GRTX: Efficient Ray Tracing for 3D Gaussian-Based Rendering

Junseo Lee   Sangyun Jeon   Jungi Lee   Junyong Park   Jaewoong Sim

Seoul National University
{junseo.lee, sangyun.jeon, jungi.lee, junyong.park, jaewoong}@snu.ac.kr

Abstract—3D Gaussian Splatting has gained widespread adoption across diverse applications due to its exceptional rendering performance and visual quality. While most existing methods rely on rasterization to render Gaussians, recent research has started investigating ray tracing approaches to overcome the fundamental limitations inherent in rasterization. However, current Gaussian ray tracing methods suffer from inefficiencies such as bloated acceleration structures and redundant node traversals, which greatly degrade ray tracing performance.

In this work, we present GRTX, a set of software and hardware optimizations that enable efficient ray tracing for 3D Gaussian-based rendering. First, we introduce a novel approach for constructing streamlined acceleration structures for Gaussian primitives. Our key insight is that anisotropic Gaussians can be treated as unit spheres through ray space transformations, which substantially reduces BVH size and traversal overhead. Second, we propose dedicated hardware support for traversal checkpointing within ray tracing units. This eliminates redundant node visits during multi-round tracing by resuming traversal from checkpointed nodes rather than restarting from the root node in each subsequent round. Our evaluation shows that GRTX significantly improves ray tracing performance compared to the baseline ray tracing method with a negligible hardware cost.

I. INTRODUCTION

The recent advent of 3D Gaussian Splatting (3DGS) [20] is rapidly transforming how we reconstruct and render 3D scenes across a wide range of applications, including robotics, AR/VR, gaming, and interactive media [19], [44], [45], [56]. By representing a scene using a set of anisotropic Gaussians that can be rendered through rasterization, 3DGS effectively captures fine geometric and appearance details while generating photorealistic novel views at significantly higher speeds than prior methods such as NeRF [34].

While Gaussian primitives can, in principle, be rendered via ray tracing, 3DGS capitalizes on the computational efficiency of rasterization to achieve real-time performance. However, rasterization-based rendering struggles to accurately render scenes captured with highly distorted cameras [35]—essential for domains such as robotics and autonomous vehicles— and fails to faithfully reproduce complex lighting effects that depend on secondary rays, including reflections, refractions, and shadows.

To address these limitations, recent research from industry-leading companies such as Google, NVIDIA, and Meta has explored ray tracing for Gaussian scenes [10], [32], [35]. Unfortunately, however, current Gaussian ray tracing methods suffer from inefficiencies and fall short of 3DGS in terms of performance. While most methods exploit hardware-accelerated ray-triangle intersection testing by employing bounding mesh proxies for Gaussian primitives, this substantially inflates the size of the bounding volume hierarchy (BVH) and increases memory requirements during BVH traversal. Furthermore, the multi-round tracing method commonly used in prior work results in redundant node visits and intersection testing across tracing rounds, thereby further decreasing traversal efficiency.

In this paper, we present GRTX, a collection of software and hardware optimizations that greatly improve ray tracing efficiency for Gaussian-based rendering. First, we point out that the existing approach to creating acceleration structures— constructing individual bounding proxy geometries for each Gaussian and building a single monolithic BVH—is both naïve and inefficient, and that we can actually build acceleration structures that are far more efficient.

For this, we exploit a fundamental geometric insight: anisotropic Gaussian primitives can be uniformly represented as *unit spheres* through ray space transformations. Importantly, these transformations can be performed natively by modern ray tracing hardware at the leaf (instance) nodes within the top-level acceleration structure (TLAS) of a two-level BVH. Leveraging this insight, we utilize a two-level acceleration structure for Gaussian ray tracing while constructing only a single, *shared* bottom-level acceleration structure (BLAS) containing unit sphere geometry. All Gaussian primitives then reference the same BLAS in the TLAS, thereby greatly reducing the BVH memory footprint and traversal overhead compared to previous ray tracing methods.

Second, we propose enhancing ray tracing hardware with checkpointing and replay capabilities. In multi-round ray tracing, BVH nodes intersected by rays in a given round may fall outside that round's traversal interval, deferring traversal into their subtrees to subsequent rounds. To visit their descendants, however, the paths from the root to these nodes need to be redundantly retraced. Our key idea is to checkpoint these nodes during the current round and resume traversal directly from them in the next round, rather than restarting from the root. This eliminates redundant node visits and intersection tests while also avoiding unnecessary processing of Gaussians that have already been found *and* blended in earlier rounds.

We evaluate GRTX using Vulkan-Sim [47], a cycle-level graphics simulator for ray tracing applications, augmented
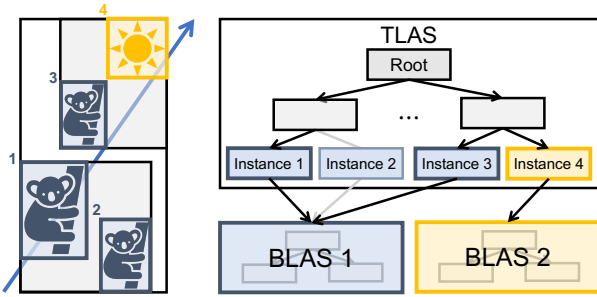
Fig. 1: 2D visualization of Bounding Volume Hierarchy (BVH) and ray traversal.

Fig. 2: Ray tracing pipeline.

with our in-house ray tracing simulator that replaces the original ray tracing module. Our results show that GRTX, combining software and hardware optimizations, achieves an average speedup of $4.36\times$ over the baseline employing an icosahedron bounding mesh. We also evaluate GRTX-SW, a software-only optimization, on a commodity GPU, where it achieves speedups of 1.44–$2.15\times$ across the evaluated scenes. In summary, this paper makes the following contributions:

- To our knowledge, this is the first work to identify the challenges and inefficiencies of rendering Gaussian primitives via ray tracing.
- We present an approach for building efficient acceleration structures tailored to Gaussian primitives, which leads to substantial reductions in BVH size and traversal cost.
- We propose checkpointing and replay capabilities for ray tracing hardware, which help eliminate redundant BVH traversal and intersection testing during multi-round Gaussian ray tracing.

II. BACKGROUND

In this section, we first provide background on ray tracing and its acceleration hardware in modern GPUs. We then briefly introduce the state-of-the-art method for representing and rendering complex 3D scenes using a set of Gaussians.

A. Ray Tracing

Ray tracing is a rendering technique that simulates the path of light from the viewer to light sources in a 3D scene. Unlike rasterization-based rendering, which projects scene primitives (e.g., triangles) onto the image plane and identifies which pixels they cover, ray tracing approaches the graphics rendering from the opposite direction—it begins with rays and determines which primitives these rays intersect. This allows us to naturally simulate the physical behavior of light, thereby enabling more accurate rendering of complex optical effects such as reflections, refractions, and shadows.

Bounding Volume Hierarchy. Since naïvely testing the intersection between each ray and all the primitives is prohibitively costly, a spatial data structure called an acceleration structure (AS) is typically used to reduce the number of intersection tests. The AS organizes scene primitives based on their spatial positions, which allows ray tracers to efficiently skip unnecessary intersection tests. One of the most widely used
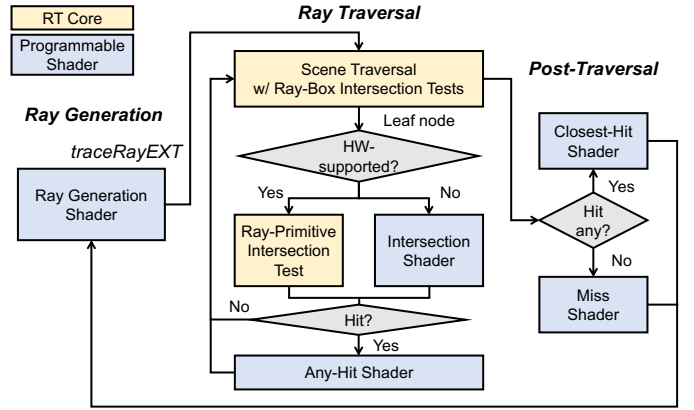
acceleration structures is a bounding volume hierarchy (BVH), a tree-based data structure where each parent node *spatially* encloses its child nodes. This hierarchical relationship provides significant optimizations—when a ray misses a parent node, all child nodes can be immediately excluded from further testing.

Figure 1 illustrates an example of ray traversal through a BVH, where each internal node represents an axis-aligned bounding box (AABB). The traversal begins at the root bounding box (Root) and proceeds through the hierarchy while performing *ray-box* intersection tests to prune unnecessary branches. When the ray reaches a leaf node containing scene primitives, *ray-primitive* intersection tests are conducted and determine the actual hit point.

In complex scenes with instanced objects, the BVH can be organized as a two-level hierarchy consisting of a *Top-Level Acceleration Structure* (TLAS) and *Bottom-Level Acceleration Structures* (BLAS) [21]. The TLAS serves as the upper level, containing references to BLAS instances as its leaf nodes. Each BLAS typically represents a distinct object or mesh. When traversal reaches a TLAS leaf, the ray is transformed into the local coordinate space of a BLAS instance using a stored transformation matrix, then traversal continues within that BLAS. Since the BLAS of a single object can be shared across multiple instances, this two-level approach significantly reduces memory usage for scenes with repeated geometry.

Ray Tracing Accelerators. Modern GPUs now feature dedicated hardware accelerators to enhance ray tracing performance, such as NVIDIA RTX's RT Cores [7], AMD RDNA's Ray Tracing Accelerators [1], and Intel Arc's Ray Tracing Units [2]. While specific implementations and supported features vary across hardware vendors and architecture generations, these accelerators commonly include ray-box and ray-triangle intersection test units, thereby alleviating the main performance bottleneck in ray tracing. In addition, many accelerators optimize the overall tree traversal process while autonomously managing traversal stacks, fetching nodes, and handling ray transformations between TLAS and BLAS.

Ray Tracing Pipeline. Figure 2 shows an overview of the typical ray tracing pipeline supported by standard graphics
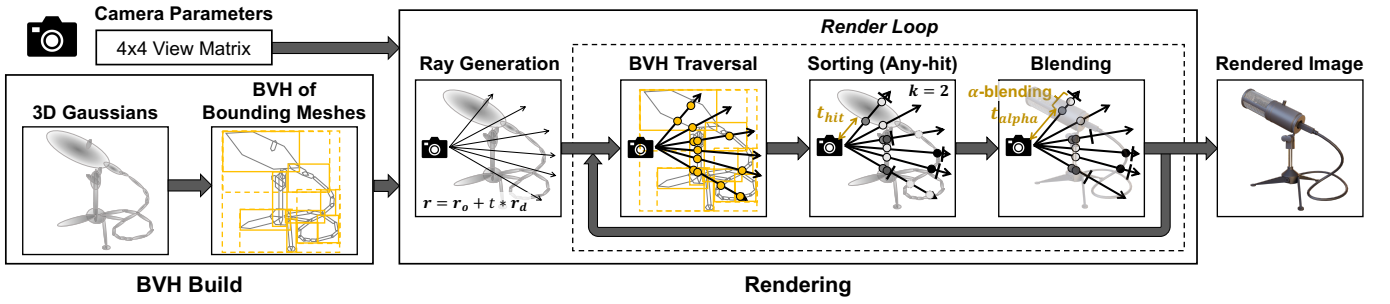
Fig. 3: Overview of 3D Gaussian Ray Tracing [35].

APIs such as Vulkan [22] and DirectX [33], or ray tracing frameworks such as OptiX [41]. Conceptually, the pipeline can be divided into three phases: ray generation, ray traversal, and post-traversal. Among the operations in the pipeline, shader programs (blue boxes) are executed on programmable shader cores in GPUs, while tree traversal and intersection tests (yellow boxes) are usually processed by fixed-function hardware, such as RT Cores in NVIDIA RTX GPUs.

To begin with, a ray generation shader calls a `traceRayEXT` function to initiate the ray tracing process. This establishes essential parameters for ray tracing, such as ray origin and direction, along with additional configuration flags. Once the function is called, the ray begins traversing the BVH to identify intersected primitives. The process continues until either a ray-primitive collision is detected or the traversal completes. Note that the traversal can proceed beyond the initial ray-primitive intersection point when the any-hit shader ignores it via `ignoreIntersectionEXT`, allowing evaluation of subsequent potential hits. For custom primitives, which are not natively supported by hardware, user-defined shaders can be used for ray-primitive intersection tests. After the traversal, either the closest-hit shader is invoked (when a hit is reported) or the miss shader is triggered (when no intersection occurs). The closest-hit shader can recursively invoke `traceRayEXT` to cast secondary rays from the intersection point, enabling effects like reflections, shadows, and global illumination.

B. 3D Gaussian-Based Rendering

3D Gaussian Splatting (3DGS) [20] introduces a novel approach for representing 3D scenes through anisotropic Gaussian primitives, achieving state-of-the-art visual quality and rendering performance. Each Gaussian is parametrized by spatial properties—its center position (mean) μ and 3×3 covariance matrix Σ—along with visual attributes including opacity o and spherical harmonic coefficients sh that encode view-dependent appearance. During training, these parameters are optimized to faithfully represent the scene geometry and appearance. At render time, each Gaussian is treated as an ellipsoid with defined boundaries around its distribution for computational efficiency. There are two methods for rendering Gaussian primitives.

Method 1: Rasterization. 3D Gaussian Splatting (3DGS) employs a *rasterization*-based rendering method [20]. That is, 3D Gaussians are *projected* onto the image plane as 2D *splats*, which are sorted by the depth value. The final color $\mathbf{C}$ of pixel position p is obtained by accumulating the colors of overlapping splats through α-blending as follows:

$$\mathbf{C} = \sum_{i=1}^{N} \alpha_i \mathbf{c}_i \prod_{j=1}^{i-1} (1 - \alpha_j),$$

$$\text{where} \quad \alpha_i = o_i \times \exp(-\frac{1}{2}(p - \mu_i')^T \Sigma_i'^{-1}(p - \mu_i')).$$

Here, $\mathbf{c}_i$ represents the color, and μ_i' and Σ_i' denote the 2D center and covariance matrix of the i-th splat, respectively.

Method 2: Ray Tracing. 3D Gaussian Ray Tracing (3DGRT) broadens the applicability of 3D Gaussian-based rendering by addressing fundamental limitations of rasterization. Many recent works [10], [15], [32], [35], [57] show its potential by enabling a variety of light effects such as shadow and reflection, extracting physical properties of a scene, and supporting complex camera models. In addition, while the original rasterization-based rendering (3DGS) performs global depth sorting shared across all pixels, ray tracing enables per-ray sorting that eliminates visual artifacts during camera movement. However, the advantages of 3DGRT over 3DGS come at a computational cost. The objective of this work is to mitigate the overhead of Gaussian ray tracing methods while preserving their benefits.

III. MOTIVATION

In this section, we analyze the Gaussian ray tracer implemented in 3DGRT [35] and compare its rendering performance against 3DGS. We then identify key inefficiencies in current ray tracing approaches that motivate the optimizations presented in this work.

A. 3D Gaussian Ray Tracing

Figure 3 presents an overview of 3D Gaussian ray tracing [35]. The process begins by constructing a BVH for the scene containing the Gaussians. Using this BVH structure along with camera parameters, rendering proceeds through four key steps: 1) ray generation based on camera parameters,

2) BVH traversal to identify intersecting Gaussians, 3) depth-based sorting of the intersecting Gaussians, and 4) alpha blending to compute the final pixel colors. In the following, we delve into the three most critical steps in 3DGRT.

BVH Traversal and Sorting. Volume rendering requires accumulating colors from all intersecting Gaussians in depth order. However, BVH traversal *does not* guarantee that Gaussians are discovered in this sorted sequence. This necessitates N traversal rounds over the scene geometry to collect the next N Gaussians along the ray with a closest-hit shader, which is computationally expensive.

To address this, 3DGRT employs a k-buffer approach [5], in which the next k *closest hit* Gaussians are gathered with a *single* traversal round using an *any-hit* shader. In ray tracing hardware, the distance (t_{hit}) from the camera to each intersection point is computed during the ray-primitive intersection test. The any-hit shader then uses the distance value as depth to maintain and update a k-entry buffer, which stores the Gaussians found thus far and keeps them in depth-sorted order. Note that the BVH traversal continues through the entire scene until all k closest Gaussians are *definitively* identified; i.e., it visits all the primitives that will intersect the ray.

The any-hit shader employs distance-based culling to isolate the k closest Gaussians. Initially, the traversal interval (t_{min}, t_{max}) is set to $(0, \infty)$ and is progressively updated throughout each tracing round. This instructs the RT core to only traverse BVH nodes that intersect the ray within the range from t_{min} to t_{max}. The k-buffer gradually fills with intersected Gaussians while sorting the buffer. When the buffer reaches capacity, it performs insertion sort to maintain only the k closest Gaussians, evicting the farthest one when a closer Gaussian is encountered.

Any subsequent Gaussian with a t_{hit} value exceeding the largest in the buffer triggers a hit report rather than being inserted into the buffer. This report updates t_{max} to the current Gaussian's t_{hit} value, instructing the RT core to restrict traversal only to the Gaussians and bounding volumes with smaller t_{hit} values. Traversal terminates when no candidates remain within the restricted t_{max} threshold. The final k-buffer contains the sorted indices and t_{hit} values of exactly the k closest Gaussians, optimally prepared for blending.

Alpha Blending. After the traversal, the colors of the Gaussians in the k-buffer are α-blended in order, as shown in Equation 1. Instead of pre-computing Gaussian colors as in 3DGS, however, 3DGRT obtains view-dependent colors per ray using SH coefficients and ray direction at runtime. As such, the alpha of the Gaussian is computed using the equation below:

$$\alpha = o \times G(\mathbf{r_o} + t_{alpha}\mathbf{r_d}), \text{ where } t_{alpha} = \frac{(\mu - \mathbf{r_o})^T \Sigma^{-1} \mathbf{r_d}}{\mathbf{r_d}^T \Sigma^{-1} \mathbf{r_d}}.$$

Here, G denotes a Gaussian function, with $\mathbf{r_o}$ and $\mathbf{r_d}$ representing the ray origin and direction, respectively. The parameter t_{alpha} is the evaluation point for alpha computation, positioned where the Gaussian achieves maximum response along the ray trajectory.
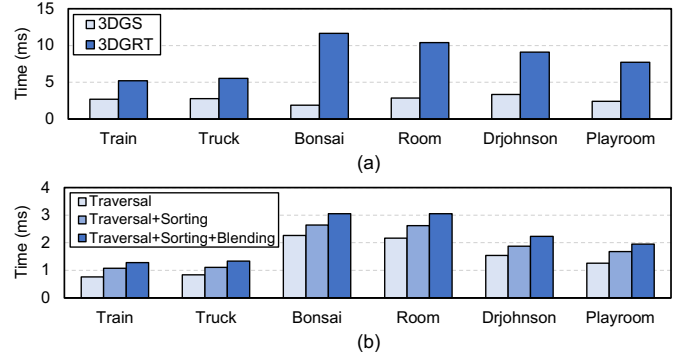

Fig. 4: (a) Rendering performance of rasterization (3DGS) and ray tracing (3DGRT). (b) Execution time for a single round of tracing while isolating each operation in 3DGRT.

After the blending operation, rays can be early terminated to reduce computational overhead when the accumulated alpha exceeds a predefined threshold. For rays that continue, tracing resumes from the t_{hit} value of the last blended Gaussian (i.e., $t_{min} = max(t_{hit})$) and proceeds until either all rays terminate or the traversal is complete.

B. Performance Analysis

We present a performance comparison between Gaussian ray tracing [35] and the original rasterization-based rendering (i.e., 3D Gaussian Splatting [20]). Using the official implementations of both 3DGS and 3DGRT, we train Gaussian models for 30K iterations and evaluate on several real-world scenes using an RTX 5090 GPU. As shown in Figure 4(a), ray tracing-based Gaussian rendering is on average approximately $3.04\times$ slower than rasterization; note that 3DGS could achieve even higher performance with additional optimizations. These results show that a large performance gap remains even with the aid of RT cores, indicating the need for further optimization.

To identify the performance bottleneck in Gaussian ray tracing, we incrementally add operations to the ray tracing pipeline and measure the execution time for a *single* tracing round comprising three operations: BVH traversal, per-ray sorting in the any-hit shader, and alpha blending in the raygen shader. While these operations execute concurrently within a single ray tracing API call—making precise isolation challenging—we can identify bottlenecks by observing significant increases in execution time when introducing each operation. As shown in Figure 4(b), BVH traversal dominates execution time, while sorting and blending contribute only marginally.

While 3DGS can directly identify which pixels intersect with Gaussians after 2D projection, 3DGRT requires exhaustive pointer chasing from root to leaf nodes for each ray to find intersecting primitives. Consequently, the performance gap becomes wider in scenes like Bonsai, where numerous small Gaussians are concentrated in specific regions, as this increases traversal time for rays passing through these dense areas. Conversely, when Gaussians are distributed more uniformly across the scene, the performance gap narrows, as observed in scenes like Train and Truck.
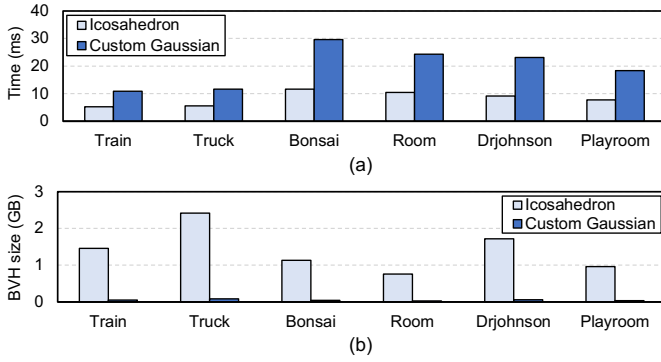
Fig. 5: (a) Rendering time and (b) BVH size when using triangles or custom primitives.

Fig. 6: (a) Performance comparison of single-round and multi-round traversal methods when $k = 16$. (b) Rendering time with different k values.

Fig. 7: Number of unique versus total visited nodes when $k = 16$. The data is extracted from Vulkan-Sim.

C. Observations and Opportunities

Observation I: Acceleration Structures and Bounding Primitives. In Gaussian ray tracing, selecting effective bounding primitives for anisotropic Gaussians is crucial when building an acceleration structure (BVH). Prior work primarily considers two types of geometric primitives: bounding triangle meshes [10], [35] or custom Gaussian (ellipsoid) primitives [6], [32]. While rendering quality remains the same regardless of bounding primitives, each offers distinct advantages and limitations, which lead to the differences in rendering performance.

Figure 5(a) compares rendering performance between two approaches for representing Gaussians in the BVH: a stretched regular icosahedron (20-faced polyhedron mesh) versus a custom ellipsoid primitive. Ideally, we want to insert just one primitive per Gaussian into the BVH to minimize node count and reduce traversal overhead. While custom primitives enable this one-to-one representation, the ray-primitive intersection tests need to be performed in software via user-defined shaders, which are substantially slower than hardware-accelerated ray-triangle intersection tests.

Using triangle meshes allows us to exploit the ray-triangle intersection hardware available in modern GPUs, resulting in faster rendering compared to using custom primitives. However, reasonably approximating a single Gaussian geometry requires a large number of triangle primitives, which increases BVH sizes and potentially more node visits, as shown in Figure 5(b). Furthermore, assigning separate bounding primitives to each Gaussian hurts the cache hit rates, as a scene typically contains hundreds of thousands or millions of Gaussians.

Ultimately, we still need a more effective solution to reduce the BVH size and better utilize the on-chip cache while still leveraging the intersection test hardware in GPUs. Section IV-A introduces our BVH construction strategy for Gaussian ray tracing before delving into hardware optimization techniques.

Observation II: Redundant BVH Traversal. Early ray termination (ERT), a widely used optimization technique in volume rendering, stops traversal when accumulated alpha exceeds a threshold, effectively reducing BVH traversal costs.
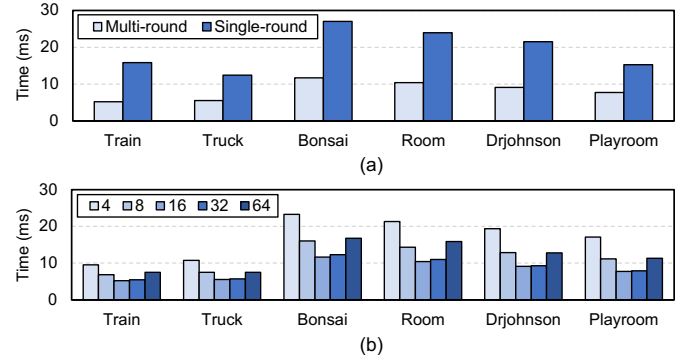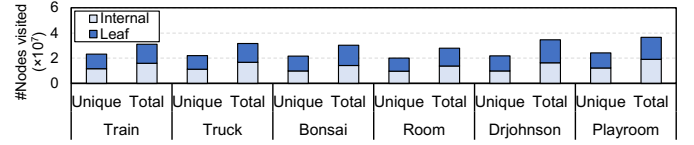
Figure 6(a) shows that multi-round traversal, which collects k Gaussians per round and enables ERT between rounds, outperforms single-round traversal that collects all intersected Gaussians before blending. For single-round traversal, we use a large k value (512–2048, depending on the scene) sufficient to store all intersected Gaussians. We collect all Gaussians in the any-hit shader without sorting, then perform sorting and blending after traversal. The results indicate that multi-round traversal reduces unnecessary traversal and sorting overhead for Gaussians that will not be blended due to ERT. Although prior work [32], [35] adopts this multi-round approach for performance, it suffers from redundancy; the RT core restarts from the root node each round despite tracing the same ray, leading to repeated node visits and intersection tests.

The choice of k presents a fundamental trade-off between the benefit of ERT and the redundancy between multiple BVH traversals. With a smaller k, ERT can be applied in a more fine-grained manner, thereby reducing the number of unnecessary node accesses and intersection tests during traversal. However, this requires more redundant traversals of internal and leaf nodes that have already been visited. Conversely, a larger k reduces the number of redundant traversals but increases unnecessary intersection tests for Gaussians that ultimately do not contribute to the final pixel color due to early ray termination. The extreme case of large k is single-round traversal, which eliminates redundancy but performs poorly due to excessive traversal and sorting beyond the ERT point.

Figure 7 quantifies this redundancy by showing unique and total node accesses and intersection tests across multiple rounds when using $k = 16$, which achieves the best performance, as shown in Figure 6(b). We observe that there is a
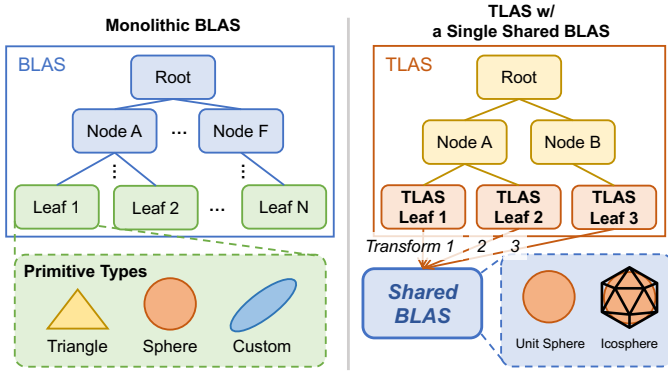
Fig. 8: Difference between a monolithic BVH AS and a TLAS with a shared BLAS BVH structure.

non-negligible gap between unique and total accesses, which implies that numerous BVH nodes are revisited and tested across rounds. Given that traversal constitutes the primary bottleneck in Gaussian ray tracing and these operations are memory latency-bound, eliminating this redundancy can effectively improve rendering performance. In Section IV-B, we introduce a checkpointing mechanism that enables subsequent rounds to resume from where previous rounds left off, rather than restarting from the root node.

IV. GRTX: GAUSSIAN RAY TRACING ACCELERATION

In this section, we present GRTX, software and hardware optimization techniques for Gaussian ray tracing. Our software optimization aims to accelerate BVH traversal by reducing the BVH size and its memory footprint while still exploiting hardware-accelerated ray-primitive intersection. Our hardware optimization introduces a checkpointing and replay mechanism to eliminate redundant BVH traversal and intersection testing across multiple rounds for each ray.

A. GRTX-SW: Leveraging Two-Level Acceleration Structure for Gaussian Primitives

Existing Gaussian ray tracing methods construct a single monolithic BVH for the entire scene while treating each Gaussian ellipsoid as a *separate* primitive. This results in excessively large BVH structures, particularly when encapsulating Gaussians with bounding meshes (e.g., 20 or 80 triangles per ellipsoid) to utilize ray-triangle intersection hardware. For instance, the Truck scene with 2.43M Gaussians requires a BVH size of approximately 2.42 GB when using 20-triangle bounding meshes.

Instead, we propose leveraging a two-level acceleration structure with a single *shared* base BLAS across all Gaussian primitives in a scene. Our key insight is that Gaussian ellipsoids can be treated as *unit spheres* once rays are transformed into their local coordinate systems, thereby eliminating the need to build individual BLAS for each Gaussian when utilizing two-level acceleration. This greatly reduces the BVH size—down to 432 MB for Truck—and the memory footprint during BVH traversal while also increasing the cache hit rate.

Figure 8 compares a monolithic BVH (i.e., all primitives in a single BVH) approach used in prior work [10], [35] and our proposed method of building BVHs for Gaussian ray tracing. In two-level acceleration, when a ray hits a leaf node in the TLAS, it is transformed to the local coordinate system by the transform matrix of each Gaussian, which is derived from its rotation and scaling matrices. Modern ray tracing hardware provides native support for this instance transform [1], [42]. Then, either a ray-sphere intersection test is performed when using a unit sphere as a primitive, or additional BLAS traversal occurs when using triangles as primitives.

Bounding Primitives for Gaussians. To exploit ray-primitive intersection hardware, we consider two alternative methods: 1) using a unit sphere, or 2) using an icosphere with multiple triangles. First, we can directly use a unit sphere as a primitive, which is optimal in terms of minimizing false positive intersection tests. After ray transformation, the Gaussian ellipsoid is equivalent to a unit sphere, so the sphere primitive exactly matches the Gaussian geometry. This avoids false positive intersections, which are the cases where a ray intersects the bounding primitive but not the actual Gaussian. In recent GPU architectures like NVIDIA Blackwell, RT cores natively support ray-sphere intersection tests that can be performed in hardware. This requires only one ray-AABB and one ray-sphere intersection test to determine whether a ray hits a Gaussian after transformation at the TLAS leaf node.

Second, instead of using a unit sphere, we can use an icosphere with multiple triangles. This method is similar to previous approaches that use a stretched icosahedron mesh [35] and a high-poly icosphere [10] to approximate Gaussian geometry. The key advantage of this method compared to the first one is that it can exploit high-throughput ray-triangle intersection test hardware in general RT units. While this approach may incur false positives in the intersection test, these can be mitigated by using a larger number of triangles (e.g., 80 triangles), as discussed by Condor et al. [10]. However, unlike their monolithic BVH, where the number of leaf nodes scales with the triangle count per mesh—resulting in multi-gigabyte BVH sizes—our shared BLAS keeps the overall BVH size small by storing only one template mesh of a few kilobytes.

We provide quantitative comparisons of these two approaches in Sections V-B and VI.

B. GRTX-HW: HW Acceleration for Gaussian Ray Tracing

Baseline Architecture and Operations. Figure 9 presents the baseline GPU architecture modeled in Vulkan-Sim [47] along with GRTX hardware extensions for ray tracing (RT) units. Each streaming multiprocessor (SM) contains a single RT unit comprising a dedicated scheduler, a warp buffer, and three types of fixed-function units: ray-box intersection units, ray-triangle intersection units, and ray transformation units.

Upon invoking the `traceRayEXT` intrinsic, a warp delegates its BVH traversal to the RT unit, which processes the ray information for each thread in the warp. The warp buffer stores and manages the associated per-ray data, including ray status
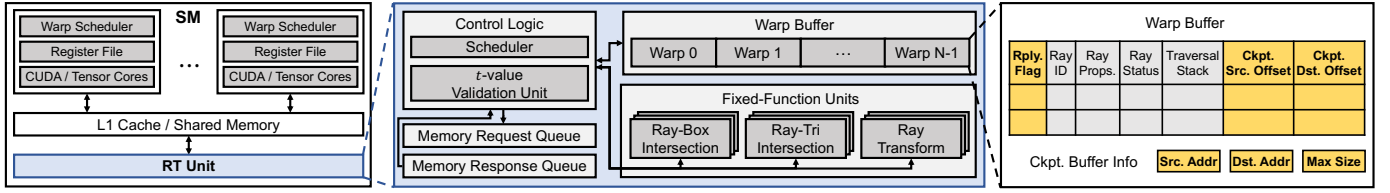
Fig. 9: Baseline GPU architecture modeled in Vulkan-Sim [47] with GRTX. Extended hardware is highlighted in yellow.
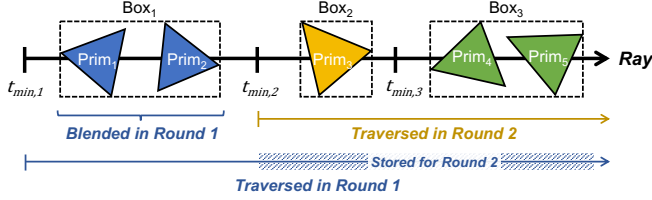

Fig. 10: High-level overview of checkpointing and replay mechanism in GRTX-HW.

(e.g., active or terminated), ray properties (e.g., ray origin, ray direction, t_{min}, t_{max}), and the traversal stack.

At each cycle, the RT scheduler selects a warp to process. The RT unit retrieves ray information from the warp buffer and fetches the node at the top of the traversal stack from memory. When the node data arrives, the RT unit performs either ray-box or ray-primitive intersection tests based on the node type.

A hit is reported when two conditions are satisfied: 1) the ray intersects with the node and 2) the hit point t_{hit} falls within the valid range ($t_{min} < t_{hit} \leq t_{max}$). Upon detecting a hit, the RT unit pushes the child node address onto the stack. For TLAS leaf nodes, the RT unit transforms the ray using the transform matrix stored in the node and pushes the address of the BLAS root node onto the stack. For the leaf nodes in the BLAS, hit information such as t_{hit} and primitive ID is also recorded in the warp buffer.

During traversal, when all active rays in a warp hit a primitive or a timeout occurs, the RT unit invokes the any-hit shader. When all threads in the warp finish the traversal, the warp retires from the RT unit and executes the rest of the shader program in the SM.

Traversal Checkpointing and Replay. Figure 10 presents the core concept of our checkpointing mechanism in GRTX-HW. During multi-round traversal, each round traces an identical ray but with different intervals; i.e., $t \in (t_{min,i}, \infty)$ for the i-th round. Because these intervals exhibit substantial overlaps, initiating each traversal from the root node results in redundant node accesses across rounds. Our key idea is to checkpoint the nodes and primitives that intersect within the overlapping intervals between consecutive rounds. These checkpointed nodes then serve as traversal starting points for the subsequent round, eliminating the need to retraverse from the root node. This approach substantially reduces the search space for each round by constraining traversal to the subtrees rooted at checkpointed nodes, which are traversed sequentially.

We checkpoint two distinct categories of elements (nodes and primitives). The first category includes BVH nodes that intersect the ray but are reported as *missed* because they lie beyond the k closest Gaussians, making further traversal unnecessary in the current round. These nodes are identified when they fail the t_{max} test ($t_{hit} > t_{max}$), and the RT unit stores the nodes in a checkpoint buffer. The second category comprises *primitives* that intersect the ray and are reported as *hit* ($t_{hit} \leq t_{max}$)—thus invoking the any-hit shader—but are ultimately rejected because they do not rank among the next k closest Gaussians for this round. These rejected primitives are stored in an eviction buffer by the any-hit shader.

```
1  rayGenShader(){
2    while (pixel.alpha < alphaThreshold) {
3      moveEvictToKBuf(evictBuffer, prd.evictOffset,
       kBuffer, k)
4      traceRayEXT()
5      blendGaussians(pixel, kBuffer)
6      if (prd.size < k) break
7    }
8  }
9
10 anyHitShader(){
11   rejected = insertionSort(kBuffer, tHit, primID)
12
13   if (prd.size == k) {
14     evictBuffer[prd.evictOffset] = rejected
15     prd.evictOffset++
16   }
17   prd.size = min(prd.size + 1, k)
18   if (tHit < rejected.tHit) {
19     ignoreIntersectionEXT()
20   }
21 }
```

Listing 1: Pseudo-code for the any-hit shader and raygen shader.

Listing 1 shows the pseudo-code for the any-hit and raygen shaders with the eviction buffer management. The any-hit shader first stores the rejected Gaussians from the k-buffer into the eviction buffer using the offset stored in the payload (Lines 13-16). Before the next traversal round starts, the raygen shader sorts and moves the first k Gaussians from the eviction buffer to the k-buffer (Line 3).

Checkpoint and Eviction Buffer. The checkpoint buffer is divided into two types: a source buffer and a destination buffer. In each tracing round, traversal starts from the root (first round; replay flag=0) or resumes from checkpointed nodes in the source buffer (subsequent rounds; replay flag=1). During traversal, newly encountered nodes requiring checkpointing are written to the destination buffer. To maintain proper buffer indexing, the source and destination offsets in the warp buffer
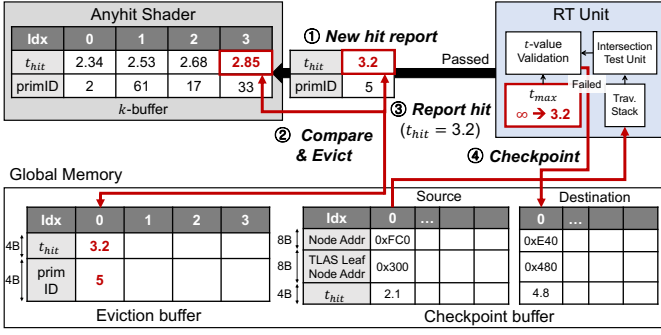
Fig. 11: Execution flow of checkpointing and replay.

TABLE I: Simulation configuration.

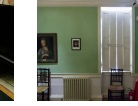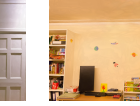| GPU	
# Streaming Multiprocessors (SM)	8, 1365 MHz, in-order
SIMT Lanes per SM	128 (4 warp schedulers)
L1I Cache	128 KB, 128B line, 16-way LRU, 20 cycles
L1D Cache	128 KB, 128B line, 256-way LRU, 20 cycles
L2 Cache (Unified)	4 MB, 128B line, 16-way LRU, 165 cycles
Memory Clock	3500 MHz
Ray Tracing Unit	
# RT Units per SM	1
Warp Buffer Size	8

V. EVALUATION

A. Methodology

Simulation Infrastructure. To evaluate the rendering performance of GRTX, we use Vulkan-Sim [47], a cycle-level graphics simulator that runs ray tracing applications, alongside an in-house cycle-level simulator that models the ray tracing behavior with any-hit shaders. The original Vulkan-Sim ray tracing implementation employs a delayed execution model that completes all ray traversal operations before executing intersection and any-hit shaders. However, this does not accurately reflect our baseline GPU behavior, where any-hit shaders are invoked during traversal, and subsequent traversal is influenced by any-hit shader results. For Gaussian ray tracing, traversal can also be early-terminated once the k closest Gaussians are found. Thus, we develop an in-house ray tracing simulator supporting immediate shading, which enables any-hit shaders to execute whenever rays in a warp detect intersected Gaussians, rather than waiting until all traversals complete. We integrate this RT simulator with Vulkan-Sim.

Table I shows the GPU configuration used in this work. We use 8 SMs and scale other parameters based on the RTX 5090 GPU, from which we collected our motivational data. We construct the BVH structure using Intel Embree [52], specifically employing a BVH-6 configuration that supports up to six children per node. We compare GRTX against the baseline RT execution that uses a stretched icosahedral mesh to approximate Gaussian geometry, as in 3DGRT [35].

We observe that baseline L1 cache hit rates on real GPUs are higher than those in our simulator, likely due to undisclosed optimizations. We assume that GPUs may employ optimizations that lead to an increase in cache hit rates during BVH traversal. We capture this effect by incorporating node prefetching into our simulator: upon the first demand fetch of any child leaf node, we issue a one-time prefetch for its sibling nodes whose bounding boxes are also intersected. This brings simulated L1 hit rates into closer alignment with those observed on real hardware.

Workloads. Table II presents the workloads used for our evaluation. We select six widely used scenes from diverse datasets [4], [18], [23], encompassing both indoor and outdoor real-world scenes with varying levels of complexity. For each scene, we train the Gaussian model for 30K iterations using the original *ray tracing-based* training implementation from 3DGRT [35], which results in approximately 0.8M to 2.4M

increment during each read and write to the corresponding checkpoint buffer. After each round, the destination buffer becomes the source buffer for the next round, creating a ping-pong buffer arrangement.

The checkpoint buffer and the eviction buffer require different entry formats due to their distinct purposes. Each checkpoint buffer entry contains the node address (8 bytes), the TLAS leaf node address (8 bytes) if it is a BLAS node, and the corresponding t_{hit} value (4 bytes), totaling 20 bytes per entry. The TLAS leaf node address is required for correct ray transformation: since we directly start traversal from a BLAS node (not from TLAS to BLAS), we need to transform the ray from world space to the object space of each Gaussian using the transformation matrix stored in the TLAS leaf node.

In contrast, eviction buffer entries have a simpler structure, containing only the primitive ID (4 bytes) and its t_{hit} value (4 bytes), since the entries in the eviction buffer will be directly moved to the k-buffer in the subsequent round, where they receive a second opportunity to be accepted as the k closest Gaussians. Note that both checkpoint and eviction buffers reside in global memory, not in the warp buffer, thus they do not require additional storage overhead.

Walkthrough Example. Figure 11 illustrates the complete execution flow of our checkpointing and replay mechanism through a concrete example. Before each round begins, we transfer evicted Gaussians from the eviction buffer to the k-buffer. During traversal, when the eviction buffer contains more than k Gaussians, we retain only the k closest to maintain the k-buffer size constraint.

Consider a scenario where the k-buffer (with $k = 4$) is full and encounters a new hit with $t_{hit} = 3.2$ and primitive ID 5. The any-hit shader first compares this new hit against the last entry (i.e., largest t_{hit}) of the k-buffer (①). Since the new hit is more distant than the current farthest Gaussian in the k-buffer (primID 33 with $t_{hit} = 2.85$), the new primitive (ID 5) is rejected and is stored in the eviction buffer (②). The shader then reports the hit to the RT unit with $t_{hit} = 3.2$ (③), which triggers an update of t_{max} from ∞ to 3.2. This updated t_{max} value influences subsequent traversal; the RT unit's t-value validation unit now rejects any intersections beyond this distance, and nodes failing this test are checkpointed to the destination buffer (④).

TABLE II: Summary of workloads. Images are rendered by our Vulkan implementation of Gaussian ray tracing. BVH sizes and memory footprints are measured using our simulator.

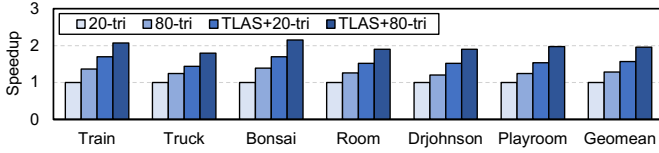| Dataset		Tanks&Temples [23]		Mip-NeRF 360 [4]		Deep Blending [18]	
Scene		**Train**	**Truck**	**Bonsai**	**Room**	**Drjohnson**	**Playroom**
Type		Real World (Indoor & Outdoor)					
Resolution		980×545	979×546	1559×1039	1557×1038	1332×876	1264×832
# of Gaussians		1.46 M	2.43 M	1.13 M	0.76 M	1.72 M	0.97 M
BVH Height (20-tri)		27	28	26	26	26	27
BVH Size	**20-tri**	2.34 GB	3.88 GB	1.81 GB	1.21 GB	2.75 GB	1.54 GB
	TLAS+20-tri	208 MB	345 MB	161 MB	107 MB	245 MB	137 MB
BVH Memory Footprint	**20-tri**	160 MB	181 MB	159 MB	150 MB	121 MB	77 MB
	TLAS+20-tri	33 MB	36 MB	30 MB	21 MB	15 MB	13 MB

Fig. 12: GRTX-SW performance with different Gaussian geometries.

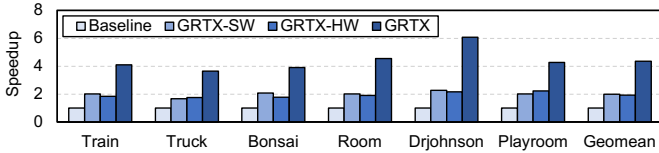Fig. 13: Speedup of GRTX over the baseline GPU using an icosahedron (i.e., 20-tri) as the bounding primitive.

Fig. 14: Number of node fetches normalized to baseline.

Gaussians per scene. To ensure tractable simulation time, we render all scenes at 128×128 pixel resolution while preserving the same field of view (FoV) as the original viewpoints.

To evaluate the end-to-end performance of GRTX, we implement a Gaussian ray tracing renderer in Vulkan. This is because the original 3DGRT implementation uses the NVIDIA OptiX framework, which is incompatible with our simulator that exclusively supports Vulkan-based ray tracing programs. We employ the same approach described in 3DGRT [35] to gather the next k closest Gaussians during a single traversal and perform early termination when possible. In Section VI, we discuss the details of our Vulkan implementation and show that it achieves performance similar to the original OptiX implementation.

B. Performance

Performance of GRTX-SW on Real GPU. Figure 12 shows the performance of GRTX-SW compared to the monolithic BVH with *20-faced* [35] and *80-faced* [10] stretched polyhedrons on an RTX 5090 while rendering images at 128×128 resolution. The results show that GRTX-SW provides noticeable speedups in both cases through the optimization of acceleration structures. Note that performance benefits may vary depending on rendering resolutions, FoVs, or the case of second ray tracing, which we discuss in Sections V-D and VI.
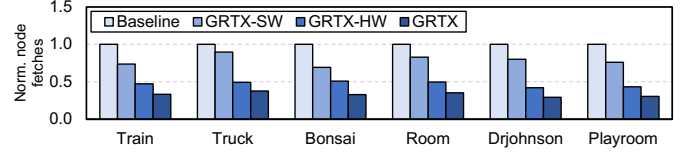
End-to-End Performance of GRTX. Figure 13 compares the end-to-end rendering performance of GRTX against the baseline 3DGRT (20-tri) implementation. GRTX-SW applies only the shared BLAS-based BVH construction (TLAS+20-tri) without hardware modifications, whereas GRTX-HW adds only traversal checkpointing to the baseline GPU. GRTX combines both optimizations. Overall, GRTX achieves an average speedup of 4.36× (up to 6.09×) over the baseline.

GRTX-HW avoids redundant node fetches and intersection tests across tracing rounds, which results in a 1.94× speedup on average. We observe that GRTX-HW delivers slightly higher speedups in scenes containing large Gaussians (e.g., the walls in Drjohnson and Playroom). In these cases, the large, overlapping bounding boxes of those Gaussians force rays to traverse deeper into the BVH—even for Gaussians that ultimately miss—thereby exacerbating redundant node visits across rounds. Our checkpointing mechanism mitigates this by resuming traversal at lower-level nodes, effectively bypassing redundant upper-hierarchy traversal.

We note that the benefits of GRTX-SW may be slightly higher in simulation, as our infrastructure may not fully capture the characteristics of state-of-the-art GPUs—though simulation results (2.00× average speedup) reasonably align with real GPU behavior. Nevertheless, with the results in Figure 12, we can conclude that GRTX substantially improves the rendering performance of Gaussian ray tracing by reducing the amount of traversal work and improving node fetch locality.

C. Source of Performance Gain

Reduction in Node Fetches. Figure 14 shows the number of BVH node fetches normalized to the baseline (20-tri). The results indicate that GRTX reduces the number of node fetches by 3.03× on average compared to the baseline.
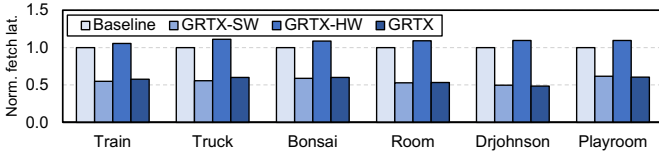
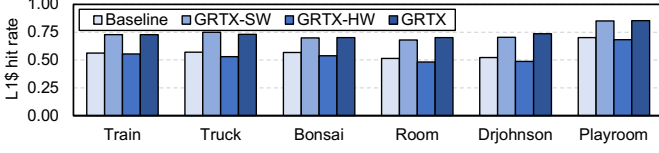Fig. 15: Average node fetch latency normalized to baseline.

Fig. 17: Total number of L2 cache accesses normalized to baseline.

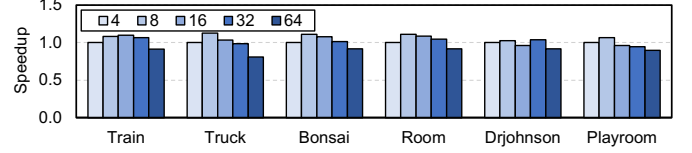Fig. 16: L1 cache hit rate for node fetches.

Fig. 18: Performance comparison across different k-buffer sizes.

GRTX-SW increases the likelihood that different rays fetch the same node by leveraging shared BLAS. These duplicate requests are merged into a single operation, thereby reducing overall node fetches. The magnitude of this reduction, however, varies depending on scene characteristics. In general, scenes with higher leaf-to-total node access ratios (e.g., Bonsai) tend to benefit more from GRTX-SW, since BLAS-level optimizations have a greater impact in such cases. Conversely, scenes with lower ratios (e.g., Truck) likely see reduced benefits, as overall traversal costs are more influenced by upper-level TLAS nodes.

On the other hand, the baseline RT unit lacks information from previous rounds, resulting in redundant node fetches and intersection tests. GRTX-HW addresses this by checkpointing the traversal state and reusing it in the next round, thereby eliminating the need to re-traverse nodes already visited in earlier rounds. As a result, GRTX-HW reduces node fetches by an additional $2.37\times$ on average on top of GRTX-SW. Since BVH traversal is the primary bottleneck in Gaussian ray tracing, avoiding these redundant operations leads to noticeable performance improvements.

Node Fetch Latency. Figure 15 shows the average node fetch latency across different configurations, all normalized to the baseline. The baseline uses a monolithic BVH with a 20-triangle bounding mesh for each Gaussian, resulting in a large BVH size. Consequently, many nodes are fetched from lower levels of the memory hierarchy, leading to high fetch latency. In contrast, GRTX employs a shared BLAS representing a unit sphere, which offers two key advantages: its compact size allows it to fit entirely within the L1 cache, and it enables high locality in node accesses during BLAS instance traversal.

With the checkpointing and replay mechanism, rays resume traversal from different nodes rather than uniformly starting from the root, which may reduce initial ray coherence. However, this does not result in a noticeable increase in average node fetch latency because traversal paths quickly diverge regardless of the starting point, making it difficult to exploit locality for node accesses even in the baseline. Overall, GRTX reduces the average node fetch latency by $1.77\times$ compared to the baseline.

L1 and L2 Cache Accesses. Figures 16 and 17 present the L1 cache hit rate and the number of L2 cache accesses for the baseline (20-tri) and our proposed approaches. The baseline exhibits relatively low L1 cache hit rates across all scenes due to the large memory footprint of its monolithic BVH structure. In contrast, GRTX-SW achieves substantial improvements, with L1 cache hit rates exceeding 70% across all evaluated scenes. This improvement stems from using a single shared BLAS for all Gaussian primitives, which enhances temporal locality during BVH traversal and allows the BLAS to reside within the L1 cache. The higher L1 hit rate directly translates to lower node fetch latency, as more BVH nodes are served from the fast L1 cache rather than from slower levels of the memory hierarchy.

GRTX maintains nearly the same L1 cache hit rates as GRTX-SW while further reducing L2 cache accesses. The reduction primarily stems from eliminating redundant node fetches through the checkpointing and replay mechanism. Overall, GRTX reduces L2 cache accesses by $4.75\times$ compared to the baseline. These results demonstrate that the shared BLAS design and checkpointing mechanism enable effective use of the cache hierarchy in Gaussian ray tracing, thereby improving rendering performance.

D. Sensitivity Study

k**-Buffer Size.** Figure 18 shows the performance of GRTX across different k-buffer sizes, which is normalized to $k = 4$. Since GRTX-HW eliminates redundant node fetches and intersection tests in subsequent traversal rounds, smaller k values can reduce the total BVH traversal cost by enabling more fine-grained ERT. However, using smaller k values leads to more traversal rounds (i.e., additional `traceRayEXT` calls), which in turn increases the overall intra-warp synchronization overhead, as threads that complete traversal early must wait for stragglers within the same warp to finish each round. The results indicate that performance generally improves as k decreases, but the increased straggler overhead eventually offsets the traversal savings; e.g., $k = 4$ performs worse than $k = 8$. In our evaluation, we use $k = 8$ as the default configuration since it delivers the best average performance.
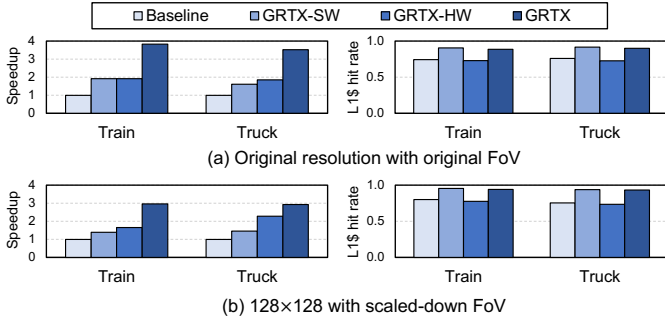
Fig. 19: Performance and L1 cache hit rate of GRTX across different resolution and FoV settings compared to the baseline.
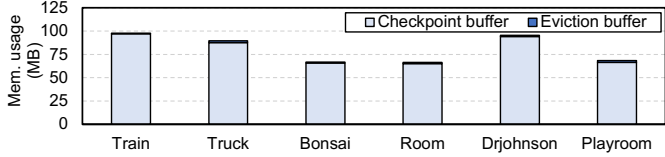

Fig. 20: Memory usage of GRTX for checkpoint and eviction buffers.

Varying Resolutions and FoVs. Figure 19 presents the performance and L1 cache hit rate of GRTX across different resolutions and FoVs. Figure 19(a) evaluates the original resolutions (listed in Table II) with the original FoVs, while Figure 19(b) uses a 128×128 resolution with proportionally scaled-down FoVs (equivalent to cropping). Higher resolutions and smaller FoVs both reduce the angular area per pixel, thereby increasing ray coherence. GRTX-HW provides consistent speedups across all scenarios, as it reduces redundant per-ray BVH traversal, independent of ray coherence. On the other hand, GRTX-SW exhibits lower relative speedups in high-coherence scenarios, as coherent rays already achieve high baseline cache locality. Nevertheless, it still provides average speedups of 1.75× and 1.43× for high-resolution and small FoV scenarios, respectively, by reducing the memory footprint.

E. Implementation Overhead

Table III shows the additional storage required for checkpointing in the warp buffer hardware. Our hardware extensions require only 1.05 KB of storage per RT core. Note that the checkpoint and eviction buffers used in GRTX-HW are allocated in global memory, as discussed in Section IV-B, with their sizes bounded by the maximum number of warps per SM multiplied by the number of SMs. Figure 20 shows the memory usage of these buffers for our baseline configuration (8 SMs). For the Train scene, which exhibits the highest memory consumption, these buffers consume only 97.68 MB

TABLE III: Hardware cost.

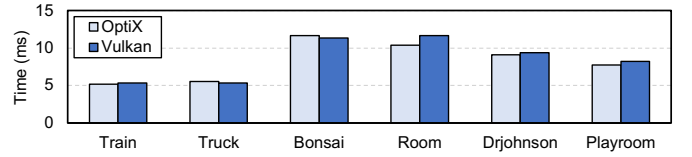| Hardware	Size
Checkpoint buffer information	(1-bit flag + 2B src offset + 2B dst offset) × 32 threads/warp × 8 warps + 8B src address + 8B dst address + 2B max size
Total	**1.05 KB**

Fig. 21: Rendering performance of the original OptiX implementation of 3DGRT [35] and our Vulkan implementation.
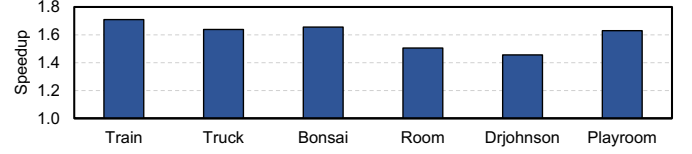

Fig. 22: Speedup of GRTX-SW with sphere primitive over the baseline icosahedron mesh measured on RTX 5090.

combined. Even when scaling to larger GPU configurations such as RTX 5090 (170 SMs), this increases proportionally to 2.03 GB—just 6.3% of the total 32 GB of GPU memory.

VI. ANALYSIS AND DISCUSSION

Vulkan Implementation of 3DGRT. As mentioned in Section V-A, we newly implement a Gaussian ray tracer in Vulkan [22] to run 3DGRT within our simulation framework, Vulkan-Sim. Following the original 3DGRT [35] implementation, we gather the next k closest Gaussians in the any-hit shader during a single traversal round and perform blending and early ray termination in the raygen shader after traversal. The original 3DGRT implementation uses payload values to store all entries of the k-buffer for each ray. Since OptiX limits the maximum number of payload values to 32 and each k-buffer entry requires two payload values, k is fixed to 16 in the original implementation. Vulkan allows more flexible use of ray payloads, but we observe that allocating the k-buffer within the payload structure results in a noticeable slowdown compared to OptiX. As such, we instead allocate the k-buffers of rays in global memory and employ a Structure of Arrays (SoA) layout to make the memory access coalesced.

Figure 21 compares the rendering performance of the original OptiX implementation and our Vulkan implementation. We observe that our Vulkan implementation achieves performance similar to OptiX.

Using Sphere Primitive in GRTX-SW. In the NVIDIA Blackwell architecture, the RT core natively supports ray-sphere intersection tests in hardware. By exploiting this hardware support, we can implement GRTX-SW using a single BLAS containing a unit sphere primitive, completely eliminating the need for triangle meshes. After transforming rays to Gaussian-local space, we only need one ray-box and one ray-sphere intersection test per Gaussian, which is more efficient than the triangle mesh-based approach.

Figure 22 shows the speedup of GRTX-SW with a sphere primitive over the baseline icosahedron mesh, measured on the RTX 5090. While the speedup is notable, we observe that the performance is lower than TLAS+80-tri, as shown in
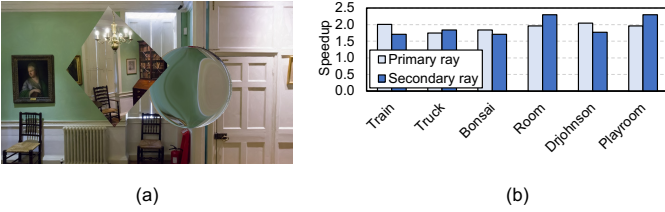
Fig. 23: GRTX-HW performance on scenes with secondary ray effects. (a) An example image rendered with light effects (refractions and reflections). (b) Speedups for primary and secondary rays.

Fig. 24: Normalized rendering time of baseline RT with monolithic BVH and GRTX-SW on an AMD GPU (Radeon RX 9070 XT). × indicates cases that cannot run as their BVHs exceed the maximum buffer allocation size (4 GB) in Vulkan.

Figure 12, potentially due to the throughput limitation of the ray-sphere intersection test in the current RT core. We expect that the performance will improve in future architectures with more advanced RT cores that provide higher throughput for ray-sphere intersection tests.

GRTX-HW on Secondary Rays. We evaluate the effectiveness of GRTX-HW on secondary rays. To assess this, we augment each scene by adding a spherical glass object for refractions and a rectangular mirror for reflections, both placed at random locations, as illustrated in Figure 23(a). We then measure performance separately for primary rays (i.e., those cast from the camera) and secondary rays (i.e., those generated by reflections and refractions) to isolate the performance impact on each ray type.

Figure 23(b) shows that GRTX-HW achieves similar speedups over the baseline for both primary and secondary rays. This is because our checkpointing mechanism reduces redundant traversal operations *within* individual rays rather than relying on ray coherence between different rays. Since replayed rays follow the exact same traversal paths as in previous rounds regardless of ray type, incoherent secondary rays also benefit from GRTX-HW.

Cross-Vendor Applicability. Modern RT accelerators vary in their implementations: some perform end-to-end traversal, including intersection tests and node fetches (e.g., NVIDIA, Intel), while others target only intersection operations with shader cores handling node fetches (e.g., AMD). Nevertheless, GRTX can provide performance benefits across GPU vendors as it addresses fundamental traversal inefficiencies—redundant traversal, divergence, and excessive memory footprint—that persist across all architectures.

Figure 24 shows the rendering time of the baseline and GRTX-SW, normalized to TLAS+80-tri, on the AMD Radeon RX 9070 XT. We observe that AMD generates larger BVHs than NVIDIA, which causes baseline RT with 20-/80-tri meshes to exceed the maximum buffer allocation size (4 GB) in Vulkan for most scenes. Our shared BLAS approach (TLAS+20-/80-tri) avoids this issue while achieving 1.73–3.42× speedup over the 20-tri baseline, demonstrating both memory efficiency and performance benefits across vendors.

Support for Dynamic and Multi-Object Scenes. One might wonder whether our two-level BVH approach conflicts with traditional dynamic scene rendering, which also uses two-level structures where each object is a BLAS instance in a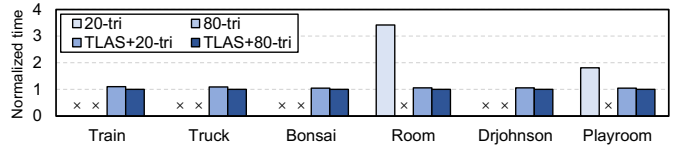 scene TLAS. However, GRTX naturally extends to dynamic and multi-object scenes through *multi-level instancing* (supported in OptiX/HIP RT), creating a three-level hierarchy: 1) a shared BLAS template for Gaussian primitives, 2) per-object instances, and 3) a scene-level TLAS. In this configuration, each Gaussian object maintains its own two-level structure (GRTX-SW), while multiple objects are organized under the scene TLAS for traditional dynamic scene management. Object additions or removals require updating the scene TLAS, and object movements require updating per-object transformation matrices—identical to conventional dynamic rendering with no additional GRTX-specific overhead.

VII. RELATED WORK

Radiance Field Rendering Acceleration. Radiance field-based rendering, exemplified by Neural Radiance Fields (NeRF) [34], has been actively studied for high-quality 3D scene reconstruction and rendering. However, NeRFs suffer from slow training and rendering, prompting numerous prior works to propose software optimizations [8], [14], [37], [51] and hardware accelerators [12], [24], [28], [29], [36], [40], [49], [50]. 3D Gaussian Splatting [20], the current state-of-the-art method, has attracted growing attention by achieving significantly faster rendering than NeRFs through rasterization while maintaining high image quality. Recent studies have also explored software and hardware optimizations to further accelerate 3D Gaussian Splatting [11], [17], [30], [43], [46], [53], [55], [56]. Among these, GSCore [26] and VR-Pipe [25] are the first to focus on hardware acceleration: GSCore proposes a dedicated accelerator, while VR-Pipe introduces a novel extension to the hardware graphics pipeline. However, both target rasterization-based Gaussian rendering. In contrast, GRTX accelerates 3D Gaussian ray tracing by extending ray tracing accelerators in modern GPUs. To our knowledge, GRTX is the first work to analyze performance bottlenecks of 3D Gaussian ray tracing and propose a hardware extension to existing GPU ray tracing accelerators.

3D Gaussian Ray Tracing. To address the limitations of rasterization-based Gaussian rendering, several studies [6], [10], [32], [57], including 3D Gaussian Ray Tracing [35] from NVIDIA, have demonstrated the potential of ray tracing for Gaussian rendering using ray tracing accelerators. While ray tracing hardware can effectively reduce rendering time, a significant performance gap remains between rasterization and ray tracing. GRTX bridges this gap through optimized BVH construction for Gaussian primitives and a minimal hardware extension to existing ray tracing accelerators in modern GPUs.

Ray Tracing Acceleration. Ray tracing architectures have been extensively explored in prior work [27], [38], [39], [48], [54]. GPU vendors now integrate dedicated ray tracing accelerators, such as NVIDIA's RT cores [7], into their GPUs. Building on these, several studies have proposed techniques to further improve ray tracing performance. Ray predictor [31] predicts ray intersections to skip traversal of upper-level nodes in the acceleration structure. While this effectively reduces traversal overhead when predictions are correct, it is limited to ambient occlusion, which only requires detecting a *single* intersection. However, 3D Gaussian ray tracing requires finding *all* intersecting Gaussians along the ray, making the ray predictor not directly applicable. Treelet prefetching [9] prefetches nodes at treelet granularity to reduce traversal latency in latency-bound ray tracing workloads. This technique is orthogonal to our work, which focuses on reducing memory footprint and the overall number of traversals.

Accelerating General-Purpose Workloads with RT Units. There have been attempts to accelerate general-purpose workloads using ray tracing units in GPUs [3], [13], [16], [58], [59]. RTNN [59] leverages RT units for nearest-neighbor search via software optimizations. TTA [16] and HSU [3] introduce hardware extensions that enable traversal of general hierarchical data structures (e.g., trees) beyond BVHs. Heliostat [13] and RTSpMSpM [58] extend RT units for page table walks and sparse matrix multiplication, respectively. In contrast, GRTX proposes software and hardware optimizations to accelerate Gaussian ray tracing, an emerging and increasingly important application within the conventional ray tracing domain.

VIII. CONCLUSION

Gaussian splatting has emerged as a leading technique for photorealistic image synthesis, attracting widespread attention across academia and industry. To overcome the inherent limitations of rasterization-based rendering, recent work has explored rendering Gaussians via ray tracing. However, existing methods suffer from low performance due to bloated acceleration structures and redundant node traversals. To address these inefficiencies, we introduce GRTX, which leverages two-level acceleration structures and provides hardware support for checkpointing and replay during BVH traversal. With these software and hardware optimizations, GRTX greatly improves the performance of Gaussian ray tracing over the existing method while incurring minimal hardware overhead.

ACKNOWLEDGMENT

REFERENCES

[1] Advanced Micro Devices, ""RDNA4" Instruction Set Architecture: Reference Guide," Tech. Rep., 2025.

[2] J. Barczak and H. Gruen, "Intel Arc Graphics Developer Guide for Real-time Ray Tracing in Games," 2023.

[3] A. Barnes, F. Shen, and T. G. Rogers, "Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[4] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[5] L. Bavoil, S. P. Callahan, A. Lefohn, J. a. L. D. Comba, and C. T. Silva, "Multi-Fragment Effects on the GPU using the k-buffer," in *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D)*, 2007.

[6] H. Blanc, J.-E. Deschaud, and A. Paljic, "RayGauss: Volumetric Gaussian-Based Ray Casting for Photorealistic Novel View Synthesis," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2025.

[7] J. Burgess, "RTX on—The NVIDIA Turing GPU," *IEEE Micro*, 2020.

[8] Z. Chen, T. Funkhouser, P. Hedman, and A. Tagliasacchi, "MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[9] Y. H. Chou, T. Nowicki, and T. M. Aamodt, "Treelet Prefetching For Ray Tracing," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[10] J. Condor, S. Speierer, L. Bode, A. Bozic, S. Green, P. Didyk, and A. Jarabo, "Don't Splat your Gaussians: Volumetric Ray-Traced Primitives for Modeling and Rendering Scattering and Emissive Media," *ACM Transactions on Graphics (TOG)*, 2025.

[11] Y. Feng, W. Lin, Y. Cheng, Z. Liu, J. Leng, M. Guo, C. Chen, S. Sun, and Y. Zhu, "Lumina: Real-Time Neural Rendering by Exploiting Computational Redundancy," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

[12] Y. Feng, Z. Liu, J. Leng, M. Guo, and Y. Zhu, "Cicero: Addressing Algorithmic and Architectural Bottlenecks in Neural Rendering by Radiance Warping and Memory Optimizations," in *Proceedings of the 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[13] Y. Feng, Y. Li, J. Lee, W. W. Ro, and H. Jeon, "Heliostat: Harnessing Ray Tracing Accelerators for Page Table Walks," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

[14] S. Fridovich-Keil, A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance Fields Without Neural Networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[15] J. Gao, C. Gu, Y. Lin, H. Zhu, X. Cao, L. Zhang, and Y. Yao, "Relightable 3D Gaussian: Real-time Point Cloud Relighting with BRDF Decomposition and Ray Tracing," *Proceedings of the European Conference on Computer Vision (ECCV)*, 2024.

[16] 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 *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[17] H. He, G. Li, F. Liu, L. Jiang, X. Liang, and Z. Song, "GSArch: Breaking Memory Barriers in 3D Gaussian Splatting Training via Architectural Support," in *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[18] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow, "Deep Blending for Free-Viewpoint Image-Based Rendering," *ACM Transactions on Graphics (SIGGRAPH Asia)*, 2018.

[19] Y. Jiang, C. Yu, T. Xie, X. Li, Y. Feng, H. Wang, M. Li, H. Lau, F. Gao, Y. Yang, and C. Jiang, "VR-GS: A Physical Dynamics-Aware Interactive Gaussian Splatting System in Virtual Reality," in *ACM SIGGRAPH 2024 Conference Papers*, 2024.

[20] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D Gaussian Splatting for Real-Time Radiance Field Rendering," *ACM Transactions on Graphics (SIGGRAPH)*, 2023.

[21] Khronos Group. Acceleration Structures. [Online]. Available: https://docs.vulkan.org/spec/latest/chapters/accelstructures.html

[22] Khronos Group. Vulkan. [Online]. Available: https://registry.khronos.org/vulkan

[23] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction," *ACM Transactions on Graphics (SIGGRAPH)*, 2017.

[24] J. Lee, K. Choi, J. Lee, S. Lee, J. Whangbo, and J. Sim, "NeuRex: A Case for Neural Rendering Acceleration," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[25] J. Lee, J. Kim, J. Park, and J. Sim, "VR-Pipe: Streamlining Hardware Graphics Pipeline for Volume Rendering," in *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[26] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[27] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "SGRT: A Mobile GPU Architecture for Real-Time Ray Tracing," in *Proceedings of the 5th High-Performance Graphics Conference (HPG)*, 2013.

[28] S. Li, C. Li, W. Zhu, B. T. Yu, Y. K. Zhao, C. Wan, H. You, H. Shi, and Y. C. Lin, "Instant-3D: Instant Neural Radiance Field Training Towards On-Device AR/VR 3D Reconstruction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[29] S. Li, Y. Zhao, C. Li, B. Guo, J. Zhang, W. Zhu, Z. Ye, C. Wan, and Y. C. Lin, "Fusion-3D: Integrated Acceleration for Instant 3D Reconstruction and Real-Time Rendering," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[30] W. Lin, Y. Feng, and Y. Zhu, "MetaSapiens: Real-Time Neural Rendering with Efficiency-Aware Pruning and Accelerated Foveated Rendering," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[31] L. Liu, W. Chang, F. Demoullin, Y. H. Chou, M. Saed, D. Pankratz, T. Nowicki, and T. M. Aamodt, "Intersection Prediction for Accelerated GPU Ray Tracing," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[32] A. Mai, P. Hedman, G. Kopanas, D. Verbin, D. Futschik, Q. Xu, F. Kuester, J. T. Barron, and Y. Zhang, "EVER: Exact Volumetric Ellipsoid Rendering for Real-time View Synthesis," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2025.

[33] Microsoft. Direct3D. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d

[34] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[35] N. Moenne-Loccoz, A. Mirzaei, O. Perel, R. de Lutio, J. M. Esturo, G. State, S. Fidler, N. Sharp, and Z. Gojcic, "3D Gaussian Ray Tracing: Fast Tracing of Particle Scenes," *ACM Transactions on Graphics (TOG)*, 2024.

[36] M. H. Mubarik, R. Kanungo, T. Zirr, and R. Kumar, "Hardware Acceleration of Neural Graphics," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[37] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding," *ACM Transactions on Graphics (SIGGRAPH)*, 2022.

[38] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, "RayCore: A Ray-Tracing Hardware Architecture for Mobile Devices," *ACM Transactions on Graphics (TOG)*, 2014.

[39] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T&I Engine: Traversal and Intersection Engine for Hardware Accelerated Ray Tracing," in *Proceedings of the 2011 SIGGRAPH Asia Conference (SA)*, 2011.

[40] S.-H. Noh, B. Shin, J. Choi, S. Lee, J. Kung, and Y. Kim, "FlexNeRFer: A Multi-Dataflow, Adaptive Sparsity-Aware Accelerator for On-Device NeRF Rendering," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

[41] NVIDIA. OptiX Ray Tracing Engine. [Online]. Available: https://developer.nvidia.com/rtx/ray-tracing/optix

[42] NVIDIA, "NVIDIA RTX Blackwell GPU Architecture," 2025. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf

[43] M. Pei, G. Li, J. Si, Z. Zhu, Z. Mo, P. Wang, Z. Song, X. Liang, and J. Cheng, "GCC: A 3DGS Inference Architecture with Gaussian-Wise and Cross-Stage Conditional Processing," in *Proceedings of the 58th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025.

[44] Z. Peng, T. Shao, Y. Liu, J. Zhou, Y. Yang, J. Wang, and K. Zhou, "RTG-SLAM: Real-time 3D Reconstruction at Scale using Gaussian Splatting," in *ACM SIGGRAPH 2024 Conference Papers*, 2024.

[45] PlayCanvas, "PlayCanvas WebGL Game Engine," 2024. [Online]. Available: https://github.com/playcanvas/engine

[46] L. Radl, M. Steiner, M. Parger, A. Weinrauch, B. Kerbl, and M. Steinberger, "StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering," *ACM Transactions on Graphics (SIGGRAPH)*, 2024.

[47] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, "Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[48] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR: A Hardware Architecture for Ray Tracing," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS)*, 2002.

[49] X. Song, Y. Wen, X. Hu, T. Liu, H. Zhou, H. Han, T. Zhi, Z. Du, W. Li, R. Zhang, C. Zhang, L. Gao, Q. Guo, and T. Chen, "Cambricon-R: A Fully Fused Accelerator for Real-Time Learning of Neural Scene Representation," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[50] Z. Song, H. He, F. Liu, Y. Hao, X. Song, L. Jiang, and X. Liang, "SRender: Boosting Neural Radiance Field Efficiency via Sensitivity-Aware Dynamic Precision Rendering," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[51] C. Sun, M. Sun, and H.-T. Chen, "Direct Voxel Grid Optimization: Super-Fast Convergence for Radiance Fields Reconstruction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[52] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Transactions on Graphics (TOG)*, 2014.

[53] H. Wang, Z. Zhu, T. Zhao, Y. Xiang, Z. Wang, J. Yu, H. Yang, Y. Xie, and Y. Wang, "REACT3D: Real-time Edge Accelerator for Incremental Training in 3D Gaussian Splatting based SLAM Systems," in *Proceedings of the 58th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2025.

[54] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing," *ACM Transactions on Graphics (TOG)*, 2005.

[55] L. Wu, H. Zhu, S. He, J. Zheng, C. Chen, and X. Zeng, "GauSPU: 3D Gaussian Splatting Processor for Real-Time SLAM Systems," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[56] Z. Ye, Y. Fu, J. Zhang, L. Li, Y. Zhang, S. Li, C. Wan, C. Wan, C. Li, S. Prathipati, and Y. C. Lin, "Gaussian Blending Unit: An Edge GPU Plug-in for Real-Time Gaussian-Based Rendering in AR/VR," in *2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2025.

[57] Z. Yu, T. Sattler, and A. Geiger, "Gaussian Opacity Fields: Efficient Adaptive Surface Reconstruction in Unbounded Scenes," *ACM Transactions on Graphics (TOG)*, 2024.

[58] H. Zhang, Y. Zhang, and H.-W. Tseng, "RTSpMSpM: Harnessing Ray Tracing for Efficient Sparse Matrix Computations," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA)*, 2025.

[59] Y. Zhu, "RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2022.