objects. One particularly common algorithm is the Barnes-Hut algorithm, which approximates distant particles as a single massive particle to reduce the number of interactions that need to be computed [9], [45].

The algorithm relies on a Point-to-Point distance calculation at each internal node of the octree to decide whether to approximate the node as a single particle or to traverse to the child nodes for more accurate force approximations. Algorithm 2 shows the Point-to-Point distance calculation operation, which takes the query point, a point to compare, and a threshold as inputs. The operation outputs a boolean value indicating whether this distance is below the specified threshold. The calculation begins with a vector subtraction to represent the distance between two points, which is then squared to calculate the squared distance and compared against the squared threshold value.

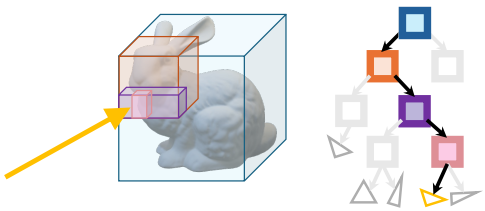


Fig. 3: Simplified example of BVH tree traversal for Bunny [88]

### B. Ray Tracing and RTAs

Ray tracing is another important tree-based application and is used in computer graphics to render photorealistic images by simulating rays of light interacting with a virtual scene. To render a scene, the ray tracing algorithm represents the scene geometry as a Bounding Volume Hierarchy (BVH) tree with axis-aligned bounding boxes (AABBs) and geometric primitives (usually triangles) at the leaf nodes. Modern GPUs include RTAs that accelerate the BVH traversal and intersection tests for each ray [1], [6], [37], [39], [71]. Algorithm 3 shows pseudocode for BVH traversal as a While-While loop [4], starting at the root node and continuing in a depth-first order, which matches the traversal abstraction in Figure 2. Figure 3 visualizes how this algorithm traverses through a BVH tree. On modern GPUs, this entire traversal is offloaded to RTAs using the `traceRay` instruction, where a state machine in the RTA warp scheduler tracks the status of each ray traversal using the traversal stack and orchestrates the while-loop in Algorithm 3. Rendering a complete frame requires several other

shader stages consisting of pixel-wise computations, which are executed on the GPU's general-purpose cores.

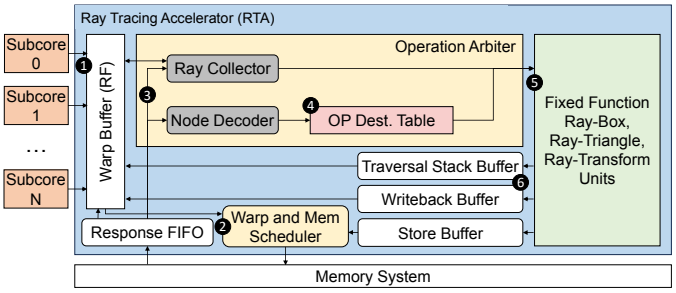
There is usually one RTA per Streaming Multiprocessor (SM) in modern GPUs, which autonomously processes ray traversals and intersections once the `traceRay` instruction is issued. Figure 4a shows a high-level example of the RTA architecture in modern GPUs, based on the Vulkan-Sim RTA model [83]. In this model, per-ray information such as the traversal stack and the ray origin and direction are stored in the warp buffer (❶) when the `traceRay` instruction is issued. In each cycle, the RTA's warp scheduler (❷) selects a warp to process based on the per-ray information from the warp buffer, and the hardware memory scheduler in the RTA coalesces and issues requests for BVH nodes to the GPU's memory system.

Memory requests return to the RTA through the RTA's response FIFO, and the operation arbiter (❸) processes the request. The operation arbiter first decodes the node type based on a flag in the BVH node (node decoder) and reads the per-ray information from the warp buffer of the warp where the returned memory request originated through the ray collector. With the node type and per-ray information, the operation arbiter can determine the appropriate intersection test to perform based on the operation destination table (❹) and forward the information to the fixed-function intersection units (❺).

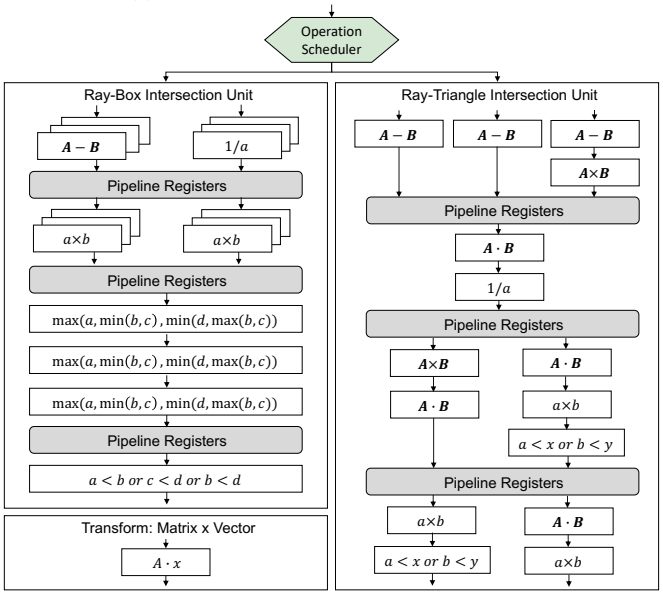
During the BVH traversal, a Ray-Box intersection test (Figure 5 left) is computed between the ray and the AABB at each tree node, continuing traversal recursively to the children only if they intersect. This test calculates the ray hit distances at each AABB plane $(tx0, ty0, tx1, ty1)$ and compares them with one another to check for a valid intersection. If the traversal reaches a leaf node, a Ray-Triangle intersection test (Figure 5 right) is performed and barycentric coordinates $(u, v)$ of the intersection point are computed by transforming the $(x, y, z)$ coordinates of the triangle vertices. The results then feed into one of three buffers (❻) before writing back to the warp buffer or memory. The barycentric coordinates are returned to the GPU's general-purpose cores for shading and material computation after traversal completes.

Figure 4b shows the detailed architecture of the RTA's operation units. The Ray-Box intersection uses a 4-stage fixed-function pipeline with a latency of 13 cycles, implementing the algorithm with floating point subtractors, multipliers, min/max units, and comparators that operate in parallel whenever possible. The min/max operation is specifically designed for the Ray-Box intersection test to collapse a sequence of min and max comparisons into a single operation, optimizing the check for ray hit distances against each AABB plane.

The Ray-Triangle intersection units take the ray origin, direction, and three triangle vertices as inputs and compute the barycentric coordinates of the intersection point, the ray hit distance, and a boolean indicating hit. The Ray-Triangle intersection also has four stages, consisting of multiple `vec3` floating point cross and dot product units, subtractors, multipliers, reciprocals, and comparators, pipelined to a total of 37 cycles. RTAs also include Transform units used in multi-level BVHs to translate between different coordinate spaces.




(a) Baseline GPU Architecture with RTA




(b) Pipeline Architecture of Operation Units in Baseline RTA

Fig. 4: Baseline GPU architecture with RTA

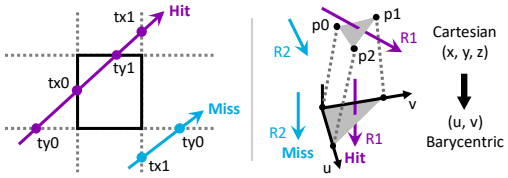



Fig. 5: Ray-Box intersection test (left) and Ray-Triangle intersection test (right) using the Möller-Trumbore algorithm [60]

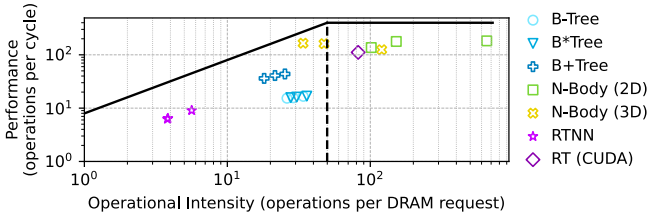



Fig. 6: GPU roofline model for tree traversal applications

## C. Advantages of RTAs

Figure 6 plots the roofline model for several tree traversal applications, highlighting a common characteristic of under-

utilized memory bandwidth due to limited data reuse and long-latency memory fetches. RTAs have introduced several advantages over traditional GPU compute cores for ray tracing, many of which can be generalized to other tree traversal applications that suffer from these inefficiencies:

**(1) RTAs employ fixed-function units to reduce dynamic instructions and improve performance.** Intersection tests at each node in the tree traversal are fixed and do not change based on the input data. As a result, RTAs can use fixed-function units to perform the same operations for each node, only distinguishing between inner and leaf nodes. This approach reduces the overhead associated with instruction fetching, decoding, issuing, and register file accesses by eliminating 91% of dynamic ALU and control flow instructions on average.

**(2) RTAs mitigate the SIMT divergence by representing the entire ray traversal as a single instruction (`traceRay`) on the GPU.** When threads diverge in the tree traversal, RTAs handle the control flow divergence directly, avoiding inactivating computing lanes in the SIMD pipeline [83]. Thus, the warps only need to synchronize the rays at the end of the traversal when RTAs complete the instruction. Prior works like Dynamic Ray Shuffling [57] mitigate SIMT divergence on GPUs for ray tracing but are still less effective than the RTA.

**(3) RTAs use a dedicated hardware memory scheduler to improve memory bandwidth utilization.** This scheduler coalesces node requests between threads when possible and arbitrates for one memory request into the GPU memory system per cycle. The dedicated memory scheduler better handles irregular memory access patterns by only focusing on node requests, allowing the scheduler to track more concurrent traversals and increase DRAM utilization by nearly 2×.

**(4) RTAs free up general-purpose cores for performing other tasks in parallel, effectively hiding the latency of ray traversal [13].** With the RTA, the compute cores can execute shading tasks for completed traversals while the RTA traverses the BVH for other rays. The GPU architecture can further exploit this parallelism if the RTA can support more tree traversal algorithms.

RTAs apply the same traversal algorithm in ray tracing as the abstraction used by other tree structures illustrated in Figure 2, with `PROCESS_INNER_NODE(...)` as Ray-Box intersections and `PROCESS_LEAF_NODE(...)` as Ray-Triangle intersections. Adapting RTAs to traverse other tree structures could leverage the existing benefits of RTAs with only minor modifications to support other node operations. Even ray-tracing applications can benefit from this change because not all ray-tracing applications use the Ray-Triangle intersection operation. For example, some scenes use spheres as the base geometry, which requires Ray-Sphere intersections [24] implemented with a programmable *intersection shader* instead.

From our observations, we find that tree-based applications typically focus on a distance relationship between some query and the node, which may be a value on a 1D number line in the case of index-based search or a coordinate in 2D or 3D space for N-Body simulations. Therefore, we propose a modified RTA that can support multiple different fixed node operations, which we refer to as the TTA. We also propose a more flexible design, TTA+, that generalizes the RTA further by providing a programmable architecture.

```
1  /* Define ray generation shader code */
2  const GLchar* raygenShader = R"
3  void simpleRayGenShader() {
4      Ray ray = makeRay();
5      traverseTreeTTA(ray, BVHRootAddr);
6  }";
7
8  /* Specify layout with byte offsets */
9  size_t internalNodeLayout[4] = [12, 12, 4, 4];
10 size_t leafNodeLayout[3] = [12, 12, 12];
11
12 /* Reserve fields for intermediate values according to the intersection test
       example (specified with byte offsets) */
13 size_t rayLayout[12] = [12, 12, 4, 4, 12, 12, 4, 4, 4, 4, 4, 4];
14
15 ConfigI("RayBoxProg.asm");
16 ConfigL("RayTriProg.asm");
17
18 DecodeI(internalNodeLayout);
19 DecodeL(leafNodeLayout);
20 DecodeR(rayLayout);
21
22 /* Specifies which ray, internal/leaf node fields should be checked for
       termination (specified with byte offsets) */
23 /* Ray tracing checks ray.tmin for termination at PC 20 of the Ray-Tri program
       */
24 ConfigTerminate("ray", 24, float, "Leaf", 20);
25
26 /* Launch tree traversal to GPU */
27 vkCreateTTAPipeline(device, info, "simpleRayTraversalShaderInRTAx");
28 vkCmdTraverseTree(GPUcommandBuffer, raygenShaderBindingTable);
```

Listing 1: Example of CPU Code using TTA APIs for Ray-Tracing

## III. Architecture

This section outlines the proposed architecture, TTA, and its programming model that enables hardware support for tree traversal algorithms. TTA extends the existing RTA architecture to support various input data structures and allows for adaptable intersection test units. To minimize the hardware overhead, TTA modifies the existing fixed-function operation units to support additional operations essential for decision-making and condition tests. We also introduce an additional configuration, TTA+, that generalizes TTA further by splitting the fixed-function operation units into granular OP units to allow programmable intersection tests, but requires more hardware modifications.

### A. Programming model

We introduce a new Application Programming Interface (API) that allows programmers to configure TTA and TTA+ and launch general-purpose tree traversals. This programming model mimics existing hardware ray-tracing APIs such as Vulkan [44] by replacing `traceRayEXT` and `vkCmdTraceRaysKHR` with new `traverseTreeTTA` and `vkCmdTraverseTree` instructions. Before launching the traversal, the programmer also needs to configure the ray data layout, node layouts for internal and leaf nodes, and the intersection tests using the new API calls `DecodeR`, `DecodeI`, `DecodeL`, `ConfigI`, and `ConfigL`, respectively. This approach also matches Vulkan, where programmers must first configure the stages of the ray-tracing pipeline before launching the traversal.

Listing 1 shows how a ray-tracing application can be adapted to the new TTA APIs. Lines 9, 10, and 13 specify the ray data layout, internal node layout, and leaf node layout, respectively, using byte offsets. These layouts are passed into

the `DecodeR`, `DecodeI`, and `DecodeL` API calls to set the `RTA_DECODE_NODE` stage from Figure 2, allowing the operation arbiter to properly decode data. Lines 15 and 16 configure the Ray-Box and Ray-Triangle intersection tests using the `ConfigI` and `ConfigL` API calls since they are no longer fixed-function units. To describe the termination condition of the tree traversal, `ConfigTerminate` specifies the data fields that TTA should check for when a specific PC of either the Ray-Box or Ray-Tri program is executed (Line 24). Finally, Lines 27 and 28 send the configurations to TTA and launch the tree traversal into the GPU command buffer.

To traverse a B-Tree with TTA, the programmer should specify the query key in the ray data layout and the child keys and addresses in the node data layout. Next, a value comparison is defined as the intersection test for `PROCESS_INNER_NODE` and `PROCESS_LEAF_NODE` using `ConfigI` and `ConfigL`. Lastly, the termination criteria for the traversal is specified using `ConfigTerminate` to terminate the traversal when the traversal stack is empty. The `traverseTreeTTA` API call is then used to start the traversal with the query and the root address of the tree. TTA will then execute the traversal following the While-While loop pattern, using the specified intersection tests and termination criteria at each node.

### B. TTA Architecture

As mentioned, the TTA architecture modifies the existing RTA architecture to support two additional operations that are common to several types of tree traversal applications: Query-Key value comparison and Point-to-Point distance calculation. Query-key value comparison is critical for efficient index-based tree search applications, including binary search, B-Tree, and various B-Tree-based algorithms. Point-to-point distance calculation is another crucial operation that 2D or 3D point-based search applications rely on, including N-Body simulation, Random Consensus Sampling in point cloud processing, and Radius search. Note that with the baseline RTA, these operations must be executed on general-purpose cores, which are less efficient than the fixed-function intersection pipeline.

We modify the node decoder in the operation arbiter to support different node data structures and make the operation destination table programmable to handle different dataflows required for each specific application. The existing Warp Buffer in the baseline RTAs is also repurposed as a general-purpose register file (RF), separate from the main GPU register file, that stores ray and node data with the programmer-defined data layout. Figure 7 shows the structure of the repurposed RTA Warp Buffer with dedicated space for ray and node data. Each ray or node entry contains 16× 32-bit ray registers (RR) or node registers (NR) and is addressable using the ray or node ID and the register ID. TTA implements Query-Key value comparison by modifying the Ray-Box intersection units and Point-to-Point distance calculation by rearranging the operations within the Ray-Triangle intersection units.

*1) Query-Key value comparison:* Figure 8 ❶ shows the modifications to support Query-Key value comparison. We

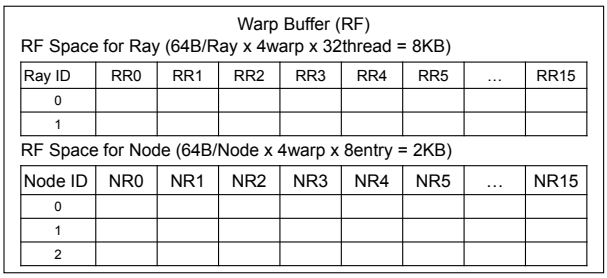



Fig. 7: Reusing Warp Buffer as General Purpose RF in TTA+

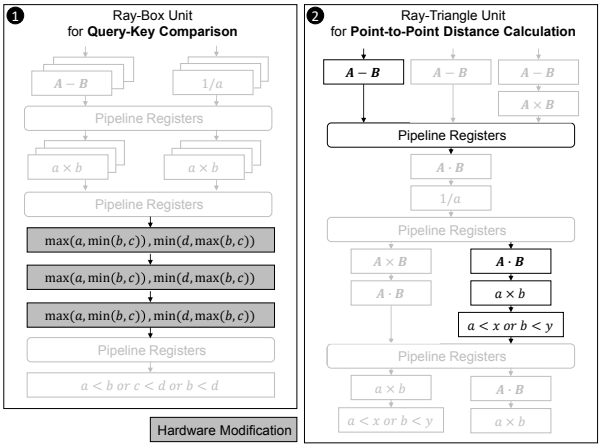



Fig. 8: Modifications in intersection test units in TTA

observe that the Query-Key comparison from Algorithm 1 can be adapted to use the existing min/max operations in the Ray-Box intersection units. Figure 9 (❶) shows the baseline min/max operation as a series of min and max comparators, grouped as a `minmax` sequence and a `maxmin` sequence. By replacing the `Tmin`, `tx1`, `tx2`, and `Tmax` values used in Ray-Box intersections with the key `K1`, query `X`, key `K2` and `K3` respectively, the min/max operations will generate results according to the table (❷). Hence, the relative magnitude of the query value compared to the keys can be determined by comparing the result of `minmax` sequence with `K1` and `X` and the result of `maxmin` sequence with `X`.

However, the Query-Key comparison also needs to check whether the key value matches the query value and determine the appropriate destination child node, which is missing in the baseline min/max operation (❶). To support this, we add equality operations to check whether the key value matches the query value (❸) and another three additional equality operations to determine the appropriate destination child node (❹). Each min/max operation compares the query value with three key values, allowing the Ray-Box intersection test units to process up to nine children at once. The output is the address of the child node and a one-hot vector length of three. The address of the next child node can be represented as an offset from the first child node's address. Thus, the three additional equality operations implemented at the last stage generate values of zero, one, or two when confirming the inputs' equality.

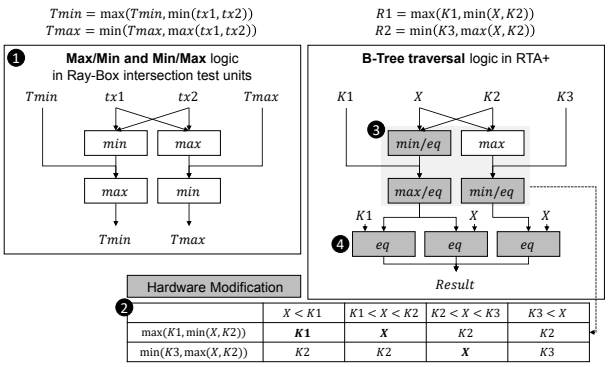



Fig. 9: Modifications in `min/max` and `max/min` operation for Query-Key comparison support

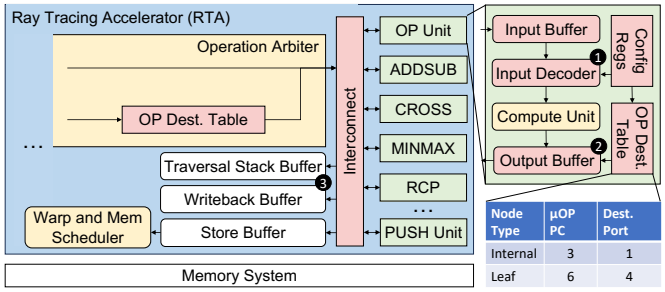



Fig. 10: TTA+ design overview with modified operation units

Additionally, since each `min/max` and `max/min` operation pair compares the input query value against three key values, the Ray-Box intersection test units can process up to nine children in a single instruction. Note that the baseline RTA operates on the BVH tree, in which nodes also have multiple children, and can traverse the child nodes iteratively. Thus, TTA can also traverse the child nodes iteratively when the number of children is larger than the capacity of a single Ray-Box intersection test unit. In our evaluation, applications employ a tree structure with a maximum of nine children per node, fully utilizing the computing units.

*2) Point-to-Point distance calculation:* We observe that the Point-to-Point distance computation in Algorithm 2 is a subset of Ray-Triangle intersection test units, which already include vector subtractors, vector dot-products, scalar multipliers, and scalar comparators. Thus, we support Point-to-Point distance computations by integrating an additional datapath within the Ray-Triangle intersection units. When a query needs to compute the Point-to-Point distance calculation with the Ray-Triangle intersection unit, the computation follows the bolded data path in Figure 8 ❷.

### C. TTA+ Architecture

While TTA already extends the usability of RTAs to more applications, there are many tree traversal applications with operation sequences that do not fit the intersection pipeline, such as the Ray-Sphere intersection or the force computation in N-Body simulation. Thus, we explore an alternative design point, TTA+, which favors flexibility by adopting a modular design philosophy that allows programmable intersection operations (referred to as $\mu$ops). TTA+ decomposes the fixed-function pipeline into individual operation units (OP units) connected via an interconnect, which is less efficient than TTA but more versatile. The granularity of the OP units in TTA+ aims to balance between increased latency per intersection computation and the flexibility to support more computation sequences, including optimizations for specific applications.

Figure 10 outlines modifications to the RTA architecture for TTA+, which breaks up the fixed-function ray intersection pipeline into individual parallel operations that communicate through an internal interconnect. Intersection tests are executed by visiting the OP units in a specified sequence, passing input data and intermediate results through the interconnect. Each OP unit includes computing units to execute its operations, configuration registers (Config Regs), and an operation destination table (OP Dest Table). The Config Regs controls the input decoder for selecting operands in the data from the interconnect. The OP Dest Table tracks the next OP unit destination port to send the result to based on the input node type and the $\mu$op PC, which identifies the correct destination port when there are multiple operations in an intersection test. The node type field allows the OP Dest Table to differentiate between internal and leaf node intersection tests. TTA+ is able to support a wide range of applications (e.g., Ray-Sphere intersection) by updating the Config Regs and OP Dest Tables that control the interconnect network before each kernel launch.

When an OP unit receives incoming data from the interconnect, the input decoder decodes source operands according to the Config Regs, then the operation is executed by the compute unit (❶). The input data includes all ray and node data and intermediate results from previous operations, as well as the previous $\mu$op's PC and destination port, which are needed for the input decoder. During execution, the OP Dest Table stores queries to find where to send the operation result. Once the compute unit completes execution, the result is stored in the output buffer along with the destination port, the $\mu$op PC, and the originally received input data that needs to be passed along to further operations (❷). The front of the output buffer is pushed to the interconnect whenever the destination port is available. When the intersection test completes, results are finally written back to the buffers and warp register (❸).

Table I lists the OP units included in TTA+ with their functionality. TTA+ includes units for functions such as floating point scalar arithmetic, vector dot and cross products, and comparison and logical operations. We also include a PUSH unit that handles pushing child addresses to the traversal stack and writes exclusively to the traversal stack buffer. Figure 11 shows a partial example of how the OP units can be used to implement a Ray-Box intersection test for TTA+, which would be configured with the `ConfigI` API. The format of the intersection test and termination criteria for TTA+ is a list of supported $\mu$ops that execute sequentially in the OP units. In this example, when a memory request for an internal type

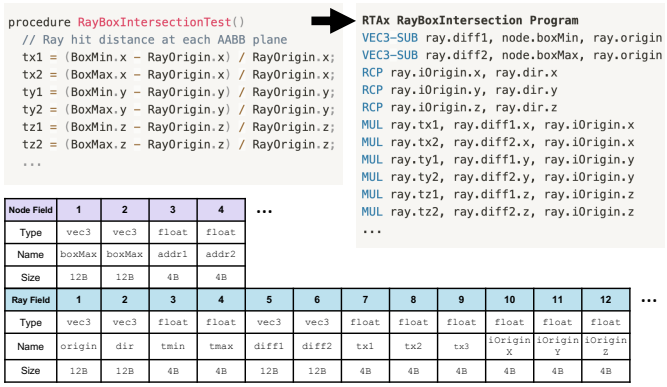



Fig. 11: Example of TTA+ intersection test

TABLE I: Operation Units in TTA+

| Unit Type | Description | Latency |
|---|---|---|
| Vec3 Add/Sub | Pipelined FP32 Vec3 ± Vec3 | 4 cycles |
| Multiplier | Pipelined FP32 Scalar multiply | 4 cycles |
| Reciprocal (RCP) | FP32 1/x, similar to RCPSS insn for CPUs | 4 cycles |
| Cross Product | Pipelined Cross product of two FP32 Vec3 | 5 cycles |
| Dot Product | Pipelined Dot product of two FP32 Vec3 | 5 cycles |
| Vec3 CMP | Return (a $\leq$ b) ? 1 : 0 for all Vec3 components | 1 cycle |
| MINMAX | FP32 MIN(a, MAX(b, c)),<br>also supports MIN(a, b) and MAX(a, b) | 1 cycle |
| MAXMIN | FP32 MAX(a, MIN(b, c)),<br>also supports MIN(a, b) and MAX(a, b) | 1 cycle |
| Logical Unit | Logical AND/OR/XOR/NOT | 1 cycle |
| SQRT | Square root | 11 cycle |
| R-XFORM | Ray transform matrix multiplication unit | 4 cycles |

node returns to the TTA+, the RTA frontend processes the request and identifies that the first $\mu$op is a vector subtraction, forwarding the ray and node data to the ADDSUB unit. The ADDSUB unit then decodes the ray and node data, executes the first two operations serially, and forwards the result to the next OP unit, RCP. Temporary variables and $\mu$op operands are specified using the ray and node layouts so OP units can decode and transfer intermediate results.

## IV. Experimental Methodology

RTAs are designed for ray tracing and only support operations in the ray tracing pipeline via dedicated APIs. However, TTA and TTA+ modify the hardware pipeline of the operation units and introduce new APIs to extend the usage of RTAs and support for more data structures. Therefore, we evaluate the performance of the baseline, TTA, and TTA+ using Vulkan-Sim [83], which simulates a cycle-level GPU architecture with a configurable RTA model for ray-tracing applications using the cross-vendor Vulkan API [44]. To model our proposed changes, we modify the functional simulation in Vulkan-Sim to support the programming model from Section III-A, which allows us to evaluate non-ray-tracing applications on the RTA and trigger hardware traversal without the Vulkan API. Vulkan-Sim does not include detailed models of intersection units, which we add to assess different operation unit configurations for TTA and TTA+. We use Vulkan-Sim with the default GPU configurations listed in Table II.

TABLE II: Vulkan-Sim Configurations

| | |
|---|---|
| # Streaming Multiprocessors (SM) | 8 |
| Max Warps / SM | 32 |
| Warp Scheduler | GTO |
| # Registers / SM | 32768 |
| Instruction Cache | 128KB, 16-way assoc., 20 cycles |
| L1 Data Cache + Shared Memory | 64KB, Fully assoc. LRU, 20 cycles |
| L2 Unified Cache | 3MB, 16-way assoc. LRU, 160 cycles |
| Compute : Interconnect : L2 : Memory Clock | 1365 : 1365 : 1365 : 3500 MHz |
| # TTA Units / SM | 1 |
| Warp Buffer Size | 4 warps |
| # TTA Intersection Units | 4 sets |

### A. Selected Applications

TTA and TTA+ are the first to extend the hardware functionality of RTAs to additional tree-based applications, which involve a high degree of parallelism commonly accelerated on GPUs. To evaluate the performance of TTA and TTA+, we select representative tree traversal applications that are commonly executed on GPUs [33], [100]. The benchmarks we evaluate are B-Tree variants, N-Body simulation, and radius search, which we implement using our proposed programming model and compare against CUDA implementations as the baseline. In our implementations, each thread represents a single query, and traversals follow a while-loop pattern [4]. Note that we focus on the tree traversal part of the applications to evaluate the performance advantages of our proposal since other computations in the applications can already be easily parallelized and accelerated on GPUs, making them orthogonal to our proposal. Hence, we believe that our evaluation can demonstrate the performance benefits of TTA and TTA+ for a wide range of applications.

Our baseline implementation of B-Tree performs similarly to the CUDA-optimized implementation [7], [8], and we evaluate B-Tree applications by randomly querying 1M keys in trees with 10k to 4M keys. We use a 9-wide B-Tree configuration to fully utilize the adapted Ray-Box intersection units in TTA. For N-Body, we evaluate 2D and 3D implementations of the Barnes-Hut approximation, reporting the performance of TTA and TTA+ on the force computation kernel. In our N-Body benchmark, inner nodes make use of the Point-to-Point distance calculation supported by both TTA and TTA+, while leaf nodes require the SQRT operation only accelerated on TTA+. We also optimize operations on the TTA+ by combining three multiplications into a single R-XFORM operation.

We evaluate radius search using RTNN [105], which adapts the problem to RTAs. The baseline RTNN implementation uses the OptiX ray tracing API and reports between 2.2× to 65× speedup over other CUDA implementations [105]. We reimplement RTNN with Vulkan for Vulkan-Sim, which we expect to perform similarly to the original OptiX implementation, and evaluate the performance of TTA and TTA+ on sets of 32k to 128k points from the KITTI dataset [23]. In the baseline RTNN, ray-sphere intersections are implemented in an intersection shader that executes on the GPU general-purpose cores as originally designed. However, such intersection shaders can be offloaded to the TTA+ by rewriting them as intersection tests, unlike TTA. To demonstrate the performance benefits of TTA+,

TABLE III: TTA+ Intersection Test Statistics

| Benchmark | Intersection Test | Total $\mu$ops | Vec3 SUB | MUL | SQRT | RCP | MIN/ MAX | CROSS | DOT | Vec3 CMP | Vec3 OR | R-XFORM |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B-Tree, B*Tree, B+Tree | Inner (Query-Key) | 12 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 3 | 3 | 0 |
| | Leaf (Query-Key) | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| N-Body 2D, 3D | Inner (Point-to-Point distance) | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| | Leaf (Force computation) | 5 | 0 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| *RTNN | Inner (Ray-Box) | 19 | 2 | 6 | 0 | 3 | 6 | 0 | 0 | 1 | 1 | 0 |
| | Leaf (Point-to-Point distance) | 5 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| *WKND_PT | Inner (Ray-Box) | 19 | 2 | 6 | 0 | 3 | 6 | 0 | 0 | 1 | 1 | 0 |
| | Leaf (Ray-Sphere) | 18 | 5 | 5 | 1 | 1 | 0 | 0 | 3 | 2 | 1 | 0 |
| LumiBench | Inner (Ray-Box) | 19 | 2 | 6 | 0 | 3 | 6 | 0 | 0 | 1 | 1 | 0 |
| | Leaf (Ray-Tri) | 17 | 3 | 3 | 0 | 1 | 0 | 2 | 4 | 2 | 2 | 0 |

we evaluate the performance of RTNN with the intersection shaders offloaded to the TTA+. These implementations are marked with a "*" symbol.

To evaluate TTA and TTA+ on ray-tracing applications, we use LumiBench [54], a ray-tracing benchmark suite designed for architectural research. Specifically, we use the representative subset of LumiBench that includes path tracing, ambient occlusion, shadows, reflections, procedural geometry, and alpha masking, covering a diverse range of ray-tracing behavior. One particular workload in LumiBench, WKND_PT, uses procedurally generated spheres in the scene. Similar to RTNN, WKND_PT uses the intersection shader pipeline stage to perform ray-sphere intersection tests, which we can optimize with TTA+. These results are also marked with a "*" symbol.

Table III lists the TTA+ $\mu$ops breakdown for inner node and leaf node intersection tests used by our evaluated applications for TTA+. Note that *RTNN, *WKND_PT, and LumiBench use two-level BVH structures which also require an R-XFORM $\mu$op between the levels.

### B. Area, Power, and Latency Evaluation Methodology

To evaluate the area and power of TTA and TTA+, we synthesize the designs with FreePDK45, specifically the modified operation units and the additional interconnect. For the interconnect in the TTA+, we use a 16x16 crosspoint switch that is publicly available from Intel FPGA Design Samples [38] and scaled up the interconnect width.

We augment the RTA model in Vulkan-Sim to include the individual operation units introduced for TTA+, simulating the execution latency of each operation unit, the crossbar latency and congestion, and any structural hazards of operational units. We use this model to evaluate the end-to-end performance of each application and estimate the energy consumption of intersection tests based on the active cycles per operation unit measured in Vulkan-Sim. Warp buffer accesses also contribute to the energy consumption, which we compute using the energy per access with CACTI7 [10] and the number of accesses from Vulkan-Sim. Energy consumption of general-purpose cores is obtained using AccelWattch [43]. $\mu$op latencies were referenced from Agner Fog's instruction tables [2], which document the latency of CPU instructions across multiple architecture generations.

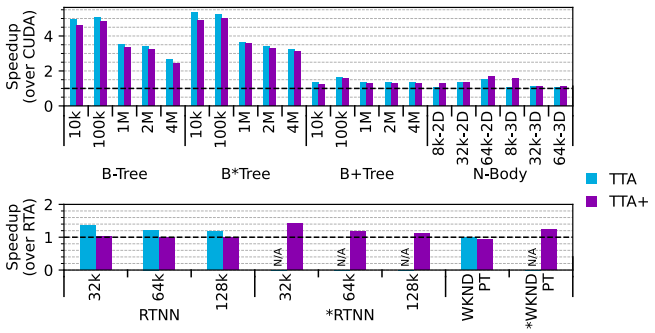



Fig. 12: Performance of selected applications on TTA and TTA+ relative to baseline GPU (CUDA applications – top, RTA applications – bottom)

## V. Results

### A. Tree Traversal Performance

Figure 12 illustrates performance improvements of our selected applications on TTA and TTA+ compared to the non-accelerated baseline GPU that only uses the general-purpose cores, with up to 5.4× speedup for queries on B-Trees and its variants. B-Tree and B*Tree queries suffer significantly from divergence on conventional GPUs, as highlighted in Figure 1, which is mitigated by the dedicated traversal function in TTA. B+Tree shows relatively low speedup as keys are only stored at the leaf nodes, so all queries traverse the same number of nodes and reduce control flow divergence. All variations of B-Tree performance differ with the number of keys stored in the tree, performing better when queries outnumber keys. In general, TTA+ performs slightly worse than TTA due to longer node processing latencies resulting from the additional interconnect overheads to support programmability.

The performance improvements on B-Tree queries are a result of the same RTA advantages that benefit ray tracing, as described in Section II-C. Similar to ray tracing, B-Tree traversals show low SIMT efficiency and, as a result, low memory bandwidth utilization. These effects are more pronounced in B-Tree traversals because of variations in the number of children per node, allowing B-Tree traversals to benefit more from RTAs. Figure 13 shows how the TTA helps improve the utilization of the DRAM bandwidth, which can be very beneficial to these memory-bound applications.

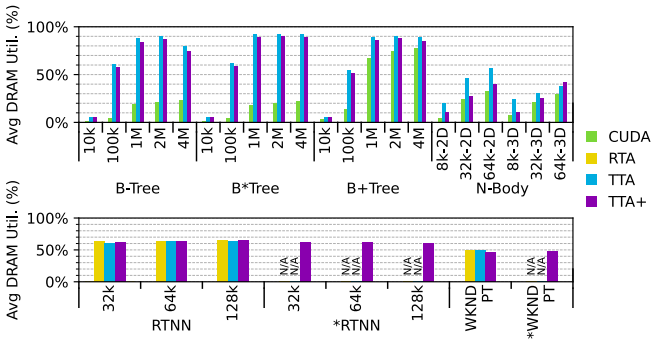



Fig. 13: DRAM utilization of selected applications on the non-accelerated baseline GPU, baseline RTA, TTA, and TTA+

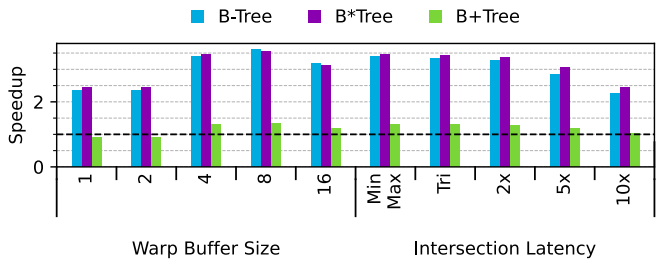



Fig. 14: TTA config sensitivity of 1M key B-Trees

For 2D and 3D N-Body simulation, we observe between 1.1-1.7× speedup by using the TTA to perform the tree traversal instead of the GPU compute cores. We find that N-Body simulation is particularly sensitive to TTA+ overheads because its traversal logic is more complex, while its memory accesses are more regular in comparison to B-Tree queries. Since the N-Body simulation features heavy computations after the tree traversal, we also evaluate the performance of merging the tree traversal kernel and the post-processing kernels, which allows the TTA and the general-purpose cores to work in parallel as explained in Section II-C. We observe that performance further improves by 1.2×, raising the overall speedup to 1.9× on TTA+. This result suggests that there is an opportunity to optimize the TTA+ by exploiting parallel computation between the TTA and the general-purpose cores.

RTNN and WKND_PT applications already utilize the RTA in their baseline implementations. However, simply by replacing costly intersection shaders with TTA, RTNN radius search performance improves by up to 1.4×, in addition to existing RTNN speedups over CUDA implementations of radius search. The same optimizing on WKND_PT is unsupported by TTA because it does not include a square root unit. For TTA+, the baseline implementation of RTNN experiences a slowdown due to the longer Ray-Box intersection latency, but with the optimization of replacing the intersection shaders (*RTNN), we observe up to 1.4× speedup.

*1) Sensitivity to TTA and TTA+ configurations:* We evaluate the sensitivity of B-Tree query performance on TTA to different warp buffer and intersection latency configurations, as shown in Figure 14. The warp buffer size has a higher impact on performance because it determines the number of queries that can be processed in parallel. Performance improves with more warps until it saturates at eight warps for all B-Tree variants, after which memory requests from additional queries begin to interfere with one another. Intersection latency configurations have a smaller impact on performance because these variations are mostly overshadowed by the memory access latency. For example, there is almost no difference between a configuration of TTA that isolates the min-max unit for a 3-cycle intersection versus a configuration that uses the full Ray-Triangle unit with a 13-cycle latency. Even if we increase the intersection latency significantly to 10×, TTA still achieves a 2.25× and 2.45× speedup for B-Tree and B*Tree, respectively.

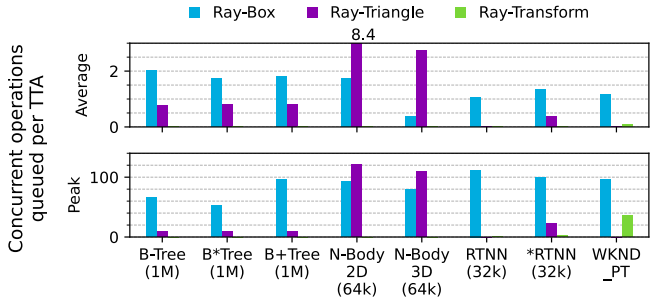



Fig. 15: TTA Intersection unit utilization for sample applications. (*WKND_PT is not supported by TTA)

*2) Intersection unit utilization:* Figure 15 shows TTA intersection unit utilization for the applications we evaluate, with both the average and peak number of concurrent threads queued and executing in each unit. We observe that node processing occurs in bursts, with high peak utilization of the intersection units, followed by periods of low utilization as TTA waits for memory accesses to return. However, even the peak number of active threads in the intersection units is much lower than the maximum pipeline register stages available. This observation highlights the potential to reduce some overheads in TTA+ by strategically reducing the number of parallel operation units to reduce the silicon area and power consumption, which we leave for future work. Figure 15 also demonstrates how RTNN can benefit from TTA by repurposing the previously idle Ray-Triangle units for distance calculations. We evaluate the utilization of the intersection operation units for TTA+ in Section V-C2.

### B. Ray-Tracing Performance

We evaluate the representative subset from LumiBench [54] to evaluate the impact of programmability overheads of TTA+. Figure 16 (left) shows the performance of each workload from LumiBench, showing a 8% slowdown on average as a result of the additional overheads of TTA+. Importantly, this performance is observed on unmodified workloads, which cannot employ many known ray-tracing optimizations due to the fixed functionality of the baseline RTA. For example,

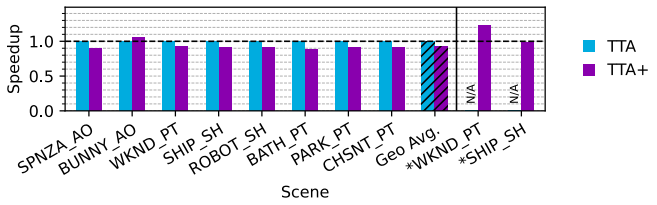



Fig. 16: Performance of LumiBench on TTA+ relative to baseline RTA

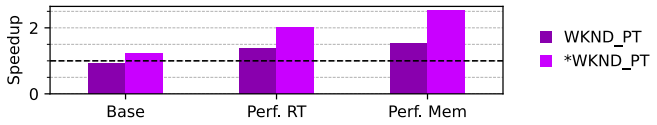



Fig. 17: Limit study of TTA+ with architectural improvements on WKND_PT and *WKND_PT

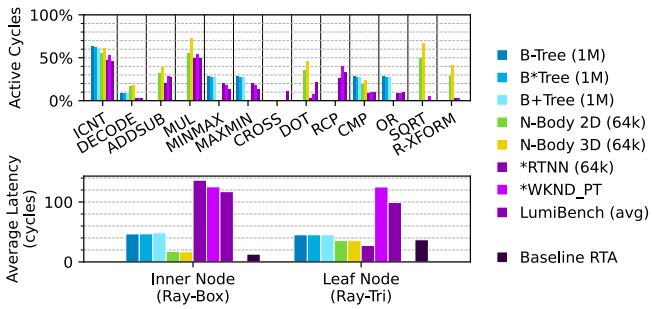



Fig. 18: OP unit utilization (top) and average intersection latency (bottom) for applications on TTA+

the SHIP scene includes many long and thin primitives, which are known to be inefficient with BVHs, and several optimization techniques have been proposed to address the problem [65], [91], [94], [99]. TTA+ enables the SATO optimization [65], which recovers the performance loss on SHIP_SH (*SHIP_SH in Figure 16). We expect that many other ray-tracing optimizations are enabled by the flexibility of TTA+, which will offset the performance drop and produce speedups for these workloads.

As an example, we evaluate how the square root unit in TTA+ can accelerate the WKND_PT workload, which uses a custom intersection shader by default to support ray-sphere intersections for procedural geometry in the scene. Intersection shaders are inefficient in the ray tracing pipeline but are also the only option available on an unmodified RTA. With TTA+, we can apply an optimization by replacing the intersection shader with TTA+ operations to execute a ray-sphere intersection. While naively running WKND_PT on TTA+ causes a slowdown, optimizing the workload improves performance by 22% (Figure 16, *WKND_PT).

Architectural improvements to TTA+ such as adding a dedicated prefetcher [16] or employing a prediction unit [53] can further improve performance. In Figure 17, we show that TTA+ optimizations are orthogonal to architectural improvements through a limited study on WKND_PT. We simulate a system with zero-latency node fetches (Perf. RT) and zero-latency memory accesses (Perf. Mem), both of which compound the existing performance gains of TTA+ with *WKND_PT.

### C. Hardware Overheads

*1) TTA Overheads:* For TTA, the additional hardware overhead is minimal as it only requires modifying the Ray-Box intersection units to support Query-Key value comparisons. The Ray-Triangle unit is unmodified. We add comparators after the `min/max` and `max/min` units and pipeline bypassing logic in the Ray-Box intersection unit, which increases the area from 0.2708 to 0.2756 $mm^2$ at 45nm. This is a 0.0048 $mm^2$ or 1.8% increase in additional area over the baseline Ray-Box unit. Power consumption of the Ray-Box intersection unit increases from 259.4 to 261.1 mW, which is a 0.7% increase.

*2) TTA+ Overheads:* Since TTA+ is a modular design where the number of components is flexible, we report the area of individual components for one complete set of operation units and compare them to the baseline Ray-Box and Ray-Triangle intersection units in Table IV. We find a 120B interconnect is sufficient to accommodate ray tracing, the most data-intensive workload in our evaluation, which contains a 64B node, 32B ray, and 24B intermediate values. For one set of operation units to provide support for ray-tracing applications, TTA+ uses 10.8% less area than the baseline because TTA+ reuses more operation units, sacrificing parallelism for a reduced area. However, an additional SQRT unit is necessary to support our new optimized workloads, which increases the area by 36.4% over the baseline. The overhead of the SQRT unit is the main source of the increased area in TTA+ and can potentially be avoided by connecting the TTA+ to the existing Special Function Units (SFU) [70] in the GPU instead. Deciding on operation units to include in the TTA+ is a design tradeoff between intersection latency and area.

Figure 18 (top) shows the utilization of the operation units in TTA+ for the applications we evaluate. We observe that different applications have different utilization patterns, which suggests that the optimal number of operation units for TTA+ is workload-dependent. However, we implement our TTA+ with one of each operation unit, which is the most general configuration, and we find there are no significant bottlenecks in the application we evaluate.

The majority of the latency overheads in TTA+ are due to the serialized operations and the interconnect (ICNT), which are a result of the modular design of TTA+. Figure 18 (bottom) shows the average intersection latencies on the TTA+ with these overheads, with Ray-Box intersection latency in ray-tracing applications increasing by nearly 10×. However, even with this large increase in latency, ray-tracing performance only drops by 8%, as shown in Figure 16. We leave the exploration of more optimized configurations of TTA+ and support for parallel operations to future work.

TABLE IV: Comparison of Baseline RTA Area vs. TTA+ Area

| Baseline Components | Area ($\mu m^2$) | % Baseline Area | TTA+ Components | Area ($\mu m^2$) | % TTA+ Area | Baseline Comparison |
|---|---|---|---|---|---|---|
| Baseline Ray-Box Unit | 270779.1 | 45.0% | Interconnect 16x16 (120B wide) | 177902.2 | 21.7% | |
| Baseline Ray-Triangle Unit | 331299.0 | 55.0% | Vec3 Add/Sub Unit | 17424.2 | 2.1% | |
| | | | Multiplier | 9551.7 | 1.2% | |
| | | | MINMAX | 2176.6 | 0.3% | |
| | | | MAXMIN | 1895.0 | 0.2% | |
| | | | Cross Product Unit | 74734.1 | 9.1% | |
| | | | Dot Product Unit | 40271.1 | 4.9% | |
| | | | RCP* Units x 3 | 212991.3 | 25.9% | |
| **Baseline Total** | **602078.1** | **100.0%** | **TTA+ without SQRT** | **536949.1** | **65.4%** | **-10.8%** |
| | | | SQRT Unit | 284367.2 | 34.6% | |
| | | | **TTA+ Total** | **821316.3** | **100.0%** | **+36.4%** |

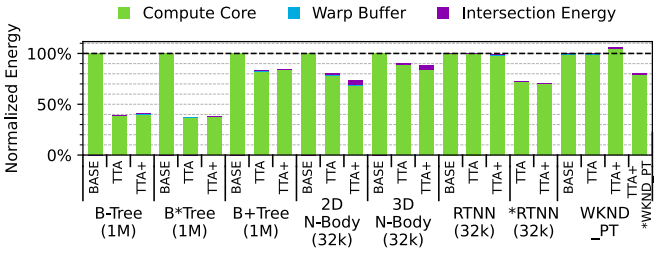



Fig. 19: Energy results for baseline, TTA and TTA+

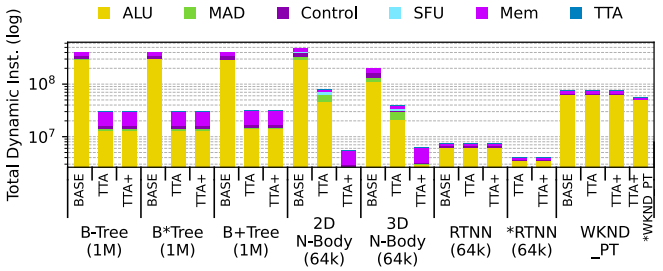



Fig. 20: Breakdown of total dynamically executed instructions for baseline, TTA, and TTA+

*3) Energy Results:* We compare the end-to-end energy consumption of TTA and TTA+ normalized to the baseline GPU with unmodified RTAs (BASE) in Figure 19, with an energy breakdown of general-purpose compute cores, warp buffer access, and intersection tests using Ray-Box, Ray-Triangle, or OP units. The *Compute Core* energy consumption includes the execution units in the general-purpose cores and the memory system from memory requests issued from both the general-purpose cores and RTAs. Notably, this reduction in energy consumption matches the reduced number of dynamically executed instructions in Figure 20, highlighting the benefits of offloading computations to TTA and TTA+. In particular, a single TTA instruction replaces the dynamic instructions from an entire traversal loop, reducing the number of dynamically executed instructions by 91% on average. TTA instructions only account for 2% of the total dynamic instructions on average.

*Warp Buffer* energy is from reading ray and node data from the warp buffer, and *Intersection Energy* tracks the energy usage of the intersection units. Both TTA and TTA+ have similar energy results for B-Tree applications, with 15-62% less energy consumption than the baseline GPU. TTA+ OP units consume more energy in N-Body applications due to the complex computations, but intersection energy is generally insignificant compared to the energy savings due to the reduced execution time. For applications that already utilize the RTA (RTNN and WKND_PT), the additional OP unit energy consumption of TTA+ can be offset by applying optimizations (*RTNN and *WKND_PT), still achieving 19-29% energy savings.

## VI. Related Work

### A. Ray-Tracing Acceleration

Compared to rasterization [75] and neural rendering techniques [47], [59], [62], [93], ray tracing is highly divergent and memory latency-bound, making it challenging to accelerate on SIMT architectures. Despite this, many studies still focus on ray tracing for its ability to produce high-quality images. Pharr et al. [74] proposed to dynamically reorder rays based on the contents of the cache, but ray sorting is considered prohibitively expensive for real-time rendering [58]. Barringer et al. [11] introduced a traversal algorithm that leverages ray coherence for high SIMD efficiency, but packet tracing is ineffective for incoherent rays. Shkurko et al. [85] proposed a dual streaming approach that organizes ray-tracing memory accesses into two predictable data streams, and Vasiou et al. [90] analyzed the energy and time cost of data movement for ray tracing. These optimizations can be applied orthogonally to our work. While these studies aim to improve ray-tracing performance, they do not fully address the fundamental limitations of the SIMT pipeline. In contrast, RTAs are introduced to address these limitations by providing a dedicated hardware accelerator specifically for ray tracing. Our work focuses on generalizing RTAs for diverse tree traversal algorithms, enabling more applications to be accelerated.

### B. Tree Traversal Acceleration

Prior works have explored accelerating tree-based applications on FPGAs and ASICs, such as k-nearest neighbor search [56], [86], dynamic search tree [98], and ray tracing [48], [66]. However, these works are limited to specific applications

and require significant hardware modifications to support new applications. TTA's and TTA+'s programming model shine in their simplicity compared to FPGAs and CGRAs, requiring only writing simple programs and specifying data and node layouts, while FPGAs and CGRAs require register-transfer-level (RTL) design knowledge and orchestrating explicit dataflow programs via interconnections [52]. Furthermore, RTAs are already integrated into modern GPUs [1], [13], [78], making them more accessible than other reconfigurable accelerators. For example, programmers can leverage the rich ecosystem of CUDA or Vulkan APIs to program RTAs [44], [72], which is not possible with FPGAs and CGRAs.

### C. Repurposing RTAs without Hardware Modification

RTAs can be repurposed for other applications without hardware modifications. RTNN [105] and RT-kNNS Unbound [64] explore accelerating k-nearest neighbor search by mapping the distance calculation between two points to the ray-tracing pipeline. RTIndeX [34] leverages RTAs for efficient database indexing, while RT-DBSCAN [63] focuses on enhancing DBSCAN clustering algorithms. Additionally, in the prior work [61], [92], authors extend RTAs to compute mesh point location. However, as mentioned in Section I, mapping applications onto the existing ray-tracing pipeline without hardware modifications presents challenges due to the added complexity of the graphics pipeline. Although the Vulkan API provides a `rayQuery` extension to support isolated ray traversals without the ray-tracing pipeline, it is still limited to BVH traversal with only ray-box and ray-triangle intersection tests. Thus, the adapted algorithms frequently incur substantial overhead from a suboptimal fit between the graphics pipeline and algorithms. In contrast, our proposal demonstrates the feasibility of using RTAs in diverse tree search applications with huge programmability, showing large performance gains in various applications.

### D. Graph Analysis Acceleration

Algorithms for determining shortest paths, like Dijkstra's and Bellman-Ford's, can be accelerated through matrix multiplication accelerations [103]. In this respect, model dynamics in social networks and opinion dynamics can be accelerated by matrix multiplication [31], [32], [73], [87], [89]. However, these works are limited to graph analysis applications, and they introduce inherent limitations in time and space complexity. Unlike these approaches, our proposal optimizes tree traversal applications, which inherently require less computational resources, providing a more efficient solution from a hardware perspective.

## VII. Conclusion

RTAs have enhanced the efficiency and performance of ray tracing, addressing key challenges of tree traversal on GPUs. This paper underscores that other tree-based applications face issues similar to ray tracing and potentially benefit from RTAs. However, mapping non-graphics applications to the ray-tracing pipelines can be difficult due to limited programmability, which leads to resource underutilization. To address these challenges, we propose two innovative solutions: (1) TTA, which adapts existing units within the ray-tracing pipeline for more diverse applications, and (2) TTA+, which focuses on increasing the flexibility of the computing units. In our evaluation, we achieve up to 5.4× speedup for tree-traversal applications and up to 62% energy reduction with less than 1% area overhead.


## Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was sponsored by two National Science Foundation (NSF) awards, CNS-2007124 and CNS-2231877. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Lastly, this work was supported by Y-BASE R&E Institute, a Brain Korea 21 four program at Yonsei University.


## Appendix

### A. Abstract

This artifact includes our modified implementation of Vulkan-Sim that models our proposed TTA and TTA+ architectures as well as binaries for the benchmark workloads used in our evaluation. We provide this artifact in a Docker container, which includes all necessary dependencies and scripts to reproduce our main results.

### B. Artifact check-list (meta-information)

- **Program:** modified Vulkan-Sim; B-Tree, N-body, RTNN, and LumiBench with WKND_PT benchmarks
- **Compilation:** gcc/g++, ninja, meson, cmake, nvcc
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** >12 GB RAM
- **Metrics:** execution time, energy consumption, other detailed simulation statistics
- **Output:** simulation statistics and result figures
- **How much disk space required (approximately)?:** 30GB (Docker image)
- **How much time is needed to prepare workflow (approximately)?:** < 1 hour
- **How much time is needed to complete experiments (approximately)?:** 3-5 days if executed serially
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.13294690

### C. Description

*1) How to access:* We package our artifact in a Docker container, which can be downloaded from Zenodo at the archived link (https://doi.org/10.5281/zenodo.13294690). Please select the latest version of the artifact.

*2) Hardware dependencies:* There are no specific hardware requirements. We recommend at least 12 GB of RAM to run the experiments.

*3) Software dependencies:* Docker is required to run the provided container. All other dependencies are included in the container.

### D. Installation

Download our Docker container from the archived link. To run the container, execute the following command:

```
docker load < tta-artifact.tar.gz
docker run -it tta:artifact /bin/bash
```

### E. Experiment workflow

The general workflow is to run a set of benchmarks using Vulkan-Sim, process the results, and generate figures based on the results. Once inside the container, experiments can be started by sourcing the setup script for Vulkan-Sim and running the provided Python automation scripts. The parameter to the Python script controls how many benchmarks are executed in parallel, which can be adjusted based on the available resources.

```
cd /home/vulkansim/gpgpu-sim_emerald
source setup_environment
cd /home
python3 run_sweep_full.py 8
```

After the experiments are complete, the results can be generated by running the following script, which summarizes all the results in a single CSV file.

```
python3 process_results.py
```

Using the processed results, the following figures in the paper can be generated using the provided Python scripts.

- Figure 12: `plot_speedup.py`
- Figure 13: `plot_dram.py`
- Figure 19: `plot_energy.py`
- Figure 20: `plot_insn_breakdown.py`

To produce additional figures, such as our LumiBench results, detailed utilization breakdowns, and sensitivity analysis, please refer to the `README` file in the container for instructions.

### F. Evaluation and expected results

After running the experiments and the processing script, results will be organized in the `all_stats.csv` file, which contains all the simulation statistics. Separate PNG files will be generated using the Python scripts for each figure. Note that results may vary slightly between different systems, but the general trends should match the figures in the paper.

### G. Experiment customization

Other benchmark configurations can be executed manually by using the provided binaries and command line arguments. For example, the B-Tree benchmark can be run with the following format:

```
./rtbtree [tree_size] [n_queries] \
  [random_seed] [tree_type]
```

The Vulkan-Sim configuration files can be modified in the respective benchmark directories to reflect other architectural configurations.

### H. Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

## References


[1] Advanced Micro Devices Inc. (2022) AMD RDNA architecture. [Online]. Available: https://www.amd.com/en/technologies/rdna

[2] Agner Fog. (2022) Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[3] M. K. Aguilera, W. Golab, and M. A. Shah, "A practical scalable distributed B-tree," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 598–609, 2008.

[4] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2009, pp. 145–149.

[5] A. Alfarrarjeh, S. H. Kim, V. Hegde, C. Shahabi, Q. Xie, S. Ravada *et al.*, "A class of R*-tree indexes for spatial-visual search of geo-tagged street images," in *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, 2020, pp. 1990–1993.

[6] Apple. (2023) Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer. [Online]. Available: https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/

[7] M. A. Awad, S. Ashkiani, R. Johnson, M. Farach-Colton, and J. D. Owens, "Engineering a high-performance GPU B-Tree," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2019, pp. 145–157.

[8] M. A. Awad, S. D. Porumbescu, and J. D. Owens, "A GPU multiversion B-Tree," in *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*, 2023, pp. 481–493.

[9] J. S. Bagla, "Cosmological N-body simulation: Techniques, scope and status," *Current Science*, pp. 1088–1100, 2005.

[10] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[11] R. Barringer and T. Akenine-Möller, "Dynamic ray stream traversal," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–9, 2014.

[12] J. Bédorf, E. Gaburov, and S. P. Zwart, "A sparse octree gravitational N-body code that runs entirely on the GPU processor," *Journal of Computational Physics*, vol. 231, no. 7, pp. 2825–2839, 2012.

[13] J. Burgess, "RTX on—the NVIDIA Turing GPU," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020.

[14] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based Barnes-Hut N-body algorithm," in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 75–92.

[15] J. Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, 2023.

[16] Y. H. Chou, T. Nowicki, and T. M. Aamodt, "Treelet prefetching for ray tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2023.

[17] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[18] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: Exposing the value of flexibility for graph processing accelerators," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2021, pp. 595–608.

[19] Z. Deng, L. Wang, W. Han, R. Ranjan, and A. Zomaya, "G-ML-Octree: An update-efficient index structure for simulating 3D moving objects across GPUs," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 5, pp. 1075–1088, 2017.

[20] H. Dong, C. Chai, Y. Luo, J. Liu, J. Feng, and C. Zhan, "RW-Tree: A learned workload-aware framework for R-tree construction," in *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, 2022, pp. 2073–2085.

[21] S. S. Eusuf, K. A. Islam, M. E. Ali, S. M. Abdullah, and A. S. Azad, "A web-based system for efficient contact tracing query in a large spatio-temporal database," in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, 2020, pp. 473–476.

[22] A. J. Gallego, J. R. Rico-Juan, and J. J. Valero-Mas, "Efficient k-nearest neighbor search based on clustering and adaptive k values," *Pattern recognition*, vol. 122, p. 108356, 2022.

[23] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the KITTI vision benchmark suite," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 3354–3361.

[24] A. S. Glassner, *An introduction to ray tracing*. Morgan Kaufmann, 1989.

[25] G. Graefe, "A survey of B-tree logging and recovery techniques," *ACM Transactions on Database Systems (TODS)*, vol. 37, no. 1, pp. 1–35, 2012.

[26] T. Gu, K. Feng, G. Cong, C. Long, Z. Wang, and S. Wang, "The RLR-Tree: A reinforcement learning based R-tree for spatial data," *ACM SIGMOD Record*, vol. 1, no. 1, pp. 1–26, 2023.

[27] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. Int'l Conf. on Management of Data*, 1984, pp. 47–57.

[28] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2016.

[29] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2010, pp. 37–47.

[30] K. He and J. Sun, "Computing nearest-neighbor fields via propagation-assisted kd-trees," in *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2012, pp. 111–118.

[31] X. He, K.-Y. Chen, S. Feng, H.-S. Kim, D. Blaauw, R. Dreslinski, and T. Mudge, "Squaring the circle: Executing sparse matrix computations on FlexTPU—a TPU-like processor," in *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*, 2022, pp. 148–159.

[32] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-TPU: Adapting systolic arrays for sparse matrices," in *Proc. ACM Conf. on Supercomputing (ICS)*, 2020.

[33] N. Hegde, J. Liu, K. Sundararajah, and M. Kulkarni, "Treelogy: A benchmark suite for tree traversals," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2017, pp. 227–238.

[34] J. Henneberg and F. Schuhknecht, "RTIndeX: Exploiting hardware-accelerated GPU raytracing for database indexing," *arXiv preprint arXiv:2303.01139*, 2023.

[35] L. Hu, S. Nooshabadi, and M. Ahmadi, "Massively parallel KD-tree construction and nearest neighbor search algorithms," in *IEEE Int'l Symp. on Circuits and Systems (ISCAS)*, 2015, pp. 2752–2755.

[36] Y.-C. Hu, Y. Li, and H.-W. Tseng, "TCUDB: Accelerating database with tensor processors," in *Proc. Int'l Conf. on Management of Data*, 2022, pp. 1360–1374.

[37] Imagination Technologies. (2022) Rays your game: Introduction to the PowerVR photon architecture. [Online]. Available: https://www.imaginationtech.com/resources/rays-your-game-en/

[38] Intel Corporation. (2002) Verilog HDL: 16x16 Crosspoint Switch. [Online]. Available: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/ver-16x16.html

[39] Intel Corporation. (2022) Introduction to the Xe-HPG architecture. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-the-xe-hpg-architecture.html

[40] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient B+-tree based indexing of moving objects," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30*, 2004, pp. 768–779.

[41] T. Johnson and D. Sasha, "The performance of current B-tree algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 18, no. 1, pp. 51–101, 1993.

[42] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018.

[43] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 738–753.

[44] Khronos Group. (2023) Vulkan. [Online]. Available: https://www.vulkan.org/

[45] A. Klypin, S. Gottlöber, A. V. Kravtsov, and A. M. Khokhlov, "Galaxies in N-body simulations: overcoming the overmerging problem," *The Astrophysical Journal*, vol. 516, no. 2, p. 530, 1999.

[46] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, 2006, pp. 21–30.

[47] J. Lee, K. Choi, J. Lee, S. Lee, J. Whangbo, and J. Sim, "NeuRex: A case for neural rendering acceleration," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2023.

[48] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "SGRT: A mobile GPU architecture for real-time ray tracing," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2013, pp. 109–119.

[49] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," in *Proc. IEEE Int'l Conf. on Data Engineering (ICDE)*, 2013, pp. 302–313.

[50] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[51] J. Liu, M. Robson, T. Quinn, and M. Kulkarni, "Efficient GPU tree walks for effective distributed N-body simulations," in *Proc. ACM Conf. on Supercomputing (ICS)*, 2019, pp. 24–34.

[52] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Computing Surveys (CSUR)*, 2019.

[53] 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 *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 709–723.

[54] L. Liu, M. Saed, Y. H. Chou, D. Grigoryan, T. Nowicki, and T. M. Aamodt, "LumiBench: A benchmark suite for hardware ray tracing," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2023.

[55] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh, "An analysis of database workload performance on simultaneous multithreaded processors," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 39–50, 1998.

[56] A. Lu, Z. Fang, N. Farahpour, and L. Shannon, "CHIP-KNN: A configurable and high-performance k-nearest neighbors accelerator on cloud FPGAs," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 139–147.

[57] Y. Lü, L. Huang, L. Shen, and Z. Wang, "Unleashing the power of GPU for physically-based rendering via dynamic ray shuffling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2017, pp. 560–573.

[58] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, "On ray reordering techniques for faster GPU ray tracing," in *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, 2020.

[59] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing scenes as neural radiance fields for view synthesis," vol. 65, no. 1, pp. 99–106, 2021.

[60] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH Courses*, 2005.

[61] N. Morrical, I. Wald, W. Usher, and V. Pascucci, "Accelerating unstructured mesh point location with RT Cores," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 28, no. 8, pp. 2852–2866, 2022.

[62] M. H. Mubarik, R. Kanungo, T. Zirr, and R. Kumar, "Hardware acceleration of neural graphics," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2023.

[63] V. Nagarajan and M. Kulkarni, "RT-DBSCAN: Accelerating DBSCAN using ray tracing hardware," *arXiv preprint arXiv:2303.09655*, 2023.

[64] V. Nagarajan, D. Mandarapu, and M. Kulkarni, "RT-KNNS unbound: Using RT Cores to accelerate unrestricted neighbor search," in *Proc. ACM Conf. on Supercomputing (ICS)*, 2023, pp. 289–300.

[65] J.-H. Nah and D. Manocha, "SATO: Surface area traversal order for shadow ray tracing," in *Computer Graphics Forum*, vol. 33, no. 6, 2014, pp. 167–177.

[66] 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," in *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*, 2011.

[67] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2017, pp. 457–468.

[68] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash *et al.*,

"The Niagara internet query system," *IEEE Data Eng. Bull.*, vol. 24, no. 2, pp. 27–33, 2001.

[69] NVIDIA, "Nvidia tesla v100 gpu architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[70] Nvidia Corporation. (2009) NVIDIA's Next Generation CUDA Compute Architecture: Fermi. [Online]. Available: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

[71] Nvidia Corporation. (2018) Nvidia Turing GPU architecture. [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[72] Nvidia Corporation. (2019) How to get started with optix 7. [Online]. Available: https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/

[73] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2018, pp. 724–736.

[74] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1997, pp. 101–108.

[75] J. Pineda, "A parallel algorithm for polygon rasterization," in *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1988, pp. 17–20.

[76] R. Pinkham, S. Zeng, and Z. Zhang, "QuickNN: Memory and performance optimization of kd tree based nearest neighbor search for 3D point clouds," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2020, pp. 180–192.

[77] J. Qi, Y. Tao, Y. Chang, and R. Zhang, "Packing R-trees with space-filling curves: Theoretical optimality, empirical efficiency, and bulk-loading parallelizability," *ACM Transactions on Database Systems (TODS)*, vol. 45, no. 3, pp. 1–47, 2020.

[78] Qualcomm. (2022) New, Snapdragon 8 Gen 2: 8 extraordinary mobile experiences, unveiled. [Online]. Available: https://www.qualcomm.com/news/onq/2022/11/new-snapdragon-8-gen-2-8-extraordinary-mobile-experiences-unveiled

[79] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "GraphPulse: An event-driven hardware accelerator for asynchronous graph processing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2020, pp. 908–921.

[80] P. Ram and K. Sinha, "Revisiting kd-tree for nearest neighbor search," in *Proc. ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, 2019, pp. 1378–1388.

[81] Y. Rayhan and W. G. Aref, "SIMD-ified R-tree query processing and optimization," in *Proceedings of the 31st ACM International Conference on Advances in Geographic Information Systems*, 2023, pp. 1–10.

[82] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The linux B-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[83] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-Sim: A GPU architecture simulator for ray tracing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2022, pp. 263–281.

[84] A. Shahvarani and H.-A. Jacobsen, "A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms," in *ACM SIGMOD Record*, 2016, pp. 1523–1538.

[85] K. Shkurko, T. Grant, D. Kopta, I. Mallett, C. Yuksel, and E. Brunvand, "Dual streaming for hardware-accelerated ray tracing," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2017.

[86] X. Song, T. Xie, and S. Fischer, "A memory-access-efficient adaptive implementation of kNN on FPGA through HLS," in *Proc. IEEE Conf. on Computer Design (ICCD)*. IEEE, 2019, pp. 177–180.

[89] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, 2015.

[87] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2020, pp. 766–780.

[88] Stanford Computer Graphics Laboratory. (1994) The Stanford 3D Scanning Repository. [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/

[90] E. Vasiou, K. Shkurko, I. Mallett, E. Brunvand, and C. Yuksel, "A detailed study of ray tracing performance: render time and energy cost," *The Visual Computer*, vol. 34, pp. 875–885, 2018.

[91] I. Wald, N. Morrical, S. Zellmann, L. Ma, W. Usher, T. Huang, and V. Pascucci, "Using hardware ray transforms to accelerate ray/primitive intersections for long, thin primitive types," *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, vol. 3, no. 2, 2020.

[92] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2019, pp. 7–13.

[93] Z. Wang, S. Wu, W. Xie, M. Chen, and V. A. Prisacariu, "NeRF–: Neural radiance fields without known camera parameters," *arXiv preprint arXiv:2102.07064*, 2021.

[94] S. Woop, C. Benthin, I. Wald, G. S. Johnson, and E. Tabellion, "Exploiting local orientation similarity for efficient ray traversal of hair and fur," *Proc. ACM Conf. on High Performance Graphics (HPG)*, vol. 3, 2014.

[95] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu, "Efficient B-tree based indexing for cloud data processing," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1207–1218, 2010.

[96] Z. Yan, Y. Lin, L. Peng, and W. Zhang, "Harmonia: a high throughput B+ tree for GPUs," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2019, pp. 133–144.

[97] J. Yang and G. Cong, "PLATON: Top-down R-tree packing with learned partition policy," *ACM SIGMOD Record*, vol. 1, no. 4, pp. 1–26, 2023.

[98] Y.-H. E. Yang and V. K. Prasanna, "High throughput and large capacity pipelined dynamic search tree on FPGA," in *Proc. ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, 2010, pp. 83–92.

[99] A. Yoshimura and T. Harada, "Subspace culling for ray-box intersection," in *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, 2023.

[100] F. Zhang, P. Di, H. Zhou, X. Liao, and J. Xue, "RegTT: Accelerating tree traversals on GPUs by exploiting regularities," in *Proc. Int'l Conf. on Parallel Processing (ICPP)*, 2016, pp. 562–571.

[101] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2018, pp. 544–557.

[102] W. Zhang, C. Zhao, L. Peng, Y. Lin, F. Zhang, and J. Jiang, "High performance GPU concurrent B+ tree," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2022, pp. 443–444.

[103] Y. Zhang, P.-A. Tsai, and H.-W. Tseng, "SIMD2: A generalized matrix instruction set for accelerating tensor computation beyond GEMM," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2022, pp. 552–566.

[104] Y. Zhao, Y. Wang, J. Zhang, C.-W. Fu, M. Xu, and D. Moritz, "KD-Box: Line-segment-based KD-tree for interactive exploration of large-scale time-series data," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 28, no. 1, pp. 890–900, 2021.

[105] Y. Zhu, "RTNN: accelerating neighbor search using hardware ray tracing," in *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, 2022, pp. 76–89.

[106] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "GraphQ: Scalable PIM-based graph processing," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2019, pp. 712–725.