

A^2 RT: Efficient Ray Tracing Accelerator with Approximate-Accurate Computing and Quantization

Zhiyuan Zhang^{††}, Zhihua Fan^{††*}, Wenming Li^{††}, Yudong Mu^{††}, Yuhang Qiu^{††}, Zhen Wang^{††}
Xiaochun Ye^{††} and Xuejun An^{††}

[‡] SKLP, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China

[†] University of Chinese Academy of Sciences, Beijing, China

{zhangzhiyuan22s,fanzhihua,liwenming,muyudong23s,qiuyuhang21b,wangzhen21s,yexiaochun,axj}@ict.ac.cn

Abstract—Ray tracing (RT) has revolutionized photorealistic rendering by simulating light transport, but existing methods face a trade-off between computational efficiency and rendering accuracy. To address this, we present A^2 RT, a software-hardware co-designed RT accelerator employing the end to end optimization of "quantization \rightarrow computation". On the software side, we introduce a customized data flow mechanism with type-specific quantization for bounding boxes, ray origins, and directions, and we organize BVH nodes into Group- and Sub-Nodes. At the hardware level, a heterogeneous RT engine allocates resources based on node criticality: accurate computing units handle Group-Nodes, while approximate units process Sub-Nodes. A custom INT-FLOAT approximate multiplier further accelerates the approximate units. Experimental results show that A^2 RT achieves 45.51% energy consumption and $2.29\times$ speedup over RT Core, and consumes 81.79% of energy while delivering $1.57\times$ performance improvement compared to state-of-the-art accelerators.

Index Terms—Ray Tracing, Accelerator, Approximate Computing

I. INTRODUCTION

RAY tracing (RT), long regarded as the holy grail of computer graphics, has fundamentally redefined the paradigm of visual content synthesis through its simulation of light transport mechanisms [1]. It could bridge virtual and physical realities, serving critical demands across gaming systems [2], augmented reality (AR) [3], and metaverse applications [4]. A key acceleration structure to optimize RT is using a bounding volume hierarchy (BVH) tree [5]. BVHs work by grouping triangles within nested axis-aligned bounding boxes (AABBs). Instead of doing Ray-Triangle tests against every triangle, rays first test against these AABBs (Ray-Box intersection), allowing most of the triangles to be quickly skipped.

Modern GPUs (e.g., NVIDIA's RTX 4090 [6]) and custom ASIC-based accelerators [7] [8] enhance BVH traversal performance through dedicated RT hardware and parallel computation optimizations. However, they still incur substantial memory access and computational overheads. To reduce storage overhead, some approaches employ low-bit-width quantization for BVH nodes [9] [10] [11] [12]. Nevertheless, these methods require dequantizing data back to FP32 during intersection tests, thus still facing expensive FP32 computations. Other method [13] adopts multi-level quantization schemes that operate directly on low-bit data without dequantization. However, due to the

inadequate ray quantization method and inflexible strategy, the traversal overhead is increased and the accuracy is degraded.

A key insight from prior research confirms the promising potential of quantized BVH for accelerating RT. However, they face a trade-off between computational efficiency and rendering accuracy. This dilemma stems primarily from conventional quantization strategies lacking adaptivity to non-uniform geometry distributions and highly variable ray directions. To overcome this, we introduce A^2 RT, a software-hardware co-designed RT accelerator, our key contributions are as follows:

- 1) **Data flow mechanism for error-resilient BVH quantization (Software-level):** First, we classify errors induced by BVH quantization into two categories: **a)** redundant hits, **b)** missed hit errors. We propose a tailored data flow mechanism that employs type-specific quantization strategies for AABBs, ray origins and directions. Concurrently, it optimizes BVH traversal by integrating hierarchical quantization, which partitions BVH nodes into Group Nodes and Sub Nodes. This mechanism not only completely eliminates missed hit errors but also enhances computational efficiency while reducing redundant hits.
- 2) **Heterogeneous Ray-tracing Engine (Hardware-level):** To implement the proposed data flow, we design a heterogeneous ray-tracing engine that allocates computing resources based on node criticality: accurate Ray-Box units are dedicated to Group Node tests, while lightweight approximate computing units are employed for quantized Sub Nodes. In approximate unit, we have custom-developed an INT-FLOAT approximate multiplier, which is specifically optimized to comply with the quantization criteria defined by the data flow mechanism.
- 3) Experimental results show that A^2 RT achieves a 45.51% energy consumption and $2.29\times$ speedup over FP32 RT Core. Against the SOTA accelerator AQB8 [13], A^2 RT consumes only 81.79% of the energy while delivering a $1.57\times$ performance speedup.

II. BACKGROUND AND PROFILING

A. BVH Traversal in RT

1) **BVH Structure and Traversal:** As illustrated in Fig.1(a), a BVH structure features a root node encompassing the entire scene's AABBs, with subsequent recursive partitioning of

* Corresponding author: Zhihua Fan

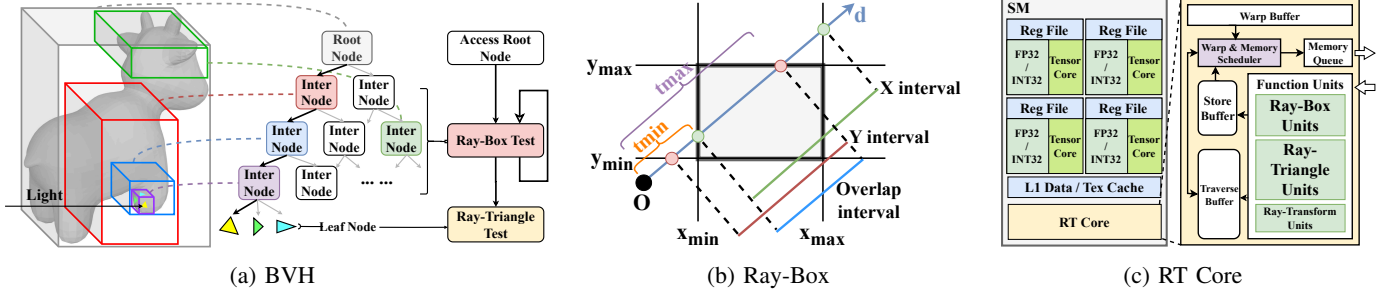


Fig. 1: Example of (a) BVH Tree structure and traversal, (b) Ray-Box intersection and (c) GPU RT Core architecture model.

geometric primitives into smaller bounding boxes comprising intermediate nodes. Leaf nodes contain the triangular facets in the lowest boxes. In BVH, ray traverse is performed through recursive examination of inter nodes (do Ray-Box tests) to identify the intersection of leaf nodes (do Ray-Triangle tests).

2) *Ray-Box Intersection Test*: The Ray-Box intersection test is the fundamental operation in RT, determining whether a ray intersects a bounding box. The most widely adopted approach is the slabs method [14]. A point in a ray can be defined as follows (o, d is the origin and direction of the ray):

$$p(t) = o + td \quad (1)$$

For AABBs, slabs employ a minimal representation using two coordinate points - $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$. Fig.1(b) illustrates this slab representation in the two-dimensional case, which can be generalized to three dimensions as follows:

$$\begin{aligned} t_{x0} &= (x_{min} - o) \times (1/d_x), \quad t_{x1} = (x_{max} - o) \times (1/d_x) \\ t_{xmin} &= \min(t_{x0}, t_{x1}), \quad t_{xmax} = \max(t_{x0}, t_{x1}) \end{aligned} \quad (2)$$

Here, t represents the signed distance from the ray origin to the intersection plane. According to the slabs method, a valid intersection between a ray and an AABB must satisfy the following two inequalities (1&2):

$$\begin{cases} t_{min} = \max(t_{xmin}, t_{ymin}, t_{zmin}) & \rightarrow 1: t_{min} < t_{max} \\ t_{max} = \min(t_{xmax}, t_{ymax}, t_{zmax}) & \rightarrow 2: t_{max} > 0 \end{cases}$$

3) *Ray-Triangle Intersection Test*: At leaf nodes of BVH tree, RT performs ray-triangle tests to identify potential ray-primitive intersections and calculate the precise hit points where such intersections occur. During BVH traversal, the number of Ray-Triangle tests is substantially smaller than Ray-Box tests.

B. GPU's RT Core Architecture

In NVIDIA's latest GPUs, dedicated ray tracing units (RT Cores) have been integrated into the Streaming Multiprocessors (SMs) [15]. GPU uses FP32 floating-point components. Fig.1(c) illustrates the architectural schematic of RT Core and the GPU ray tracing pipeline. Upon generation, each ray undergoes BVH traversal within the RT Core, followed by Ray-Box/Ray-Triangle tests. The ray's hit status is routed back to SMs for subsequent rendering operations.

C. RT's Computation and Memory Breakdown

We are profiling the RT Core's performance using Vulkan-Sim [16] with configurations in Table I across scenes from LumiBench [17] and others. Results are shown in Fig.2.

TABLE I: Vulkan-Sim Configurations

SMs, RT Cores	8
Max Warp / SM	32
Max Warp / RT	4
Warp Scheduler	GTO
Instruction Cache	128KB, 16-way
L1 Data Cache	64KB, LRU, Fully assoc
L2 Cache	3MB, 16-way, LRU
Compute Clock	1365 MHz

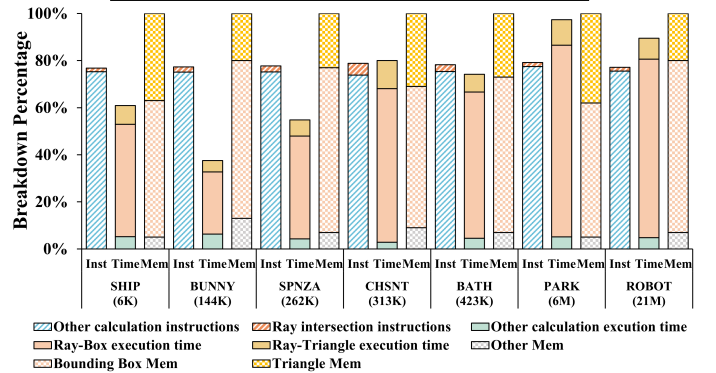


Fig. 2: Instruction number, execution time (cycles \times unit numbers) and L1 data cache memory breakdown of RT.

1) *Ray-Box Intersection Tests Dominate RT*: Fig.2 reveals a significant computational disparity: ray intersection instructions (Ray-Box and Ray-Triangle) account for merely 3.37% of total instructions on average, yet consume 67.23% of execution time. In ray intersection computations, Ray-Box tests account for an average of 86.37% of total execution time, significantly dominating the computational cost compared to Ray-Triangle tests. Therefore, accelerating Ray-Box tests is crucial for optimizing RT performance.

2) *BVH's Memory Overhead Slowdown RT*: As illustrated in Fig.2, BVH-related data occupies an average of 89.72% of the L1 data cache that directly feeds the RT Core. Within BVH structures, AABBs account for 70.13% of total BVH storage on average. Compressing AABB nodes can effectively reduce BVH memory overhead and improve RT efficiency.

Thus, BVH quantification can reduce memory access overhead and accelerate Ray-Box, which is of crucial importance.

III. DATA FLOW MECHANISM FOR BVH QUANTIZATION

Quantization approaches for AABB in BVH can be fundamentally categorized into two methodologies [9] [13]:

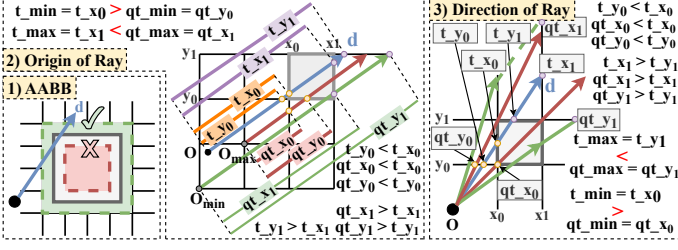


Fig. 3: Criteria for BVH quantification.

1) Global uniform quantization: This quantization method preserves the original coordinate system while compressing the original FP32 AABB coordinates to INT8, INT16, or FP16.

2) Multi-level quantization of AABBs: It divides the world space into subgroups, each maintaining an independent coordinate system. Within each subgroup, objects undergo coordinate transformation before global quantization in the subgroup.

Additionally, quantization can be extended to encompass ray origins and directions, beyond just AABBs, thereby enabling further optimization of RT. In this chapter, we present a comprehensive analysis of multi-level quantization to both rays and AABBs. The complete data flow is shown in Fig. 4.

A. Quantified Ray-Box Intersection Test

Let (x_0, y_0, z_0) denote the local coordinate frame origin within a quantization subgroup. Given: Original AABB coordinates: (x_1, y_1, z_1) ; Ray origin: O ; Ray direction: D ; Scaling factor: S, SO, SD . The quantized coordinates are derived as follows (D_x is the scalar components of vector D):

$$\begin{bmatrix} q_{x1} \\ q_{y1} \\ q_{z1} \end{bmatrix} = \begin{bmatrix} (x_1 - x_0) \times S_x \\ (y_1 - y_0) \times S_y \\ (z_1 - z_0) \times S_z \end{bmatrix}$$

$$\begin{bmatrix} q_{Ox} \\ q_{Oy} \\ q_{Oz} \end{bmatrix} = \begin{bmatrix} (O_x - x_0) \times SO_x \\ (O_y - y_0) \times SO_y \\ (O_z - z_0) \times SO_z \end{bmatrix}, \begin{bmatrix} q_{(1/D_x)} \\ q_{(1/D_y)} \\ q_{(1/D_z)} \end{bmatrix} = \begin{bmatrix} 1/(D_x \times SD_x) \\ 1/(D_y \times SD_y) \\ 1/(D_z \times SD_z) \end{bmatrix}$$

By substituting the quantified data into formula (2), the quantified t , which is q_t can be obtained:

$$q_{t_{x1}} = (q_{x1} - q_{Ox}) \times q_{(1/D_x)} = \frac{(x_1 - x_0)S_x - (O_x - x_0)SO_x}{D_x \times SD_x} \quad (3)$$

To ensure that the quantized results do not alter the original magnitude relationships, q_t must be equal to t multiplied by scaling factor $Scale$:

$$q_{t_{x1}} = Scale \times t_{x1} = Scale \times (x_1 - O_x) \times \frac{1}{D_x} \quad (4)$$

From the preceding formulation (3) and (4), we derive two critical quantization constraints:

- The scaling factor for AABB coordinates must equal the scaling factor applied to ray origin: $S = SO$.
- The spatial and directional scaling ratio must remain uniform across all axes: $S_x/SD_x = S_y/SD_y = S_z/SD_z$.

B. Errors in Ray-Box Tests Brought by BVH Quantification

Due to the reduced precision of the quantized BVH, errors may occur during Ray-Box tests. We define the following terminology: Hit (intersection result using the original BVH)

and Q_hit (intersection result using the quantized BVH). We classify quantization-induced errors into two categories:

- **Redundant Hits:** $Hit = False, Q_hit = True$. It introduces unnecessary Ray-Box or Ray-Triangle tests but does not affect the final rendering correctness. As it only impacts performance, this appearance is acceptable.
- **Missed Hit Errors:** $Hit = True, Q_hit = False$. This error leads to incorrect rendering results, as a potentially valid intersection is missed, affecting shading computations for the corresponding primitive. As it introduces visual artifacts and errors, this type of error is unacceptable.

To ensure correctness, the quantization scheme must enforce constraints that strictly prevent missed hit errors, while tolerating (but ideally minimizing) redundant hits.

C. Criteria for BVH Quantification

To completely eliminate missed-hit errors, distinct quantization strategies must be applied to AABBs, ray origins, and ray directions, as illustrated in the 2D example in Fig.3. The key requirement is ensuring that the quantized intersection bounds satisfy $qt_{min} \leq t_{min}, qt_{max} \geq t_{max}$. This guarantees that the quantized Ray-Box test remains conservative, never missing valid intersections and there will be no missed hit errors. Below, we detail the quantization strategies for each component:

- **AABB:** The quantized AABB must fully enclose the original AABB (green box in Fig.3(1)). If the quantized AABB is smaller than original AABB (red box), missed hit errors occur.
- **Ray Origin:** After quantizing the ray origin, two values are generated: O_{max} (result of ceiling operation) and O_{min} (result of floor operation). As shown in Fig.3(2), qt_{max} is calculated using O_{min} (green ray), while qt_{min} is calculated using O_{max} (red ray). This approach effectively avoids missed hit errors.
- **Ray Direction:** For the ray direction, each component undergoes both ceiling and floor operations during quantization. As illustrated in Fig.3(3), when calculating the min and max values, we use different quantized rays. For computational convenience, we quantize the inverse direction ($1/D$) in practical implementation. Specifically, we use the floor operation of $1/D$ (red rays, ceiling for D) to compute qt_{min} and the ceiling operation of $1/D$ (green rays, floor for D) to calculate qt_{max} .

D. BVH Nodes Classification Method

This work adopts multi-level quantization for BVHs: specific AABBs are selected as references and designated as Group Nodes, after which the following formula is applied to each AABB (child_node) within a Group Node to determine whether it should be quantized to a Sub Node (as detailed in Section III.A) or designated as a new Group Node.

$$Q(child_node) \text{ if } \begin{cases} S(c)/S(G) \geq S_t \\ L(c)/L(G) \geq L_t \end{cases} \quad (5)$$

Let S denote the surface area of an AABB, and L denote the length of an AABB along the x, y, z axes. Quantization is only applied to a child node if the ratios of its surface

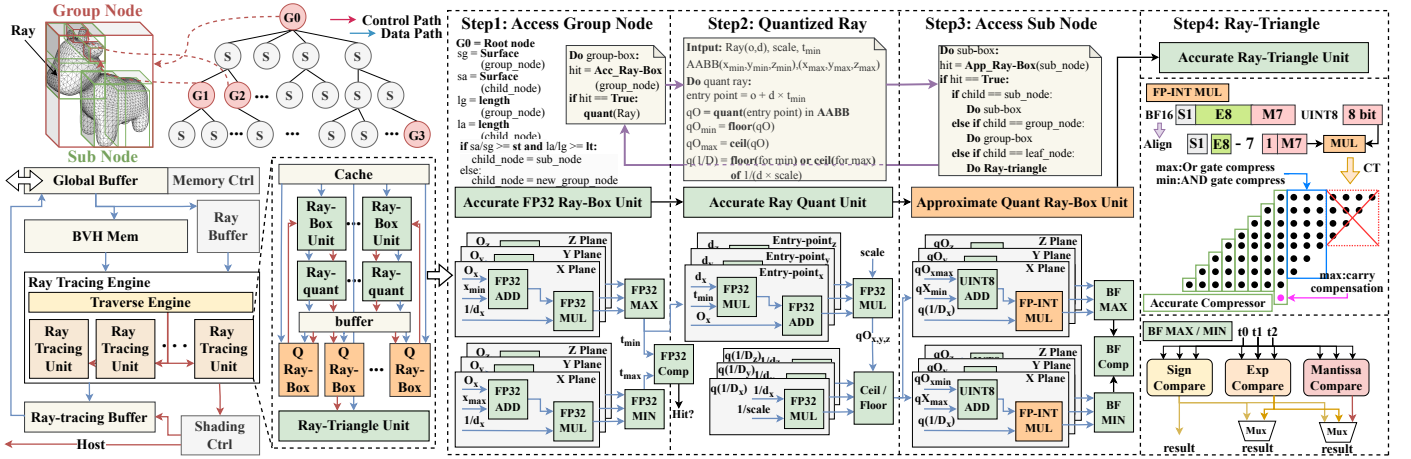


Fig. 4: The overall architecture and data flow of A^2RT .

area and its length (along each axis) to the Group Node, exceed the thresholds S_t and L_t (User can customize). Since the Surface Area Heuristic (SAH) [18]—a standard method for BVH construction—already involves calculating the surface area and coordinates of each node, the proposed method does not introduce significant additional computational overhead. Time complexity of BVH classification is $O(n \log n)$.

A key distinction from prior works is that the proposed method does not require modifying the BVH structure—instead, it only needs to add identifiers to original BVH nodes to distinguish between Group Nodes and Sub Nodes. Meanwhile, the node classification approach proposed in this work can be completed during the BVH tree construction process, meaning a BVH with classified nodes can be generated directly. This eliminates the need for post-construction node clustering and separation (a step required in [13]), thereby reducing the overhead associated with BVH construction.

In this work, Sub Nodes AABBs within Group Node and the ray origins are quantized to the UINT8, $SD = S = 255/l$, $l = (l_x, l_y, l_z)$ is the length vector of Group Node in x, y, z axes. The inverse directions $(1/D)$ are quantized to BF16.

IV. PROPOSED A^2RT HARDWARE ARCHITECTURE

To implement the proposed data flow, we design a dedicated architecture illustrated in Fig.4. The entire execution process begins with off-chip data input: the quantized BVH tree and all triangular primitive information are stored in the Global Buffer, while ray information is held in the Ray Buffer. Ray Tracing Engine retrieves the data it requires from the BVH Memory. Upon completion of ray tracing, the Shading Control unit determines whether the process should terminate; the results are then stored in the Ray-Tracing Buffer, and finally transmitted to the host via the Global Buffer.

A. Heterogeneous Ray-tracing Engine

The traversal of BVH trees constitutes the most time and energy-consuming operation in ray tracing. Its core computational operators include Ray-Box and Ray-Triangle intersection tests, among which Ray-Box tests account for the majority of computational overhead (see Section II.C). To address this challenge, the heterogeneous ray tracing engine proposed in

this work comprises four key components: accurate Ray-Box units, ray quantization units, accurate Ray-Triangle units, and approximate quantized Ray-Box units that employ approximate multiplication components.

- 1) **Accurate Ray-Box Unit:** This unit is designed to determine whether a ray intersects a Group Node and compute an accurate t_{min} for ray quantization calculations. To ensure result precision, all computations within this unit are performed using the FP32 format.
- 2) **Accurate Ray Quantization Unit:** This unit is dedicated to ray quantization. An accurate entry point can be computed using the precise t_{min} . Since a ray is defined as a semi-infinite line, its origin can be shifted along the direction. The entry point could be treated as the ray's origin within the current Group Node. Both the ray direction and this entry point are then quantized as described in Section III: directions use BF16 (8-bit exponent) to prevent missed hits, while origins and AABBs are quantized to UINT8 integers.
- 3) **Approximate Quantized Ray-Box Unit:** This unit performs Ray-Box tests using quantized AABBs and rays for Sub Nodes. We have designed a custom approximate multiplier that supports $INT \times FLOAT$, with the structure of its compression tree illustrated in Fig.4. To enable direct integer-mantissa multiplication, we first perform a pre-alignment step, the compression tree is used to compute the product of expanded mantissas and integers. For less significant bits, we directly discard them or implement OR gates (for t_{max}) and AND gates (for t_{min}) for approximate compression. Additionally, carry compensation is applied for t_{max} calculation to avoid missed hit errors. For more significant bits, we use exact compressors to ensure computation accuracy. After multiplication, the product must be normalized back to a floating-point number. For MAX and MIN comparison, we perform parallel comparison on the sign bit, exponent bits, and mantissa bits of floating-point numbers to accelerate the comparison process.
- 4) **Accurate Ray-Triangle Unit:** This unit performs Ray-Triangle tests. Since both the intersection status (whether

a ray intersects a triangle) and the intersection point are critical to subsequent rendering processes, all computations within this unit are executed using the exact FP32 floating-point format to prevent errors.

B. Memory Layout

For storage, the BVH is stored using a breadth-first strategy, where nodes located at the same level are stored contiguously. Nodes in the multi-level quantized BVH fall into two types: Group Nodes and Sub Nodes, as detailed in Fig.5.

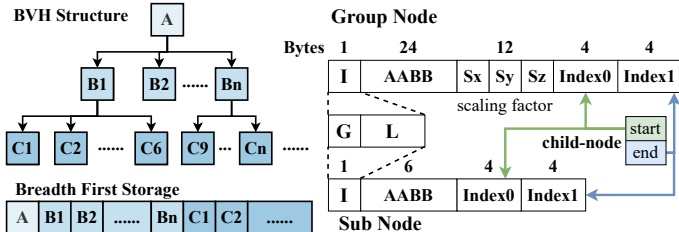


Fig. 5: The detailed memory layout of A^2RT

Group and Sub nodes adopt a distinct representation format:

- **Group Node:** The AABB of a Group Node is stored in the FP32 format, occupying 24 bytes. The scaling factors for x, y, z three axes (used for quantifying ray) are represented in FP32, taking up 12 bytes in total. Additionally, the Group Node includes the starting and end addresses of its child nodes (8 bytes total), as well as a 1-byte identifier. For the identifier in a Group Node, the G bit is set to $1'b1$, while the L field is configured as $7'b00000000$.
- **Sub Node:** Each Sub Node stores its AABB in 6 bytes using UINT8 format, along with 8 bytes for the start and end addresses of its children and a 1-byte identifier. For the identifier in an Sub Node, the G bit is set to $1'b0$, while the L field is configured as either $7'b00000000$ (indicating an AABB-associated Sub Node) or $7'b11111111$ (indicating a leaf node). If the Sub Node is designated as a leaf node, it signifies that the corresponding bounding box contains triangular primitives.

When a node is fetched, its neighboring nodes in the same group are concurrently loaded into the cache, enhancing memory access efficiency since such nodes are typically accessed sequentially during BVH traversal [19].

V. EVALUATION OF A^2RT

A. Experiment Setup

We implemented A^2RT using HDL and synthesized it using Synopsys Design Compiler (DC) at 12nm GP standard VT library to synthesize the data of area and power, the version of the DC tool is Q-2019.12-SP2. Table II shows the hardware parameters. Two configurations were compared:

- **Baseline:** Employing FP32 precision RT Core.
- **AQB8:** A state-of-the-art (SOTA) accelerator [13].

In experiments, we employed the open-source Vulkan-sim [16] simulator with configurations specified in Table I for benchmarking against RT Core and AQB8. The test cases were derived from the LumiBench [17], containing triangles of varying scales ($6K \sim 21M$). For fair comparison, we use 6-wide BVH tree proposed by Intel Embree [20].

TABLE II: A^2RT hardware parameters

	Area (mm^2)	Power (W)
Global Buffer (2MB)	1.66 (29.96%)	0.87
Other Mem (BVH etc.) 1MB	0.93 (16.78%)	0.53
Control Logic	0.64 (11.55%)	0.16
128KB Cache	0.72 (12.99%)	0.27
Ray Tracing Unit 120 QBOX, 15 BOX, 15 Ray, 24 TRI	1.26 (22.74%)	1.48
Othes	0.33 (5.96%)	0.08
Total	5.54	3.39
Compare QBOX's computing part	Area (μm^2)	Power (mW)
AQB8	1668.72	3.31
A^2RT	973.53	1.89

B. Evaluation of Errors in Ray-Box Tests of Quantized BVH

To evaluate the errors introduced by quantization, we conducted tests using a large set of random ray directions and AABBs. For fair comparison, A^2RT quantified AABBs and ray origins to UINT8 (like AQB8), ray directions were represented in BF16. Test results are shown in Fig. 6.

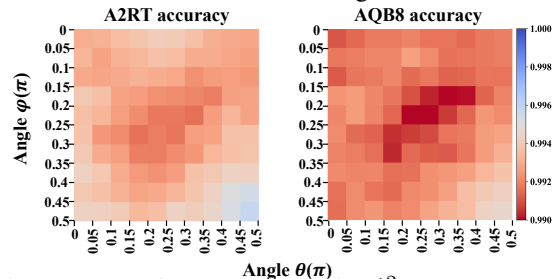


Fig. 6: Accuracy of Ray-Box tests in A^2RT and AQB8 under different ray directions compared with baseline

Since ray directions are represented as unit vectors, which are determined by two angles (θ, ϕ), we select angles in the range of $0 \rightarrow 0.5\pi$ as examples; the other angle ranges follow a similar trend. **1) For Redundant Hits**, the error rate of A^2RT is 15.44% lower than AQB8. This is because AQB8 employs a fused multiply-add (FMA)-based Ray-Box calculation—a method that has been shown in prior work [21] to introduce substantial errors. Furthermore, A^2RT incorporates three scaling factors, enabling more refined and flexible scaling for x, y, z axes of AABBs, whereas AQB8 relies on a single scaling factor. **2) For Missed Hit errors**, A^2RT completely eliminates this error. In contrast, due to the ray quantization scheme adopted by AQB8 (use 5e8m format for direction and using INT32 instead of FP), errors may arise when ray directions are close to the x, y, z axes.

C. Comparative Analysis of A^2RT

1) BVH Build and Clustering Time: Fig. 7(a) presents a comparison of BVH construction and node classification time consumed by AQB8 and proposed A^2RT . AQB8 can only perform node classification after the entire BVH tree has been constructed, resulting in an algorithmic time complexity of $O(n(\log n)^2)$. In contrast, the classification method proposed in this work can be executed concurrently with BVH construction, achieving a reduced time complexity of $O(n \log n)$. On average, the total time consumed by A^2RT for BVH construction and classification is only 38.97% of AQB8.

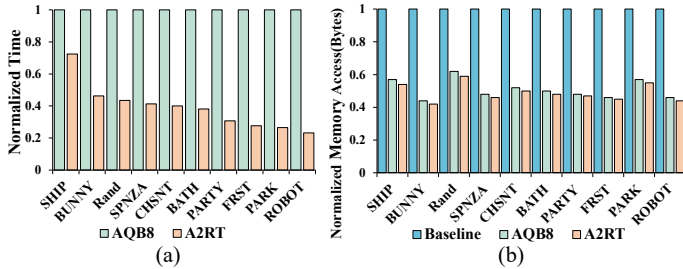


Fig. 7: Normalized BVH build and clustering time (a) and memory access bytes (b) of Baseline, AQB8 and A^2RT .

2) **Memory Traffic:** Fig. 7(b) illustrates the access volume of L1 data cache (measured in bytes). In the original RT core, all BVH nodes are represented using 56 bytes. AQB8 partitions nodes into anchor nodes (36 bytes) and regular nodes (16 bytes). Our work, A^2RT , classifies nodes into Group Nodes (45 bytes) and Sub Nodes (15 bytes). Our Group Nodes require more space than AQB8’s anchor nodes because we allocate more bits to store scaling factors—enabling independent scaling for the x, y, z axes, which achieves more flexible and accurate quantization. In contrast, AQB8 enforces a uniform scaling factor across all three axes. Furthermore, since the vast majority of nodes in the classified BVH are Sub Nodes, our overall memory access volume is slightly lower than AQB8. On average, AQB8’s access volume is 51.21% of the baseline, while ours A^2RT is 49.83% of the baseline.

3) **QBOX Unit:** Quantized Ray-Box unit (referred to as QBOX in Table II) serves as the most critical component in quantized RT accelerators, consuming substantial on-chip resources and energy. Under the same manufacturing process and at 1 GHz frequency, we synthesized and compared the area and power of QBOX’s computing part from AQB8 and A^2RT . The results are presented in Table II. AQB8 employs a FMA based computation approach, with the final result represented in INT32. In contrast, our design first performs INT8 subtraction operations, followed by computing the result in BF16 format using the custom-designed INT-FLOAT approximate multiplier. QBOX proposed in A^2RT ’s area is 58.34% of AQB8’s, and power consumption is 57.09% of AQB8’s.

4) **Energy Consumption:** A^2RT achieves reductions in both computational energy and memory access energy. This is attributed to two key factors: first, most Ray-Box intersection operations are processed using low-bitwidth arithmetic, which reduces the dynamic power consumption of hardware units; second, the quantized data representation lowers memory access energy compared to the baseline. On average, the total energy consumed by A^2RT is 45.51% of that consumed by the baseline and 81.79% of that consumed by AQB8.

5) **Speedup for BVH Traversal:** A^2RT achieves performance improvements compared to both the Baseline and AQB8. The performance improvement of A^2RT over AQB8 primarily stems from two factors: first, its lower Ray-Box intersection error rate; second, the QBOX unit in A^2RT incorporates approximate computing and more optimized quantization schemes, enabling a shorter critical path delay than that of AQB8. Specifically, it delivers an average speedup of $2.29\times$ relative to the baseline and $1.57\times$ over AQB8.

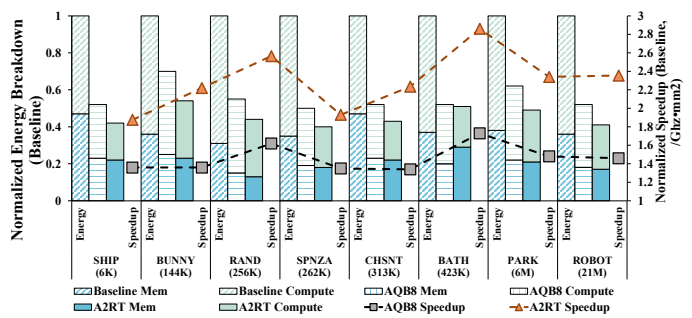


Fig. 8: Normalized speedup (in line) and energy breakdown (in bar) of Baseline, AQB8 and A^2RT .

TABLE III: Comparison of A^2RT and other accelerators

	Multi-level AABB Quant	Ray Origin Quant	Direction Quant	Low-bit Arithmetic	Approximate Computing
[9]	✗	✗	✗	✗	✗
[10]	✓	✗	✗	✗	✗
[13]	✓	✗	✗	✓	✗
A^2RT	✓	✓	✓	✓	✓

VI. RELATED WORKS

In prior BVH quantization works [9] [10], a decompression step is required for Ray-Box, followed by FP32 computations. A key limitation of these works is their one-sided focus on storage reduction rather than end-to-end optimization of “quantization \rightarrow computation”. Since they use FP32 without quantizing rays, no missed hit error occur. AQB8 [13] performs computations directly using quantized data. However, it uses a single scaling factor for all axes, inappropriate ray quantification and FMA for Ray-Box, it increases traversal overhead and degrades accuracy. Comparison shows in Table III.

In contrast, the proposed A^2RT in this work introduces end-to-end optimization. It employs distinct quantization strategies for rays and AABBs, completely eliminating missed hit errors while further reducing Ray-Box intersection error rate. Furthermore, a specialized INT-FLOAT approximate multiplier is designed to further optimize computations and efficiency.

VII. CONCLUSION

This paper presents A^2RT , a software-hardware co-designed RT accelerator employing end-to-end optimization of “quantization \rightarrow computation”. On software front, we propose a customized data flow mechanism integrated with type-specific quantization strategies, tailored for bounding boxes, ray origins and directions. At hardware level, we design a heterogeneous RT engine that allocates computing resources based on node criticality: accurate units handle Group-Node, while approximate units for Sub-Node. Across different test scenarios, A^2RT achieves both energy efficiency (45.51% of baseline and 81.79% of AQB8) and performance improvement ($2.29\times$ over baseline and $1.57\times$ over AQB8).

VIII. ACKNOWLEDGMENT

This work was supported by National Key R&D Program of China (Grant No.2023YFB4503500), National Natural Science Foundation of China (Grant No.62502498), Beijing Natural Science Foundation (Grant No.L234078). We sincerely thank Li Ning for his precious feedback and contribution.

REFERENCES

- [1] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 4th ed. MIT Press, 2023. [Online]. Available: <https://www.pbr-book.org/>
- [2] Epic Games, Inc., “Unreal engine 5 documentation,” <https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-5-documentation>, 2025, accessed: 2025-7-21.
- [3] S. Guo, S. S. Sapatnekar, and J. Gu, “Software-hardware codesign of ray-tracing accelerator for edge ar/vr with viewpoint-focused 3d construction and efficient data structure,” in *2024 IEEE 67th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2024, pp. 267–271.
- [4] S.-M. Park and Y.-G. Kim, “A metaverse: Taxonomy, components, applications, and open challenges,” *IEEE Access*, vol. 10, pp. 4209–4251, 2022.
- [5] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, “A Survey on Bounding Volume Hierarchies for Ray Tracing,” *Computer Graphics Forum*, 2021.
- [6] NVIDIA Corporation. (2023) Nvidia ada gpu architecture. Updated to include information on NVIDIA L40 and L4 Data Center GPUs. [Online]. Available: <https://images.nvidia.cn/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf>
- [7] R. Yan, Y. Su, H. Guo, Y. Lü, J. Wang, N. Xiao, L. Shen, Y. Wang, and L. Huang, “Mprta: An efficient multilevel parallel mobile accelerator for high-performance ray tracing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 2, pp. 396–400, 2024.
- [8] E. Vasiou, K. Shkurko, E. Brunvand, and C. Yuksel, “Mach-rt: A many chip architecture for high performance ray tracing,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 3, pp. 1585–1596, 2022.
- [9] C. Benthin, I. Wald, S. Woop, and A. T. Áfra, “Compressed-leaf bounding volume hierarchies,” in *Proceedings of the Conference on High-Performance Graphics*, ser. HPG ’18. New York, NY, USA: Association for Computing Machinery, 2018.
- [10] H. Ylitiö, T. Karras, and S. Laine, “Efficient incoherent ray traversal on gpus through compressed wide bvhs,” in *Proceedings of High Performance Graphics*, ser. HPG ’17. New York, NY, USA: Association for Computing Machinery, 2017.
- [11] S. Keely, “Reduced precision for hardware ray tracing in gpus,” ser. HPG ’14. Goslar, DEU: Eurographics Association, 2014, p. 29–40.
- [12] G. Liktov and K. Vaidyanathan, “Bandwidth-efficient bvh layout for incremental hardware traversal,” in *Proceedings of High Performance Graphics*, ser. HPG ’16. Goslar, DEU: Eurographics Association, 2016, p. 51–61.
- [13] Y.-C. Huang, C.-P. Yang, and T. T. Yeh, “Aqb8: Energy-efficient ray tracing accelerator through multi-level quantization,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 374–387.
- [14] T. L. Kay and J. T. Kajiya, “Ray tracing complex scenes,” in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’86. New York, NY, USA: Association for Computing Machinery, 1986, p. 269–278.
- [15] D. Ha, L. Liu, Y. H. Chou, S. Go, W. W. Ro, H.-W. Tseng, and T. M. Aamodt, “Generalizing ray tracing accelerators for tree traversals on gpus,” in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024, pp. 1041–1057.
- [16] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, “Vulkan-sim: A gpu architecture simulator for ray tracing,” in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 263–281.
- [17] L. Liu, M. Saed, Y. H. Chou, D. Grigoryan, T. Nowicki, and T. M. Aamodt, “Lumibench: A benchmark suite for hardware ray tracing,” in *2023 IEEE International Symposium on Workload Characterization (IISWC)*, 2023, pp. 1–14.
- [18] I. Wald, “On fast construction of sah-based bounding volume hierarchies,” in *2007 IEEE Symposium on Interactive Ray Tracing*, 2007, pp. 33–40.
- [19] Y. H. Chou, T. Nowicki, and T. M. Aamodt, “Treelet prefetching for ray tracing,” in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 742–755.
- [20] Intel Corporation, “Embree: High performance ray tracing kernels,” <https://www.embree.org/>, Intel Corporation, 2025.
- [21] T. Ize, “Robust BVH ray traversal,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 2, pp. 12–27, July 2013.