AQB8: Energy-Efficient Ray Tracing Accelerator through Multi-Level Quantization

Yen-Chieh Huang
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
lashhw.cs09@nycu.edu.tw

Chen-Pin Yang
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
ycpin.cs12@nycu.edu.tw

Tsung Tai Yeh
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
ttyeh@cs.nycu.edu.tw

Abstract

Ray tracing (RT) is a rendering technique that produces high-fidelity images by simulating how light physically interacts with objects in a scene. This realism comes at a high computational and memory cost, largely driven by the need to find numerous ray-object intersections. To accelerate this process, scenes are typically structured using bounding volume hierarchy (BVH) trees, where objects are grouped within bounding boxes to minimize the number of required intersection tests. Specialized hardware, known as RT accelerators, accelerates BVH processing to boost computational speed, yet memory traffic persists as a major bottleneck, largely due to the bandwidth consumed by standard 32-bit floating-point (FP32) bounding boxes. Consequently, previous work has aimed to reduce memory traffic by compressing these boxes into low-bit (e.g., 8-bit) representations. However, existing compression techniques typically require decompressing bounding boxes back to FP32 for intersection tests, thus failing to eliminate the computational burden and energy cost associated with complex FP32 arithmetic during BVH traversal. This work introduces *AQB8*, an RT accelerator designed to operate on a *quantized BVH tree* constructed using a novel *multi-level quantization* technique. This approach enables RT to operate *directly* on low-bit integers using simpler, area-efficient hardware units, thereby drastically reducing the need for FP32 arithmetic during BVH traversal while mitigating overheads associated with reduced precision. As a result, across various scenes, AQB8 achieves a 70% reduction in DRAM accesses, a 49% reduction in energy consumption, a 27% hardware area reduction, and a 1.82x performance speedup over modern GPU RT accelerators. Our code is publicly available at https://github.com/nycu-caslab/AQB8.

CCS Concepts

• **Computing methodologies** → **Ray tracing**; **Graphics processors**; **Modeling and simulation**.

Keywords

Ray Tracing, Domain-Specific Accelerator, GPU

ISCA '25, June 21–25, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/2025/06
https://doi.org/10.1145/3695053.3731104

ACM Reference Format:
Yen-Chieh Huang, Chen-Pin Yang, and Tsung Tai Yeh. 2025. AQB8: Energy-Efficient Ray Tracing Accelerator through Multi-Level Quantization. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3695053.3731104

1 Introduction

Ray tracing (RT) is a computer graphics rendering technique that simulates the physical behavior of light as it interacts with objects in a scene to generate high-fidelity images. Although images generated through RT are intricately detailed, its adoption in computer graphics applications such as video gaming and film production was historically restricted due to its high computational demand. Modern GPUs, such as those made by NVIDIA [50], AMD [1], and Intel [21], have addressed this issue by incorporating dedicated RT hardware accelerators. These accelerators enable real-time RT, improving the visual effects of computer graphics applications.

RT functions by sending out rays (e.g., from a camera viewpoint) that then intersect with objects in the scene (termed *ray-object intersections*), aiming to determine visibility and simulate light transport. Since generating a detailed image requires tracing numerous rays per pixel, managing the cost of potentially billions of these intersection tests is crucial. A key technique used by accelerators to optimize this process involves structuring the scene geometry using a *bounding volume hierarchy* (BVH) tree. Since complex scenes are often represented by millions of individual triangles, BVHs work by grouping these triangles within nested *bounding boxes* (BBs). Instead of testing intersections against every triangle, rays first test against these BBs (termed *ray-box intersection tests*), allowing most of the triangles to be quickly skipped. This drastically reduces the number of expensive ray-triangle intersection tests required.

However, while accelerators improve throughput, significant efficiency challenges remain, stemming from how BVH trees are conventionally implemented. Usually, the BBs and ray-box intersection tests rely on the standard 32-bit floating-point (FP32) data format for accuracy. While precise, representing BBs in FP32 format leads to significant memory traffic due to the large data size. Moreover, the computations performed during BVH traversal require intricate FP arithmetic hardware units, which consume considerable energy and contribute to hardware complexity. These factors combine to limit the overall system efficiency and energy savings, even with hardware acceleration.

Recognizing the memory issue caused by FP32 BBs, past research has explored compressing BBs into low-bit data formats [7, 23, 26, 32, 37, 69, 77]. However, these methods typically require converting the compressed BBs back into the FP32 format for RT accelerators

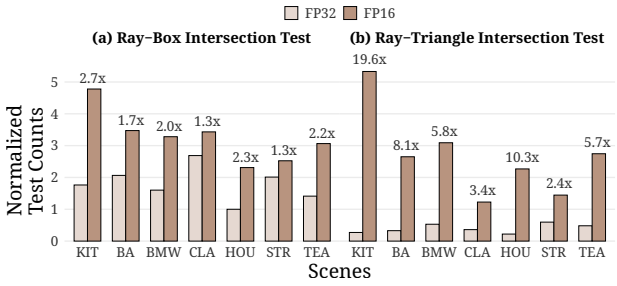

Figure 1: Impact of reduced-precision BVH traversal.

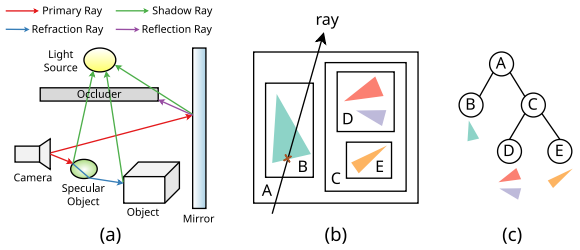

Figure 2: (a) Ray tracing. (b) Bounding boxes. (c) BVH tree.

to perform ray-box intersection tests. Consequently, while memory traffic might decrease, the computational workload during BVH traversal still relies heavily on energy-intensive FP32 hardware units. This motivates exploring whether BVH traversal can be performed *using low-bit arithmetic directly*, thereby eliminating the reliance on FP32 computations. To investigate the viability of this approach, we simulated a scenario where both the BBs were represented and the ray-box intersection tests were calculated using 16-bit floating-point (FP16), instead of the standard FP32 format. As shown in Figure 1 (obtained by inserting counters into the RT program across different scenes), this direct use of FP16 arithmetic for BVH traversal can cause a 2.7x increase in ray-box intersection tests and a staggering 19.6x increase in ray-triangle intersection tests compared to standard FP32 BVH traversal. This occurs because the reduced precision alters the effective size and position of BBs (an effect referred to as *quantization errors*), leading to substantially more unnecessary tree traversal steps. This demonstrates that naively using low-bit formats for BVH traversal is infeasible.

The main challenge, therefore, is to leverage the efficiency benefits of low-bit BB while minimizing the penalty of increased BVH traversal steps. To address this challenge, this work proposes *multi-level quantization*, a novel technique for constructing a *quantized BVH tree*. This technique structures the BVH tree into *clusters*, each containing a high-precision FP32 *anchor BB*. Other bounding boxes within the cluster, termed as *quantized BBs*, are then represented using low-bit integers relative to this anchor. This organization allows for compact data representation and direct operation using simpler integer (INT) arithmetic, while the relative encoding scheme anchored to a high-precision reference mitigates the traversal step increase typically caused by low-bit quantization errors. By enabling the direct use of simpler INT operations, this approach provides a foundation for building more efficient RT accelerators.

Specifically, we present *AQB8* (Accelerator for Quantized BVH with 8-bit Traversal), an RT accelerator designed to process the quantized BVH tree. In AQB8, anchor BBs are maintained at full FP32 precision, while quantized BBs are encoded as 8-bit integers (INT8). This design allows AQB8 to perform a large portion of the BVH traversal using simple INT8 arithmetic, minimizing reliance on complex FP32 units and avoiding frequent format conversions during traversal. As a result, AQB8 achieves significant improvements in terms of memory traffic, energy consumption, hardware area, and performance speedup. The contributions of this work are summarized below.

- We propose a novel multi-level quantization technique that creates a quantized BVH tree to enable using low-bit INT arithmetic during BVH traversal.
- We present AQB8, an efficient RT accelerator for GPUs specifically designed to process quantized BVH trees.
- We demonstrate that AQB8 reduces memory traffic by 70%, energy consumption by 49%, and hardware area by 27%, which collectively yield a geometric mean of 1.82x speedup in realistic RT workloads.

2 Background

2.1 Ray Tracing & BVH Tree

2.1.1 Ray Tracing. RT generates 3D graphics by simulating the interaction of light rays with objects in a scene, which are typically represented as *triangle meshes*—collections of interconnected triangles that approximate 3D surfaces. As illustrated in Figure 2(a), primary rays emitted from a camera intersect objects in the scene, spawning secondary rays to simulate effects such as reflection, refraction, and shadows. To enhance graphic rendering quality, RT emits multiple rays per image pixel, sampling more light sources and producing realistic visual effects. However, processing such a large number of rays is challenging because it requires performing an enormous number of ray-triangle intersection tests. To address this, RT utilizes a BVH tree [40], which organizes the triangles in a scene into nested bounding boxes (BBs) (Figure 2(b)(c)). This hierarchical structure allows rays to bypass intersection tests with triangles in regions they do not intersect, significantly improving computational efficiency.

2.1.2 BVH Traversal. The BVH tree organizes scene geometry into a tree structure for efficient RT. As illustrated in Figure 2(b), the scene's triangles are grouped into BBs, with each BB representing a spatial region. Figure 2(c) demonstrates the corresponding BVH tree, where each node contains a BB, and the leaf nodes store the triangles enclosed within their respective BBs.

RT begins by testing whether an emitted ray intersects the BB at the root of the BVH tree. A ray then continues to visit child nodes of the BVH tree until it reaches a leaf node containing triangles. At the leaf node, ray-triangle intersection tests are performed to determine the intersection point on the triangle. Structuring RT as BVH traversal significantly improves RT performance by avoiding the need to examine all triangles in a scene. For example, in Figure 2(b), the ray only intersects bounding boxes A and B, and hence does not pass node C and its descendants.

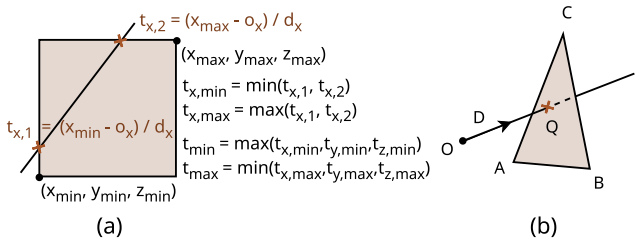

Figure 3: (a) Ray-box and (b) ray-triangle intersection test.

2.1.3 Ray-Box Intersection Test. The ray-box intersection test is a fundamental step while traversing BVH trees, where RT checks if a ray intersects the BB associated with a node. This test determines whether to traverse deeper into the node's children or skip the node entirely. As shown in Figure 3(a), a BB is defined by two corner points: $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$. A ray is represented parametrically as:

$$\mathbf{p}(t) = \mathbf{o} + t\,\mathbf{d},$$

where $\mathbf{o} = (o_x, o_y, o_z)^T$ is the ray's origin, $\mathbf{d} = (d_x, d_y, d_z)^T$ is its direction, and t is a scalar parameter that specifies the distance along the ray. For any t, $\mathbf{p}(t)$ yields the coordinates of the corresponding point along the ray. To find the parameter $t_{x'}$ where the ray intersects a plane $x = x'$, the following equation is used:

$$t_{x'} = (x' - o_x)/d_x$$

This can be reformulated for computational efficiency by introducing:

$$\mathbf{w} = (w_x, w_y, w_z)^T = (1/d_x, 1/d_y, 1/d_z)^T$$
$$\mathbf{b} = (b_x, b_y, b_z)^T = (-o_x/d_x, -o_y/d_y, -o_z/d_z)^T$$

where $\mathbf{w}$ and $\mathbf{b}$ are precomputed values based on the ray's direction and origin. Preprocessing rays into $\mathbf{w}$ and $\mathbf{b}$ eliminates the need for repeated division operations during traversal, which are computationally expensive. By replacing divisions with multiplications and additions, the ray-box intersection tests can be performed more efficiently. This yields:

$$t_{x'} = w_x x' + b_x \quad (1)$$

Using Equation 1 as the building block, we can find where the ray intersects a BB based on the Slab method [22] as shown in Figure 3(a). If we end up with $t_{max} > t_{min}$, there is an intersection between a ray and a BB, and vice versa.

2.1.4 Ray-Triangle Intersection Test. At the leaf nodes of the BVH tree, RT performs ray-triangle intersection tests to determine if the ray intersects any of the triangles within the node and calculates the intersection points if they exist. As shown in Figure 3(b), a ray originating at O with direction D intersects a triangle defined by vertices A, B, and C at point Q.

2.2 GPU RT Accelerator Architecture

Modern GPUs feature various dedicated hardware accelerators [50], such as tensor and RT accelerators within each streaming multiprocessor (SM), as illustrated in Figure 4. The RT accelerator typically consists of tree traversal (TRV) units, ray-box intersection (BOX) units, and ray-triangle intersection (TRIG) units [11], along with a small on-chip L0 cache that stores frequently accessed BVH tree nodes and triangles. The TRV unit handles the BVH traversal logic, maintaining stack memory to store ray-intersected nodes that have yet to be visited. The BOX unit performs ray-box intersection tests to check whether a ray intersects a BB. The TRIG unit performs ray-triangle intersection tests for triangles located in the leaf nodes, calculating exact intersection points for graphic rendering.

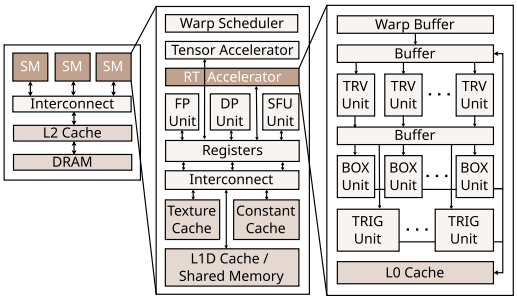

Figure 4: GPU RT accelerator architecture [56].

Programmers can use the Vulkan ray tracing API [67], which translates high-level ray tracing calls into specialized instructions executed by the RT accelerator. When a warp issues a ray tracing instruction, the GPU warp scheduler dispatches the instruction to the warp buffer in the RT accelerator during the GPU pipeline's execution stage. Each cycle, the RT accelerator selects a warp from the buffer to perform ray tracing operations and issues memory requests to its L1 memory access queue [10, 56].

2.3 RT Memory Traffic Breakdown

RT generates a significant amount of memory traffic during rendering. This is because high-quality rendering requires many rays per pixel, and each ray often makes frequent, irregular memory requests as it traverses the BVH tree [10]. For instance, profiling indicates trace ray instructions can account for roughly 60% of memory accesses [56], making RT workloads frequently memory-bound [34, 56]. Consequently, RT performance often becomes limited by memory bandwidth, as the sheer volume and irregular pattern of data requests can overwhelm the memory system, even with specialized RT accelerator units designed to speed up computations. This bottleneck is particularly acute in mobile GPU environments where DRAM bandwidth is typically more constrained [56].

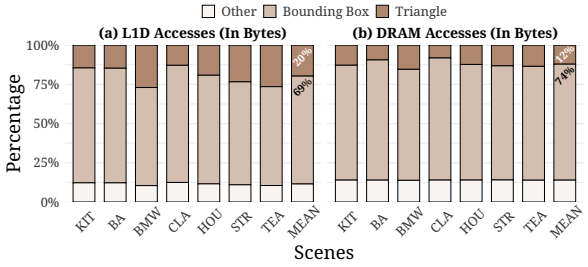

Figure 5: RT accelerator memory traffic breakdown.

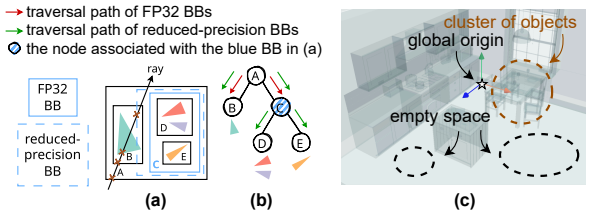

Figure 6: (a) Illustration of quantization error in BBs. (b) Increased traversal steps due to quantization error. (c) Sparsity of a scene.

Figure 5 illustrates the breakdown of both L1D cache accesses and DRAM accesses for triangles, BBs, and other data (such as child indices and metadata) across various scenes, based on statistics collected using Vulkan-Sim [56]. As shown, BB data constitutes the largest portion of memory traffic. It accounts for an average of 69% of L1D cache accesses, primarily because each ray typically intersects multiple BBs while traversing the BVH tree before reaching a leaf node. For DRAM accesses, BB data remains dominant, representing 74% of the total traffic. This breakdown clearly indicates that BBs are the primary contributor to the memory traffic of RT.

3 Multi-Level Quantization

As established in Section 2.3, bounding boxes (BBs) represented in the standard FP32 format are a primary source of memory traffic in RT. Furthermore, conventional ray-box intersection tests rely on complex and energy-intensive FP32 arithmetic units. Reducing the bitwidth of BBs appears promising for alleviating both memory and computational costs.

However, naive precision reduction introduces significant challenges. Simply using lower precision formats like FP16 for both storage and computation leads to unacceptable performance degradation. As demonstrated in Figure 1, this approach introduces severe *quantization errors*—inaccuracies stemming from the limited precision. When quantizing FP32 BBs to a lower precision format like FP16, the necessary rounding or truncation must be performed carefully. Crucially, to ensure rendering correctness, the resulting quantized BB must fully enclose the original FP32 BB. If the quantized BB were smaller, a ray might intersect the original BB but miss the quantized one, leading to incorrect rendering. Therefore, the conversion process often slightly shifts the minimum and maximum bounds outwards. This conservative enclosing effectively alters the BB's size and position (illustrated in Figure 6(a)). While guaranteeing correctness, this potential expansion leads to numerous *false positive intersections* (rays incorrectly reported as intersecting the reduced-precision BB when they would miss the original FP32 BB). Such false positives increase unnecessary BVH traversal steps (Figure 6(b)), ultimately hindering performance. Moreover, existing BB compression techniques [7, 23, 32, 37, 69, 77] also fall short, often requiring decompression back to FP32 for ray-box intersection tests, thus failing to eliminate the reliance on complex FP32 arithmetic during BVH traversal. Therefore, a solution is needed that not only compresses BB but also leverages simpler low-bit arithmetic to truly harness efficiency benefits without prohibitive overheads.

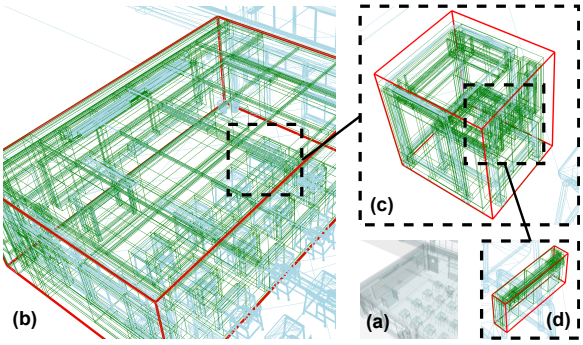

Figure 7: Visualization of multi-level quantization. (a) Overview of the 3D scene. (b) Quantized BBs (green) shown within a large anchor BB (red), forming a coarse-level cluster. (c) A smaller anchor BB (red) establishing a finer-detail cluster. (d) An even smaller anchor BB (red) establishing a very fine-detail cluster.

We observe that typical computer graphics scenes exhibit *sparsity*, often containing large volumes of *empty space* interspersed with *localized clusters of objects* (Figure 6(c)). Standard world-space coordinates, defined relative to a single *global* origin, do not inherently exploit this characteristic. However, representing geometry relative to *local* reference points within these clusters offers a path to efficiency.

Motivated by this observation, we propose *multi-level quantization*. This technique restructures the BVH tree to enable compact BB representation while strategically mitigating quantization errors. As illustrated in Figure 7, multi-level quantization organizes BVH nodes into *clusters*, each comprising:

- An *anchor BB* (shown in red): Defined using high-precision (FP32) world-space coordinates, establishing an accurate local reference frame for the cluster.
- Several *quantized BBs* (shown in green): Defined using compact, low-bit integer formats (e.g., INT8) within the local coordinate system defined by the anchor BB.

During BVH traversal, when a ray enters a cluster (i.e., intersects the FP32 anchor BB), subsequent ray-box tests against the quantized BBs within that cluster operate *directly* on their low-bit integer representations within the anchor's local coordinate frame. This avoids decompression and eliminates the need for complex FP32 arithmetic for a significant portion of the traversal process.

Furthermore, the "multi-level" aspect strategically reduces quantization error. Anchor BBs are used throughout the BVH tree *across multiple levels* (Figure 7). Larger anchors represent coarse scene structures (Figure 7(b)), while progressively smaller anchors handle finer-scale features (Figure 7(c)(d)), ensuring that the low-bit quantized BB coordinates, representing offsets within these progressively smaller anchor BBs, maintain sufficient precision even for detailed geometry. This hierarchical structure allows the quantization to adapt to varying scene scales and densities, preserving accuracy where needed while maximizing the benefits of low-bit integer operations in localized regions. By enabling direct low-bit integer traversal within these localized, relatively encoded BBs,

multi-level quantization offers a promising direction for reducing memory traffic and computational complexity, effectively leveraging scene sparsity and mitigating the pitfalls of precision reduction.

4 Quantized BVH Tree

In the previous section, we introduced multi-level quantization as a technique to mitigate the memory bandwidth and computational costs associated with traditional FP32-based BVH traversal. This technique results in a specialized data structure, termed *quantized BVH tree*, which is specifically designed to be processed efficiently by hardware accelerators like AQB8 (detailed in Section 5). This section describes the structure of this quantized BVH tree, covering its quantization approach, data layout, and traversal algorithm.

4.1 Overview

Recall from Section 3 that multi-level quantization leverages scene sparsity by organizing BVH nodes into clusters established by high-precision anchor BBs. Building upon this concept, the quantized BVH tree structure differs significantly from a standard BVH tree where all BBs are stored using FP32 precision (as depicted in Figure 8(a)). For example, in a standard BVH tree, node A might have its BB stored as (0.66, 1.64, 0.69), node C as (0.66, 0.82, 0.68), and node G as (0.51, 0.82, 0.31), all in FP32.

In contrast, our quantized BVH tree (Figure 8(b)) organizes nodes into *clusters* (e.g., U1-U4). Each cluster contains:

- An *anchor BB*, retained in FP32, serving as the high-precision reference frame for that cluster.
- Multiple *quantized BBs*, stored in INT8, whose coordinates are encoded *relative* to the anchor BB.

For instance, consider cluster U1 in Figure 8(b) (purple dotted line). Its anchor BB is derived from node A (colored purple), using its FP32 coordinates (0.66, 1.64, 0.69). Nodes B, C, D, and E all belong to U1, and thus have their quantized BBs stored in INT8 relative to the anchor BB of U1. Specifically, node C's BB representation within U1 is the quantized INT8 coordinate (254, 126, 194), interpreted relative to the anchor BB of U1.

Now, consider cluster U4 (blue dotted line). Its anchor BB is derived from node C (colored blue), using its FP32 coordinates (0.66, 0.82, 0.68). Nodes within U4, such as G, have their quantized BBs stored in INT8 relative to the anchor BB of U4. Specifically, node G's BB representation within U4 is the quantized INT8 coordinate (61, 32, 200), interpreted relative to the anchor BB of U4.

The specifics of how these anchor and quantized BBs are represented are detailed next.

4.2 Bounding Box Quantization

In our quantized BVH tree, each anchor BB is defined by its minimum and maximum coordinates, represented using six 32-bit floating-point (FP32) values, consistent with the approach used in standard BVH trees. Each cluster contains a single anchor BB, which serves as the spatial reference for all quantized BBs within that cluster. The cluster operates within its own local coordinate system, spanning the range [0, 255] for the x, y, and z axes, with 256 evenly spaced values. Quantized BBs are encoded using six 8-bit integers (INT8), with their coordinates expressed in this [0, 255] coordinate system.

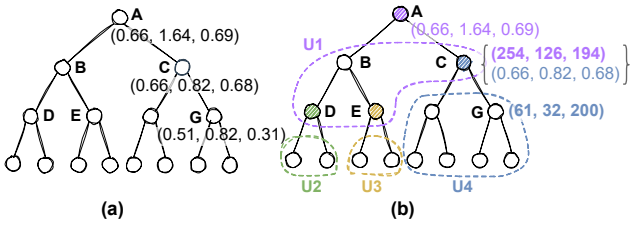

Figure 8: (a) The standard BVH tree. (b) The proposed quantized BVH tree. Non-bold coordinates represent FP32 values, while *bold coordinates* indicate quantized INT8 values.

A quantized BB is constructed as the smallest BB within the [0, 255] coordinate system that fully encloses the corresponding FP32 BB. To ensure complete coverage, the minimum and maximum coordinates of the quantized BB are adjusted outwards, which may result in a slightly larger volume compared to the original FP32 BB due to quantization error. Building upon this quantization scheme, the following section details the algorithm used to construct the complete quantized BVH tree.

4.3 Tree Construction

4.3.1 Overview. The construction of a quantized BVH tree involves two steps. First, a standard BVH tree is constructed using any existing BVH tree construction algorithm. Next, a clustering algorithm is applied to partition the BVH tree into multiple clusters. Notably, our quantized BVH construction method is agnostic to the specific BVH construction algorithm employed in the first step, offering flexibility to substitute it with alternative algorithms as needed.

4.3.2 The Cost Function. The clustering strategy for the quantized BVH tree is guided by a cost function derived from the Surface Area Heuristic (SAH) [73]. The SAH models the geometric probability of traversing a child node N_c given that its parent node N is intersected. This probability is expressed as:

$$P(N_c|N)^{\text{SAH}} = \frac{S(N_c)}{S(N)},$$

where $S(N)$ represents the surface area of the BB for node N. Using this principle, the total cost of a BVH tree rooted at node N is defined recursively as:

$$C(N) = \begin{cases} c_t + \sum_{N_c} \frac{S(N_c)}{S(N)} C(N_c), & \text{if } N \text{ is an internal node,} \\ c_i|N|, & \text{otherwise.} \end{cases}$$

Here, N_c refers to the children of node N, and c_t and c_i are the costs of a traversal step and a ray-triangle intersection, respectively. $|N|$ denotes the number of triangles in the leaf node N. By expanding this recurrence relation, the total cost of a BVH tree can be unrolled into:

$$C(N) = \frac{1}{S(N)} \left[c_t \sum_{N_i} S(N_i) + c_i \sum_{N_l} S(N_l)|N_l| \right],$$

where $\sum_{N_i}$ iterates over all internal nodes, and $\sum_{N_l}$ iterates over all leaf nodes in the subtree rooted at N.

To incorporate the concept of clusters into the cost function, the cost of a traversal step (c_t) is modified as a function of the node N_i.

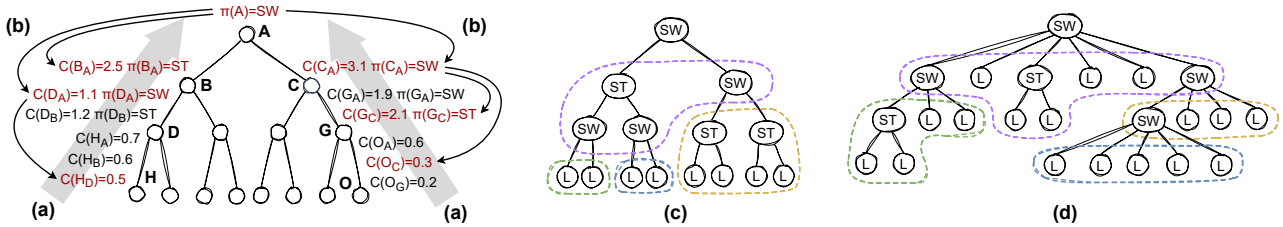

Figure 9: Constructing a Quantized Binary (b) and 6-Wide (c) BVH tree with (SW: `SWITCH`, ST: `STAY`, L: `LEAF`) States.

Specifically, the cost of the quantized BVH tree rooted at N is:

$$C(N) = \frac{1}{S(N)} \left[\sum_{N_i} T(N_i) S(N_i) + c_i \sum_{N_l} S(N_l) |N_l| \right],$$

where $T(N_i)$ is defined as:

$$T(N_i) = \begin{cases} c_t + c_s, & \text{if } \pi(N_i) = \texttt{SWITCH}, \\ c_t, & \text{if } \pi(N_i) = \texttt{STAY}. \end{cases}$$

Here, c_s represents the cost of switching clusters, and $\pi(N_i)$ denotes the clustering policy for node N_i, which can be either `SWITCH` or `STAY`. When $\pi(N_i)$ = `SWITCH`, traversing N_i involves switching clusters; otherwise, $\pi(N_i)$ = `STAY`.

For instance, in Figure 8(b), the clustering policies for nodes A, C, D, and E are `SWITCH`, as traversing to these nodes requires a cluster change. In contrast, all other internal nodes follow the `STAY` policy.

The parameters c_t, c_i, and c_s play a crucial role in shaping the structure of the quantized BVH tree, as they directly influence the cost function guiding the clustering process. Empirically, we set $[c_t, c_i, c_s]$ to $[0.5, 1, 1]$, respectively. However, these values are not fixed and can be adjusted by the programmer to generate different quantized BVH trees. For example, lowering c_s reduces the cost for switching clusters, encouraging the formation of more clusters. Similarly, modifying c_t or c_i alters the balance between traversal steps and ray-triangle intersection tests, providing further flexibility in optimizing the BVH tree for specific hardware or graphic rendering workloads.

4.3.3 The Clustering Algorithm. After modeling the quality of a quantized BVH tree into a cost value, the optimal quantized BVH tree can be selected by *choosing the tree with the minimum cost*. The clustering algorithm determines the clustering policy $\pi(N_i)$ for every internal node N_i. Once every $\pi(N_i)$ is defined, the cluster scheme is established, and the quantized BVH tree is fully determined.

The clustering policy $\pi(N_i)$ can, in theory, be determined by enumerating all possible configurations and selecting the one that minimizes the cost of the quantized BVH tree. However, this brute-force approach is computationally infeasible due to its complexity of $O(2^n)$, where n represents the number of internal nodes—a typically large number in practical applications. To address this challenge, we take advantage of the insight that finding the quantized BVH tree with minimum cost possesses the property of *optimal substructure*, allowing us to solve the problem efficiently using dynamic programming. Specifically, we associate multiple costs with each node. As illustrated in Figure 9(a), consider the node D with its ancestor nodes A and B. The node D has two associated costs—$C(D_A)$, and $C(D_B)$—and two corresponding clustering policies—$\pi(D_A)$ and $\pi(D_B)$. Here, $C(X_Y)$ and $\pi(X_Y)$ represent the cost and clustering policy of the cost-optimal subtree rooted at X when the BB of X is quantized with respect to the BB of Y.

These costs and clustering policies are computed in a bottom-up manner (Figure 9(a)). Once all costs and clustering policies have been determined, the optimal quantized BVH tree can be identified by backtracking through the tree in a top-down approach (Figure 9(b)). The constructed quantized BVH tree is illustrated in Figure 9(c), where each node is identified as a `SWITCH`, `STAY`, or `LEAF` node (the specific definitions of these node types are elaborated in Section 4.6). Our clustering algorithm runs in $O(n(\log n)^2)$, which is similar to the $O(n \log n)$ complexity of standard BVH construction algorithms [66]. Consequently, this algorithm does not introduce significant overhead to the quantized BVH construction process.

4.3.4 Generalizability. While Figures 9(a)-(c) illustrate the construction algorithm in the context of a binary BVH tree, the approach generalizes naturally to other tree topologies. For example, Figure 9(d) demonstrates an example of the quantized 6-wide BVH tree. The clustering strategy remains fundamentally the same, with the key difference being that the cost function now considers a wider set of possible child policy assignments. For 6-wide BVH tree, we empirically set $[c_t, c_i, c_s]$ to $[0.5, 1, 0.6]$, respectively.

4.4 Ray Quantization

As described in Section 2.1.3, before performing ray-box intersection tests, rays are preprocessed into two 3-dimensional vectors, $\mathbf{w}$ and $\mathbf{b}$. In most RT implementations, these six numbers are represented using FP32 values. However, since our quantized BBs are encoded with 8-bit integers in a cluster-specific local coordinate system, the original ray-box intersection equation cannot be applied directly. To address this, we quantize the rays to align with the coordinate system of the quantized BBs.

Our approach differs from prior works on BB compression [7, 23, 32, 37, 69, 77], which typically transform quantized BBs into a data format compatible with rays for ray-box intersection tests. In contrast, we take a reverse approach: we *transform the ray to fit the quantized BBs*. Although this requires re-quantizing rays when traversing into a different cluster, we found that the number of clusters is significantly fewer than the number of BBs. As a result, the re-quantization of ray data introduces negligible performance overhead while simplifying the overall ray-box intersection process.

4.4.1 Details on Ray Quantization. For clarity in the following derivations, we will use w and b to refer to the individual scalar components (e.g., w_x, b_x) of the vectors $\mathbf{w}$ and $\mathbf{b}$. To quantize these components, we introduce scaling factors S_w and S_x for quantization. The scaling factor S_w is fixed at 2^{-7}, while S_x varies depending on the cluster. Each cluster has its specific S_x, which is defined as:

$$S_x = \max(l_x, l_y, l_z)/255,$$

where l_x, l_y, and l_z are the length of the anchor BB along the x, y, and z axes for the cluster.

Using these scaling factors, w is quantized by dividing it by S_w ($q_w = w/S_w$), where q_w is represented in a custom 14-bit floating-point (FP14) format with 1 sign bit, 8 mantissa bits, and 5 exponent bits. b is quantized using both scaling factors ($q_b = b/(S_w S_x)$), which is represented as a 32-bit integer (INT32). Since each cluster has its own S_x, b must be re-quantized when transitioning to a new cluster.

4.4.2 Custom Floating-Point Representation of q_w. We represent q_w using a custom FP14 format. This choice is motivated by the distribution characteristics of the inverse direction component w. When the ray direction is a uniformly distributed unit vector, the inverse direction component w follow a probability density function given by:

$$f(w) = \begin{cases} \frac{1}{2w^2}, & \text{if } |w| \geq 1, \\ 0, & \text{otherwise.} \end{cases}$$

This distribution indicates that smaller values of w occur with higher frequency compared to larger values, leading to a non-uniform density that favors lower magnitudes.

If w were quantized using uniform intervals, the resulting q_w values would be unevenly distributed, with a higher density at lower values. Hence, to better capture this distribution, we first apply quantization to w and then represent q_w in the custom FP14 format. Since floating-point representation provides finer granularity for values near zero, it aligns well with the distribution characteristics of w.

4.5 The Quantized Ray-Box Intersection

In our quantized BVH tree, the original ray-box intersection formulation cannot be applied directly due to the use of quantized ray and BB representations. This necessitates the development of a *quantized* ray-box intersection algorithm. Below, we derive the formula for quantized ray-box intersection step by step.

As described in Section 2.1.3, the ray-box intersection test can be formulated as:

$$t = wx + b$$

where t is the parameter indicating the intersection point along the ray, w and b are the preprocessed ray components, and x represents the coordinates of the BB.

To adapt this equation for quantized representations, we linearly quantize t, w, x, and b using scaling factors $S_w S_x$, S_w, S_x, and $S_w S_x$, respectively. Substituting these quantized variables into the original equation gives:

$$S_w S_x q_t = S_w q_w S_x q_x + S_w S_x q_b$$

where q_t, q_w, q_x, and q_b are the quantized counterparts of t, w, x, and b.

Next, represent q_w in floating-point form as:

$$q_w = i_w 2^{r_w} m_w,$$

where i_w is the sign bit, r_w is the exponent, and m_w is the mantissa. Substituting this into the quantized equation yields:

$$S_w S_x q_t = S_w i_w 2^{r_w} m_w S_x q_x + S_w S_x q_b$$

By canceling the common factor $S_w S_x$, we derive the final equation for the quantized ray-box intersection:

$$q_t = i_w \, 2^{r_w} \underbrace{m_w q_x}_{\text{INT8}} + q_b$$

$$\underbrace{\qquad\qquad}_{\text{left shift}}$$

$$\underbrace{\qquad\qquad\qquad}_{\text{2's complement}}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{\text{INT32}}$$

This quantized ray-box intersection algorithm can be computed efficiently in hardware using integer arithmetic, left shifts, and 2's complement operations. The steps are: (1) Multiply m_w by q_x using INT8 arithmetic. (2) Perform a left shift of the result by r_w bits. (3) If i_w equals 1, negate the result using 2's complement. (4) Add the final value to q_b using INT32 arithmetic.

This approach eliminates the need for costly FP32 arithmetic (originally, t, w, x, and b are all represented in FP32). By leveraging low-bit integer arithmetic, the proposed quantized ray-box intersection algorithm reduces computational cost and minimizes hardware complexity.

4.6 Memory Layout

The quantized BVH tree introduces the concept of clusters and redefines how nodes are represented in memory to enhance encoding efficiency. Figure 10 illustrates the memory layouts for both the standard BVH tree (Figure 10(a)) and our quantized BVH tree (Figure 10(b)).

A cluster in the quantized BVH tree consists of several components: the bounds of the anchor BB, $S_w S_x$, the node base index, and the triangle base index. The anchor BB is encoded using six FP32 numbers, occupying a total of 24 bytes. The $S_w S_x$ value is used to quantize rays when they enter a new cluster. The node base index and triangle base index are offsets used to adjust the indices of nodes and triangles within the cluster.

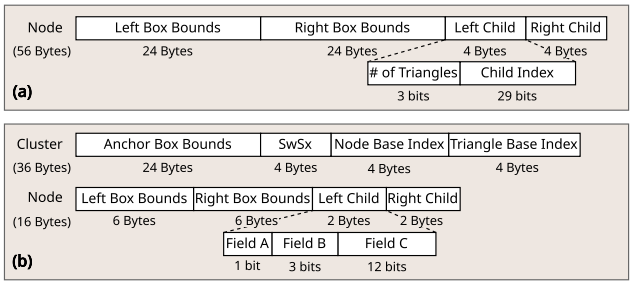

Figure 10: Memory layout for (a) standard BVH tree and (b) our quantized BVH tree.

To reduce memory usage, the quantized BVH tree uses a significantly compressed memory layout for nodes by:

- Reducing BB size by a quarter. BBs are encoded using INT8 instead of FP32 as in the standard BVH tree, reducing the node size from 56 bytes to just 16 bytes.
- Redesigning the bitfield for child data. In the quantized BVH tree, each child data is represented using 2 bytes, divided into three fields (Field A, B, and C), whereas the standard BVH tree uses 4 bytes with two fields (number of triangles and child index).

Although the quantized BVH tree introduces an additional "cluster" structure, which incurs extra memory overhead, the size of a cluster is small (36 bytes). Moreover, the number of clusters is significantly smaller than the number of nodes (Table 1). Consequently, the overall memory footprint of the quantized BVH tree is significantly reduced, making it far more memory-efficient than the standard BVH tree.

Children of each quantized BVH tree node are categorized into three types: `SWITCH`, `STAY`, and `LEAF`. Each type is uniquely identified by examining the values of Field A and Field B:

- `SWITCH`: The child is an internal node and traversing to this node necessitates a cluster switch. Field A is set to `0`, while Fields B and C together represent the index of the new cluster.
- `STAY`: The child is an internal node and no cluster switch is necessary to traverse this node. Field A is set to 1, Field B is set to `000`, and Field C stores the index offset of the child node relative to the cluster's node base index.
- `LEAF`: The child is a leaf node. Field A is set to 1, Field B stores the number of triangles (at least 1), and Field C holds the starting index offset of the triangles relative to the cluster's triangle base index.

Nodes and triangles within the same cluster are stored contiguously in memory, with the base indices encoded in the cluster data. This ensures that child nodes and triangles can be efficiently addressed using smaller offsets rather than larger absolute indices. Moreover, fetching one node brings in its neighboring nodes in the same cluster into cache, improving memory access efficiency. As nodes in the same cluster are likely to be accessed in sequence during traversal, this spatial locality reduces cache misses and DRAM accesses, further alleviating memory bottlenecks and enhancing overall RT efficiency.

4.7 Tree Traversal

Algorithm 1 outlines the tree traversal algorithm for the quantized BVH tree, which aims to efficiently find the closest intersection point between a ray and triangles. For compactness in typesetting, we use "q" to denote "quantized" in variable and function names. For instance, "qRay" refers to the quantized ray, and "qRayBoxTest" refers to the quantized ray-box intersection test.

The algorithm begins by initializing a stack with the BVH tree's root node (line 3). The main traversal loop continues until the stack is empty (lines 4-21). In each iteration, the node at the top of the stack is popped (line 5), and its type determines subsequent actions:

- `SWITCH` node (lines 7-12): A ray-box test is performed on the ray and the anchor BB associated with the current node's cluster, during which the quantized ray is also computed (line 8). If the ray intersects the anchor BB (line 9), quantized ray-box tests are conducted on the quantized ray and the quantized BBs of the child nodes (lines 10-11). Intersected child nodes are pushed onto the stack (line 12).
- `STAY` node (lines 13-16): For each child node, a quantized ray-box test is conducted (lines 14-15). Nodes that are intersected are pushed onto the stack (line 16).
- `LEAF` node (lines 17-21): For each triangle in the current node, a ray-triangle test is performed (line 18-19). If the ray intersects that triangle (line 20), the closest intersection point is updated (line 21).

Algorithm 1 Quantized BVH Tree Traversal Algorithm

```
1:  inputs: ray, rootNode
2:  initialize: qRay ← null; stack ← empty; intersectionPoint ← null
3:  stack.push(rootNode)
4:  while stack is not empty do
5:      curNode ← stack.pop()
6:      re-quantize ray if needed
7:      if type(curNode) = SWITCH then
8:          hit, qRay ← RayBoxTest(ray, curNode.cluster.anchorBox)
9:          if hit then
10:             for childNode in curNode do
11:                 if qRayBoxTest(qRay, childNode.qBox) then
12:                     stack.push(childNode)
13:     else if type(curNode) = STAY then
14:         for childNode in curNode do
15:             if qRayBoxTest(qRay, childNode.qBox) then
16:                 stack.push(childNode)
17:     else if type(curNode) = LEAF then
18:         for triangle in curNode do
19:             hit, tmpPoint ← RayTriangleTest(ray, triangle)
20:             if hit then
21:                 intersectionPoint ← tmpPoint
22: return intersectionPoint
```

In certain scenarios, the ray must be *re-quantized* (line 6). Specifically, when "jumping back" to a node that was previously placed onto the stack, if the cluster of the node being popped from the stack differs from the current cluster, the ray must be re-quantized to align with the local coordinate system of the cluster associated with the node being processed. This step ensures that the ray can correctly interact with the quantized BBs in the associated cluster. After the traversal loop concludes, it returns the closest intersection point (line 22) for further processing.

5 Quantized Ray Tracing Accelerator

RT accelerators play a pivotal role in speeding up ray tracing operations. To fully exploit the benefits of the quantized BVH tree, we propose the *quantized RT accelerator*, which builds upon a *baseline RT accelerator*—with key architectural modifications highlighted in bold in Figure 11—to efficiently handle the traversal of quantized BVH trees. Due to the lack of publicly available details on the internal design of RT accelerators in commercial GPUs, we design our baseline RT accelerator similar to the RT Unit described in [56].

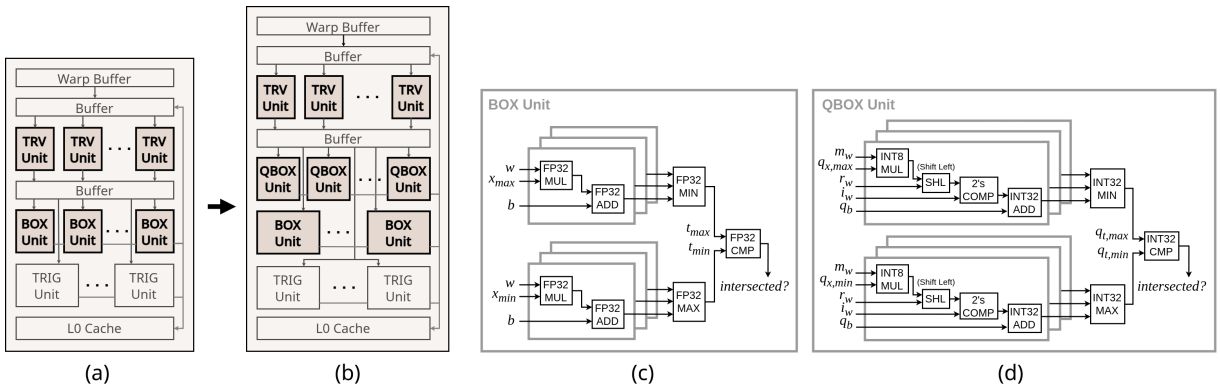

Figure 11: Architectural modifications from a (a) baseline RT accelerator to our (b) quantized RT accelerator. The hardware block diagrams for the BOX and QBOX units are detailed in (c) and (d), respectively.

5.1 Architectural Modifications

The baseline RT accelerator (Figure 11(a)) consists of four core components:

- TRV Units: Handle tree traversal logic and stack management.
- BOX Units: Perform ray-box intersection tests for internal nodes.
- TRIG Units: Perform ray-triangle intersection tests for leaf nodes.
- L0 Cache: A small on-chip memory that stores nodes and triangles. If the requested data is not found in the L0 cache, the next-level memory access is triggered.

Our quantized RT accelerator (Figure 11(b)) builds on these units by adapting existing components and adding new hardware:

- Modified TRV Units: These units handle the traversal logic for quantized BVH trees, incorporating the necessary cluster switching operations and the traversal policies for SWITCH, STAY, and LEAF nodes as described in Algorithm 1.
- Modified BOX Units: The BOX units are redesigned to account for the calculation of quantized rays when performing ray-box intersection tests on anchor BBs (Algorithm 1, line 8).
- New QBOX Units: Introduced specifically for handling quantized ray-box intersection tests on quantized BBs. These units use efficient integer arithmetic operations, which makes them fast, energy-efficient, and smaller in area compared to the BOX units.

Figures 11(c) and 11(d) illustrate the difference between BOX and QBOX units. The BOX unit performs ray-box intersection using FP32 arithmetic (following the equations in Section 2.1.3), which requires costly floating-point multiplications, additions, and comparisons. In contrast, the QBOX unit operates entirely on integer arithmetic, replacing FP32 operations with efficient low-bit integer multiplications, shifts, and additions (following the equations in Section 4.5).

5.2 Design Implications

The architectural modifications in the quantized RT accelerator bring significant changes to the way BBs are processed. By introducing specialized QBOX units, the majority of ray-box intersection tests—performed on quantized BBs—are efficiently handled using lightweight integer-only operations. This approach minimizes the reliance on the more complex floating-point-based BOX units, which are now primarily reserved for processing the relatively small number of anchor BBs. As a result, most of the BOX units in the baseline design are replaced with more efficient and compact QBOX units, leading to a reduction in hardware complexity. While the TRV units experience a slight increase in area due to the increased traversal logic required for cluster-switching operations, this increase is offset by the reduction in BOX units. The design thus achieves a net reduction in area while delivering substantial improvements in computational efficiency.

The memory subsystem also benefits from these changes. Reducing the size of BB data allows the cache to store more BBs, improving cache utilization and reducing DRAM traffic. This alleviates memory bottlenecks, particularly for memory-bound workloads.

These architectural changes collectively contribute to both computational and memory efficiency. Detailed evaluation on these improvements will be discussed in Section 7.

6 Methodology

In this section, we describe our experimental setup and evaluation approach for assessing the effectiveness of our proposed multi-level quantization technique in both low-level hardware simulations and full-system GPU simulations. We begin with an explanation of our comparison approach, then present the details of our evaluated workloads, accelerator models, and GPU integration.

6.1 Comparison Approach

Our primary goal is to demonstrate that multi-level quantization *is a broadly applicable optimization*, rather than only for a specific BVH tree structure. In principle, almost all BVH trees can benefit from substituting their BBs with quantized ones and adapting the traversal logic accordingly. To evaluate this general applicability, we consider two BVH configurations: (1) a *2-ary BVH tree* constructed with an open-source library [4], and (2) a *6-wide BVH tree* built using Embree [75]. For each configuration, we compare a *baseline RT accelerator* (storing all bounding boxes in FP32) with a *compressed RT accelerator* (compressing all bounding boxes in INT8 based on [77]) and our *quantized RT accelerator* (compressing most bounding boxes in INT8 with multi-level quantization). All configurations share the same tree topology for their respective

Table 1: Statistics of evaluated scenes.

	KIT[1]	BA	BMW	CLA[1]	HOU	STR[1]	TEA
							
# of rays	0.92M	1.05M	1.40M	0.92M	0.92M	0.92M	0.92M
# of triangles	1.44M	0.59M	0.79M	0.10M	1.82M	0.26M	0.13M
# of nodes	1.39M	0.55M	0.78M	0.09M	1.79M	0.23M	0.11M
# of clusters	0.021M	0.005M	0.012M	0.003M	0.032M	0.003M	0.001M
Baseline-2 BVH Size[2]	39.0MB	15.4MB	22.0MB	2.6MB	50.1MB	6.4MB	3.1MB
Compress-2 BVH Size[2]	16.0MB	6.3MB	9.0MB	1.1MB	20.6MB	2.6MB	1.3MB
AQB-2 BVH Size[2]	11.9MB	4.6MB	6.7MB	0.8MB	15.5MB	1.9MB	0.9MB

[1] The following scenes are licensed under Creative Commons Attribution 3.0 Unported (CC BY 3.0): Country Kitchen (KIT) by Jay-Artist, Japanese Classroom (CLA) by NovaZeeke, and The Wooden Staircase (STR) by Wig42. License details: https://creativecommons.org/licenses/by/3.0/.

[2] Including BBs, child indices, and other metadata.

branching factors to isolate the impact of multi-level quantization without introducing additional factors that could obscure or inflate its benefits.

6.2 Evaluated Workloads

Our evaluation spans seven diverse scenes [9, 51], covering a wide range of geometric and lighting complexities. Table 1 overviews these scenes, including indoor settings like the bathroom (BA), with reflective surfaces and indirect lighting, and outdoor environments like the house (HOU), which emphasize environmental daylight. For each scene, we employ two types of workloads:

(a) *Replay-based:* For energy consumption analysis, we pre-collect rays from pbrt-v4 [52] (one sample per pixel) and replay these ray traces on our accelerators in a cycle-accurate simulation. Here, we use a functionally simulated memory subsystem rather than a full GPU model.

(b) *Vulkan-Sim-based:* For analyses other than energy measurement, we generate rays at runtime and evaluate them using Vulkan-Sim [56]. These simulations were conducted at a resolution of 256×256 per scene, integrating a complete GPU memory subsystem, which allows us to capture a more realistic performance perspective at scale.

By separating these two evaluation strategies, we can (a) precisely isolate the energy impacts through a controllable "replay" method, and (b) demonstrate practical, real-time efficiency improvements within a complete GPU simulation environment.

6.3 RT Accelerator Hardware Modeling

6.3.1 Design and Implementation. We model three RT accelerator designs—*baseline*, *compressed*, and *quantized*—in a cycle-accurate manner. All designs are synthesized with Catapult HLS [60] and Design Compiler [64] with TSMC 40nm cell library [68], and simulated cycle-accurately with QuestaSim [61]. The main processing units—responsible for BVH traversal, ray-box, and ray-triangle intersection—are fully synthesized. FIFO buffers and on-chip interconnects are cycle-accurately simulated but not synthesized.

6.3.2 Hardware Resource Sizing. We determine the number of each processing unit (e.g., TRV, BOX, QBOX, TRIG) based on typical RT workloads. The exact counts are shown in Figure 15. Both *compressed* and *quantized RT accelerator* incur slightly more traversal steps, so to maintain similar throughput, these accelerators can include additional TRV, BOX (or QBOX), and TRIG units. While this increases hardware area, it ensures all accelerators operate at comparable throughput when tested under the same scenarios.

6.3.3 Energy Measurement. We assess the overall energy consumption of the rendering process by combining measurements from both the compute and memory subsystems. For compute part, we simulate the synthesized hardware at gate level, and use PrimePower [65] to obtain each unit's power consumption. We then compute the energy consumed by each unit by multiplying its power by the corresponding operation count (gathered from simulations) and by the clock period.

For the memory subsystem, we follow the approach in [19] and use CACTI [5] to determine the energy per read/write access in caches, while modeling DRAM energy consumption at a fixed 6.5 pJ/bit (GDDR6 [41]). Our functional memory model includes an L1–L2 cache hierarchy, where the L1 cache is 32KB, 4-way associative, with a 64B line size, and the L2 cache is 1MB, 8-way associative, also with 64B lines. We employ our cache simulator to count various cache statistics and then compute the energy by multiplying these counts with the corresponding per-access energy reported by CACTI and with the modeled DRAM energy.

To avoid conflating memory-access delays with the accelerators' energy consumption, our functional memory model assumes unlimited memory bandwidth and zero-latency data transfers. This abstraction lets us focus on measuring the intrinsic energy consumption of both the accelerators and the memory subsystem.

6.4 GPU Hardware Modeling

We implement the *baseline*, *compressed*, and *quantized RT accelerators* on the Vulkan-Sim [56] to enable full-system GPU simulation. In Vulkan-Sim, the BVH traversal algorithm is functionally modeled to generate each accelerator's memory access stream, which is then fed into Vulkan-Sim's timing model. We enhance the simulator to support both tree traversal algorithms: for the *compressed RT accelerator*, we use its standard INT8-to-FP32 decompression scheme [69, 77], while for the *quantized RT accelerator*, we implement modifications that correctly compute and fetch anchor and quantized BBs. This integration allows us to evaluate the impact of multi-level quantization on memory traffic and rendering performance under a realistic GPU execution model. Table 2 summarizes the key simulation settings used in our evaluations on Vulkan-Sim.

Table 2: Vulkan-Sim simulation configurations.

Number of SMs	30	L1 Data Cache / SM	64KB
Max Warps / SM	32	L2 Unified Cache	3MB
Warp Size	32	Compute Core Clock	1365MHz
Warp Scheduler	GTO	# of RT Accelerators / SM	1
# of Registers / SM	32768	RT Accelerator Warp Buffer Size	4
Global Memory Clock	3500MHz	RT Dedicated L0 Cache	8KB

7 Evaluation Results

In this section, we evaluate the effectiveness of our proposed method across five perspectives: memory traffic, energy consumption, tree traversal steps, hardware area, and performance. We compare multiple variants of the *baseline* RT accelerator, the *compressed* RT accelerator, and our *AQB8* quantized RT accelerator under different

BVH branching factors. We denote these evaluations as *Baseline-2*, *Compress-2*, and *AQB8-2* when using 2-ary BVH trees, and as *Baseline-6*, *Compress-6*, and *AQB8-6* when using 6-wide BVH trees.

7.1 Memory Traffic

7.1.1 L1D/L2 Cache Requests. Figure 12(a) shows the total number of L1D cache requests, while Figure 12(b) shows the total number of L2 cache requests for different accelerator designs. With multi-level quantization, *AQB8* reduces the average L1D cache requests by 65–82% (74% on average) and L2 cache requests by 48-68% (60% on average). This reduction is achieved not only because the BBs are stored in a substantially smaller size, allowing more data to fit within the upper-level caches, but also because *AQB8* exploits spatial locality—nodes within the same cluster are stored contiguously. This contiguity ensures that when one node is fetched, its neighboring nodes in the same cluster are likely brought into the cache as well, further lowering the overall frequency of cache requests.

7.1.2 DRAM Accesses. Figure 12(c) compares the total DRAM accesses. *AQB8* reduces DRAM accesses by 61-76% (70% on average). As DRAM accesses are especially costly in terms of both latency and power, this reduction alleviates the memory bottleneck and contributes to faster and more energy-efficient graphic rendering.

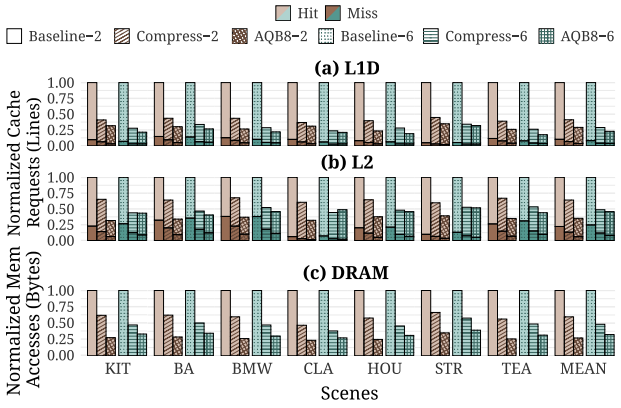

Figure 12: Comparison of memory traffic on (a) L1D cache, (b) L2 cache, and (c) DRAM.

7.2 Energy Consumption

Figure 13 presents the normalized energy consumption of *Baseline-2*, *Compress-2*, and *AQB8-2* across the evaluated scenes. Although *Compress-2* effectively reduces memory traffic by storing BBs in compressed format, its on-the-fly decompression and continued reliance on FP32 arithmetic keep compute energy high and actually increase it across all tested scenes. By contrast, *AQB8-2* lowers both compute and memory energy: most ray-box intersections are handled with low-bit integer arithmetic, reducing dynamic power in hardware units, and its quantized (INT8) representation combined with the cluster-based layout improves cache locality, reducing energy-consuming accesses to bottom-level caches and DRAM beyond what existing compression methods achieve. As a result, *AQB8-2* achieves 29–49% (42% on average) total energy savings compared to *Baseline-2* across the evaluated scenes.

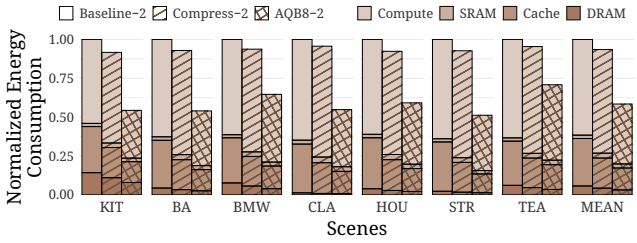

Figure 13: Comparison of energy consumption.

7.3 Tree Traversal Steps

Figure 14 compares the number of ray-box and ray-triangle intersection tests among *Baseline-2*, *Compress-2*, and *AQB8-2*. Storing bounding boxes in low-bit representation inevitably introduces quantization errors, slightly expanding the bounding boxes and causing false-positive intersections. For *AQB8-2*, this expansion increases ray-box intersection tests by 3–6% and ray-triangle intersection tests by 6–31%. Although the percentage increase is larger for ray-triangle tests, their total count is much lower than that of ray-box tests. As a result, the overall impact on traversal remains modest, and the notable gains in memory efficiency and energy savings outweigh this minor overhead.

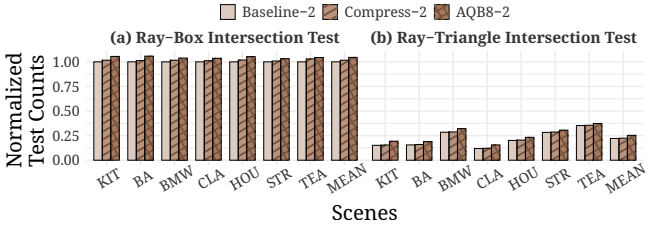

Figure 14: Normalized counts of (a) ray-box and (b) ray-triangle intersection tests while traversing BVH trees.

7.4 Hardware Area

Table 3 presents the hardware area of each single compute unit on different RT accelerator designs. As shown in Table 3, the QBOX unit that operates on INT8 arithmetic has a 5.1x smaller hardware area than the FP32 BOX unit. The modified TRV units in *AQB8-2* require slightly more area to handle the cluster-switching logic

Table 3: Hardware area, energy per operation, and average operation counts of each unit on different accelerators.

		TRV	QBOX	BOX	TRIG
Hardware Area (mm^2)	Baseline-2	7.88	-	**82.13**	192.92
	Compress-2	8.12	-	**90.58**	192.92
	AQB8-2	9.80	**16.08**	97.05	192.92
Energy (nJ) Per Operation	Baseline-2	6.0e-3	-	**1.38e-1**	2.90e-1
	Compress-2	8.4e-3	-	**1.51e-1**	2.90e-1
	AQB8-2	5.5e-3	**2.43e-2**	1.56e-1	2.90e-1
Average Operation Counts	Baseline-2	35.9M	-	28.8M	6.07M
	Compress-2	36.3M	-	29.2M	6.17M
	AQB8-2	43.1M	30.0M	4.3M	7.0M

needed for the quantized BVH tree. Similarly, the modified BOX units in *AQB8-2*, which additionally calculate quantized rays, are also slightly larger than the BOX units in *Baseline-2*. Despite this, *AQB8-2* reduces overall hardware area by replacing most of the FP32 BOX units with area-efficient INT8 QBOX units.

Figure 15 compares the hardware area of *Baseline-2*, *Compress-2*, and *AQB8-2*, with numbers inside the bars indicating each hardware unit counts. Since both *Compress-2* and *AQB8-2* introduce slightly more traversal steps, we use linear extrapolation based on the operation counts in *Baseline-2* to determine the required number of additional TRV, BOX (or QBOX), and TRIG units in *Compress-2* and *AQB8-2*, ensuring a fair comparison by maintaining comparable throughput across all accelerators. While *Compress-2* reduces memory traffic, it retains FP32-based BOX units and introduces decompression logic, leading to a net area increase of 8%. In contrast, *AQB8-2* replaces *Baseline-2*'s 51 BOX units with 53 smaller QBOX units (which use simplified integer arithmetic) and 9 BOX units for anchor bounding boxes, resulting in a 27% reduction in total hardware area compared to *Baseline-2*, demonstrating the cost-effectiveness of our quantization approach.

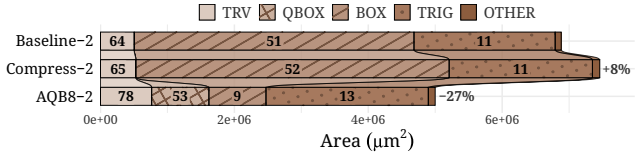

Figure 15: Comparison of hardware area. Numbers inside the bars indicate the count of each hardware unit.

7.5 Performance

Figure 16 illustrates the performance speedups of different RT accelerator designs. *Compress-2*, *AQB8-2*, *Compress-6*, and *AQB8-6* respectively achieve a geometric mean speedup of 1.31x, 1.82x, 1.19x, and 1.43x compared to their respective *Baseline* implementations. The *Compress* implementations improve performance mainly by reducing memory traffic via bounding box compression; however, the gains are limited due to decompression overhead and continued reliance on FP32 compute units. In contrast, *AQB8* delivers greater performance gains across both 2-ary and 6-wide setups by synergistically combining memory traffic reduction from the quantized BVH structure with the efficiency of performing most traversal operations directly using INT8 arithmetic, thereby minimizing costly FP32 ray-box intersection operations.

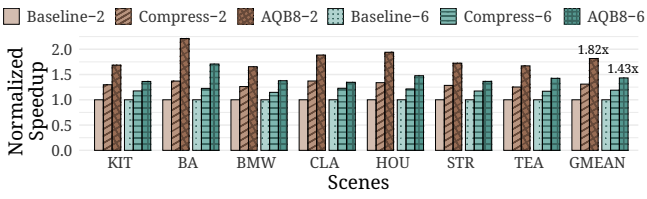

Figure 16: Comparison of RT accelerator performance.

8 Related Work

Memory-Efficient Ray Tracing. Prior work on improving RT's memory efficiency spans several directions, including compression [7, 16, 20, 23, 26, 32, 37, 54, 58, 69, 77], prefetching [2, 3, 10], stackless traversal [8, 17, 28, 70], and cache-efficient layouts [32, 47, 78]. While compression techniques can reduce memory traffic, they often require decompression before traversal and still rely on FP32 arithmetic, limiting computational efficiency.

RT Accelerators and Optimizations. To meet real-time rendering requirements, numerous RT accelerators have been developed [13, 18, 24, 25, 27, 30, 31, 45, 46, 48, 55, 57, 59, 62, 72, 76], with many targeting GPUs, where divergent rays can still incur high traversal overhead [3]. Techniques such as thread compaction [29, 36, 63, 71, 74], ray prediction [33], and maximizing ray coherency [12, 14, 39, 42, 49, 53] have been employed to boost efficiency.

Generalizing RT Accelerators on Other Domains. Recent research leverages RT accelerators for applications in different domains, such as approximate nearest neighbor search [35] and other applications structured as tree traversals [6, 15, 38, 43, 44].

Comparison to Prior Work. Our work differentiates itself by *co-designing* a BVH structure and a corresponding RT accelerator that takes advantage of efficient low-bit integer arithmetic hardware to operate *directly* on compressed data. Unlike prior methods that reduce memory traffic but still rely on FP32 arithmetic for traversal, our approach simultaneously minimizes both memory and computational overhead, achieving simultaneous gains in memory, performance, and energy efficiency. Since bounding boxes are fundamental to virtually all BVH-based RT systems, our technique remains largely orthogonal to prior work, thereby offering a more scalable path to enhancing ray tracing efficiency. Moreover, our proposed method also provide a gateway to generalize other potential applications structured as tree traversals through data quantization.

9 Conclusion

This paper introduces AQB8, a domain-specific accelerator leveraging our proposed multi-level quantization technique for BVH trees and addressing the persistent challenge of high computational and memory demands in ray tracing (RT). By structuring BVH nodes into clusters with high-precision anchor bounding boxes (BBs) and low-bit quantized BBs, our approach alleviates the performance degradation of RT when encoding BBs in the reduced precision format. Our proposed quantized BVH structure and the AQB8 hardware accelerator significantly reduce memory traffic and replace costly FP32 operations with integer arithmetic, achieving substantial energy efficiency. Our evaluations demonstrate that our AQB8 reduces DRAM accesses by 70%, energy consumption by 49%, and hardware area by 27%, yields a 1.82x performance speedup over modern GPU RT accelerators, thereby offering a more scalable and efficient RT accelerator for high-fidelity graphics rendering.

Acknowledgments

We thank MingLun Hsieh, Kai Chieh Hsu, and Yu Lun Ning for their valuable feedback. We also acknowledge the generous support provided by grants from MediaTek and the National Science and Technology Council (NSTC) in Taiwan (111-2221-E-A49-131-MY3).

References

[1] Advanced Micro Devices, Inc. 2023. "RDNA3" Instruction Set Architecture: Reference Guide. Retrieved Feb. 10, 2025 from https://www.amd.com/content/dam/amd/en/documents/radeon-tech-docs/instruction-set-architectures/rdna3-shader-instruction-set-architecture-feb-2023_0.pdf

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

[3] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 145–149.

[4] Arsène Pérard-Gayot. 2024. madmann91/bvh: A modern C++ BVH construction and traversal library. Retrieved Feb. 10, 2025 from https://github.com/madmann91/bvh

[5] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[6] Aaron Barnes, Fangjia Shen, and Timothy G Rogers. 2024. Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1027–1040.

[7] Carsten Benthin, Ingo Wald, Sven Woop, and Attila T. Áfra. 2018. Compressed-leaf bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 1–4.

[8] Nikolaus Binder and Alexander Keller. 2016. Efficient stackless hierarchy traversal on GPUs with backtracking in constant time.. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 41–50.

[9] Benedikt Bitterli. 2024. Rendering Resources | Benedikt Bitterli's Portfolio. Retrieved Feb. 10, 2025 from https://benedikt-bitterli.me/resources/

[10] Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. 2023. Treelet Prefetching For Ray Tracing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 742–755.

[11] Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. 2017. Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–41.

[12] Kirill Garanzha and Charles Loop. 2010. Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum* 29, 2 (2010), 289–298.

[13] Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 176–187.

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

[15] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M Aamodt. 2024. Generalizing Ray Tracing Accelerators for Tree Traversals on GPUs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1041–1057.

[16] Johannes Hanika and Alexander Keller. 2007. Towards Hardware Ray Tracing using Fixed Point Arithmetic. In *IEEE Symposium on Interactive Ray Tracing*. 119–128.

[17] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. 2011. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the Spring Conference on Computer Graphics*. 7–12.

[18] Jacob Haydel, Cem Yuksel, and Larry Seiler. 2023. Locally-Adaptive Level-of-Detail for Hardware-Accelerated Ray Tracing. *ACM Transactions on Graphics (TOG)* 42, 6 (2023), 1–15.

[19] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher Fletcher. 2018. UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 674–687.

[20] Seok Joong Hwang, Jaedon Lee, Youngsam Shin, Won-Jong Lee, and Soojung Ryu. 2015. A mobile ray tracing engine with hybrid number representations. In *Proceedings of the SIGGRAPH Asia Conference*. 1–4.

[21] Intel Corporation. 2023. Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games. Retrieved Feb. 10, 2025 from https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html/

[22] Timothy L Kay and James T Kajiya. 1986. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics* 20, 4 (1986), 269–278.

[23] Sean Keely. 2014. Reduced Precision for Hardware Ray Tracing in GPUs. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 29–40.

[24] Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2011. MRTP: Mobile Ray Tracing Processor With Reconfigurable Stream Multi-Processors for High Datapath Utilization. *IEEE Journal of Solid-State Circuits (JSSC)* 47, 2 (2011), 518–535.

[25] Hong-Yun Kim, Young-Jun Kim, Jie-Hwan Oh, and Lee-Sup Kim. 2013. A Reconfigurable SIMT Processor for Mobile Ray Tracing With Contention Reduction in Shared Memory. *IEEE Transactions on Circuits and Systems I: Regular Papers* 60, 4 (2013), 938–950.

[26] Tae-Joon Kim, Bochang Moon, Duksu Kim, and Sung-Eui Yoon. 2009. RACBVHs: Random-Accessible Compressed Bounding Volume Hierarchies. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* 16, 2 (2009), 273–286.

[27] D. Kopta, J. Spjut, E. Brunvand, and A. Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *IEEE International Conference on Computer Design (ICCD)*. 9–16.

[28] Samuli Laine. 2010. Restart trail for stackless BVH traversal. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 107–111.

[29] Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 137–143.

[30] Won-Jong Lee, Youngsam Shin, Seok Joong Hwang, Seok Kang, Jeong-Joon Yoo, and Soojung Ryu. 2015. Reorder buffer: an energy-efficient multithreading architecture for hardware MIMD ray traversal. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 21–32.

[31] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: a mobile GPU architecture for real-time ray tracing. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 109–119.

[32] Gábor Liktor and Karthikeyan Vaidyanathan. 2016. Bandwidth-efficient BVH layout for incremental hardware traversal.. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 51–61.

[33] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. 2021. Intersection Prediction for Accelerated GPU Ray Tracing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 709–723.

[34] Lufei Liu, Mohammadreza Saed, Yuan Hsi Chou, Davit Grigoryan, Tyler Nowicki, and Tor M. Aamodt. 2023. LumiBench: A Benchmark Suite for Hardware Ray Tracing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*. 1–14.

[35] Zihan Liu, Wentao Ni, Jingwen Leng, Yu Feng, Cong Guo, Quan Chen, Chao Li, Minyi Guo, and Yuhao Zhu. 2024. JUNO: Optimizing High-Dimensional Approximate Nearest Neighbour Search with Sparsity-Aware Algorithm and Ray-Tracing Core Mapping. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*. 549–565.

[36] Yashuai Lü, Libo Huang, Li Shen, and Zhiying Wang. 2017. Unleashing the power of GPU for physically-based rendering via dynamic ray shuffling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 560–573.

[37] Jeffrey Mahovsky and Brian Wyvill. 2006. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. In *Computer Graphics Forum*, Vol. 25. Wiley Online Library, 173–182.

[38] Durga Keerthi Mandarapu, Vani Nagarajan, Artem Pelenitsyn, and Milind Kulkarni. 2024. Arkade: k-Nearest Neighbor Search With Non-Euclidean Distances using GPU Ray Tracing. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS)*. 14–25.

[39] Daniel Meister, Jakub Boksansky, Michael Guthe, and Jiri Bittner. 2020. On Ray Reordering Techniques for Faster GPU Ray Tracing. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*. 1–9.

[40] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum* 40, 2 (2021), 683–712.

[41] Micron. 2024. Graphics memory | Micron Technology Inc. Retrieved Feb. 10, 2025 from https://www.micron.com/products/memory/graphics-memory

[42] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. 2010. Cache-oblivious ray reordering. *ACM Transactions on Graphics (TOG)* 29, 3 (2010), 1–10.

[43] Vani Nagarajan and Milind Kulkarni. 2023. RT-DBSCAN: Accelerating DBSCAN using Ray Tracing Hardware. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973.

[44] Vani Nagarajan, Durga Mandarapu, and Milind Kulkarni. 2023. RT-kNNS Unbound: Using RT Cores to Accelerate Unrestricted Neighbor Search. In *Proceedings of the 37th International Conference on Supercomputing (ICS)*. 289–300.

[45] Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, Dinesh Manocha, and Tack-Don Han. 2014. HART: A Hybrid Architecture for Ray Tracing Animated Scenes. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* 21, 3 (2014), 389–401.

[46] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. 2014. RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices. *ACM Transactions on Graphics (TOG)* 33, 5 (2014), 1–15.

[47] Jae-Ho Nah, Jeong-Soo Park, Jin-Woo Kim, Chanmin Park, and Tack-Don Han. 2010. Ordered depth-first layouts for ray tracing. In *ACM SIGGRAPH ASIA Sketches*. 1–2.

[48] Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I engine: traversal and intersection engine for hardware accelerated ray tracing. In *Proceedings of the SIGGRAPH Asia Conference*. 1–10.

[49] Paul Arthur Navrátil, Donald S Fussell, Calvin Lin, and William R Mark. 2007. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization. In *IEEE Symposium on Interactive Ray Tracing*. 95–104.

[50] Nvidia Corporation. 2023. NVIDIA ADA GPU ARCHITECTURE. Retrieved Feb. 10, 2025 from https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[51] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. mmp/pbrt-v4-scenes: Example scenes for pbrt-v4. Retrieved Feb. 10, 2025 from https://github.com/mmp/pbrt-v4-scenes

[52] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2023. *Physically based rendering: From theory to implementation.* MIT Press.

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

[54] Krishna Rajan, Soheil Hashemi, Ulya Karpuzcu, Michael Doggett, and Sherief Reda. 2020. Dual-precision fixed-point arithmetic for low-power ray-triangle intersections. *Computers & Graphics* 87 (2020), 72–79.

[55] Karthik Ramani, Christiaan P. Gribble, and Al Davis. 2009. StreamRay: a stream filtering architecture for coherent ray tracing. *ACM SIGARCH Computer Architecture News* 37, 1 (2009), 325–336.

[56] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. 2022. Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 263–281.

[57] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. 2002. SaarCOR: a hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. 27–36.

[58] Benjamin Segovia and Manfred Ernst. 2010. Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface*. 153–160.

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

[60] Siemens AG. 2024. Catapult High-Level Synthesis & Verification | Siemens Software. Retrieved Feb. 10, 2025 from https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/

[61] Siemens AG. 2024. Questa Advanced Simulator | Siemens Software. Retrieved Feb. 10, 2025 from https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[62] Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 28, 12 (2009), 1802–1815.

[63] Michael Steffen and Joseph Zambreno. 2010. Improving SIMT Efficiency of Global Rendering Algorithms with Architectural Support for Dynamic Micro-Kernels. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 237–248.

[64] Synopsys, Inc. 2024. Design Compiler: Timing, Area, Power, & Test Optimization | Synopsys. Retrieved Feb. 10, 2025 from https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[65] Synopsys, Inc. 2024. PrimePower: RTL to Signoff Power Analysis | Synopsys. Retrieved Feb. 10, 2025 from https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html

[66] Lorenzo Tessari, A Dittebrand, Michael J Doyle, and Carsten Benthin. 2023. Stochastic Subsets for BVH Construction. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, 255–267.

[67] The Khronos Group, Inc. 2024. Home | Vulkan | Cross platform 3D Graphics. Retrieved Feb. 10, 2025 from https://www.vulkan.org/

[68] TSMC. 2024. 40nm Technology - Taiwan Semiconductor Manufacturing Company Limited. Retrieved Feb. 10, 2025 from https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_40nm

[69] Karthikeyan Vaidyanathan, Tomas Akenine-Möller, and Marco Salvi. 2016. Watertight Ray Traversal with Reduced Precision. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 33–40.

[70] Karthik Vaidyanathan, Sven Woop, and Carsten Benthin. 2019. Wide BVH traversal with a short stack. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 15–19.

[71] Dietger Van Antwerpen. 2011. Improving SIMD efficiency for parallel Monte Carlo light transport on the GPU. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 41–50.

[72] Elena Vasiou, Konstantin Shkurko, Erik Brunvand, and Cem Yuksel. 2020. Mach-RT: A Many Chip Architecture for High Performance Ray Tracing. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* 28, 3 (2020), 1585–1596.

[73] Ingo Wald. 2007. On fast Construction of SAH-based Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*. 33–40.

[74] Ingo Wald. 2011. Active thread compaction for GPU path tracing. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 51–58.

[75] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–8.

[76] Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 434–444.

[77] Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of the Conference on High Performance Graphics (HPG)*. 1–13.

[78] Sung-Eui Yoon and Dinesh Manocha. 2006. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum* 25, 3 (2006), 507–516.