VR-Pipe: Streamlining Hardware Graphics Pipeline for Volume Rendering

Junseo Lee Jaisung Kim Junyong Park Jaewoong Sim

Seoul National University
{junseo.lee, jaisung.kim, junyong.park, jaewoong}@snu.ac.kr

***Abstract*—Graphics rendering that builds on machine learning and radiance fields is gaining significant attention due to its outstanding quality and speed in generating photorealistic images from novel viewpoints. However, prior work has primarily focused on evaluating its performance through software-based rendering on programmable shader cores, leaving its performance when exploiting fixed-function graphics units largely unexplored.**

In this paper, we investigate the performance implications of performing radiance field rendering on the hardware graphics pipeline. In doing so, we implement the state-of-the-art radiance field method, 3D Gaussian splatting, using graphics APIs and evaluate it across synthetic and real-world scenes on today's graphics hardware. Based on our analysis, we present VR-Pipe, which seamlessly integrates two innovations into graphics hardware to streamline the hardware pipeline for volume rendering, such as radiance field methods. First, we introduce native hardware support for early termination by repurposing existing special-purpose hardware in modern GPUs. Second, we propose multi-granular tile binning with quad merging, which opportunistically blends fragments in shader cores before passing them to fixed-function blending units. Our evaluation shows that VR-Pipe greatly improves rendering performance, achieving up to a 2.78× speedup over the conventional graphics pipeline with negligible hardware overhead.

I. Introduction

The advent of graphics techniques that combine machine learning and radiance fields has sparked significant interest in a new direction of representing 3D scenes via implicit neural fields (e.g., NeRF [32]) or explicit rendering primitives that are also differentiable (e.g., 3D Gaussian [18]). Compared to traditional methods that use meshes and textures, these radiance field-based techniques allow us to capture intricate details in 3D scenes and produce far more realistic images from new and unseen viewpoints, leading to their rapid adoption and plug-in development for popular graphics engines such as Unity [46] and Unreal [10]. In particular, the graphics community is actively exploring the Gaussian splatting [18] technique due to its ability to generate high-fidelity images with much faster rendering speeds than other radiance field methods.

The core of the impressive rendering performance of Gaussian splatting lies in the use of explicit *rasterization* primitives. While Gaussian primitives can, in principle, be rendered through the hardware graphics pipeline using graphics APIs such as OpenGL [41], Vulkan [19], and Direct3D [31], prior work has primarily focused on assessing their performance based on software-based rendering [18], [38] or building a specialized accelerator [24], thereby leaving the potential of exploiting graphics-specific hardware largely unexplored.

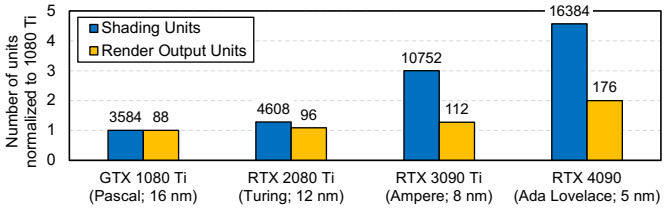

Fig. 1: Number of shader cores and render output units in the recent generations of top-of-the-line NVIDIA desktop GPUs. The labels on the bars indicate the absolute numbers for each.

In this paper, instead of developing a dedicated accelerator, we explore leveraging fixed-function hardware in GPUs to improve the rendering efficiency of Gaussian splatting on commodity GPUs. In doing so, we first implement Gaussian splatting rendering using an industry standard graphics API and evaluate it on modern desktop and edge GPUs. Our implementation shows that hardware-based radiance field rendering generally offers better or comparable performance compared to state-of-the-art software-based rendering, though this can vary slightly depending on the scenes and hardware configurations, such as the number of programmable shader cores or fixed-function raster operation (ROP) units in GPUs. However, we also observe that the existing hardware graphics pipeline falls short of efficiently performing Gaussian splatting rendering. This is because Gaussian splatting is a volume rendering technique that produces a pixel color by accumulating a huge amount of *transparent* primitives (i.e., Gaussians) into the color buffer through per-pixel blending.

In contrast, today's graphics hardware is primarily designed for mesh-based rendering with mostly *opaque* geometry. Although GPUs have seen a gradual increase in the number of ROP units over successive generations, the growth has been relatively modest, as shown in Figure 1. For conventional mesh-based rendering, which typically involves one or only a few fragments per pixel, ROP units rarely become the pipeline bottleneck. However, volume rendering puts substantially more pressure on ROPs due to the blending of hundreds of fragments per pixel. Moreover, a commonly used optimization technique for volume rendering, *early ray termination*, is not natively supported in graphics hardware, further limiting the potential for maximizing rendering performance.

Unfortunately, software-based optimizations aimed at reducing the ROP pressure or supporting early termination do not provide much performance benefit. For example, to alleviate the ROP pressure, one could use an extension feature available in certain graphics APIs (e.g., OpenGL's `ARB_fragment_shader_interlock`) that allows for atomic pixel blending within fragment shaders. However, this *in-shader* blending approach can lead to a significant drop in rendering performance due to the overhead of lock acquisition (§IV-A). Also, previous research has explored multi-pass rendering [21] to perform early termination using graphics APIs, but this approach offers limited speedups or can even degrade performance due to the overhead of invoking multiple intermediate rendering passes (§IV-B).

To this end, we present VR-Pipe (Volume Rendering Pipeline), which features two innovations that streamline the hardware graphics pipeline to better support volume rendering workloads such as radiance field rendering. The first is native hardware support for early termination by leveraging existing special-purpose units (i.e., stencil test hardware) in contemporary GPUs with minimal extension. Our key observation is that both stencil test and early termination share a similar purpose, so we can *repurpose* the stencil test hardware for checking early termination with negligible changes to ROPs. The second is multi-granular tile binning with quad merging, which reduces the number of blending operations performed in ROPs. A key insight is that we can perform *opportunistic* blending of multiple fragments within a warp before passing them to ROPs by leveraging the associative property of the blending equation and changing the computation order.

We implement VR-Pipe on the Emerald simulator [15], which builds on gem5 [6] and GPGPU-Sim [4], while making extensive modifications to the baseline implementation to better model contemporary NVIDIA-like GPUs based on our analysis on real graphics hardware. Our evaluation shows that VR-Pipe improves Gaussian rendering performance by 2.07× on average compared to the baseline graphics pipeline. In summary, we make the following contributions:

- To our knowledge, this is the *first* work to identify the challenges and inefficiencies associated with Gaussian-based radiance field rendering on the hardware graphics pipeline using graphics APIs.
- We present VR-Pipe, a hardware graphics pipeline that features native support for early termination and multi-granular tile binning with quad merging, which substantially improves the performance of volume rendering workloads such as radiance field rendering.
- We implement OpenGL-based microbenchmarks and provide an analysis on several key fixed-function units in modern graphics hardware.

II. Background

This section first provides the background on 3D graphics rendering. It then describes the state-of-the-art radiance field rendering method: 3D Gaussian splatting.

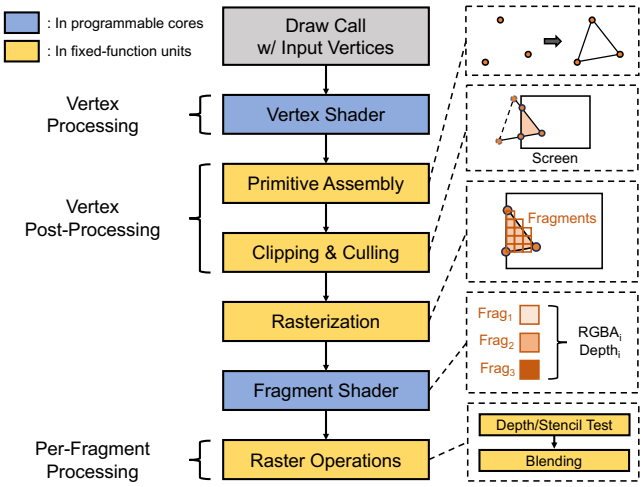

Fig. 2: OpenGL rendering pipeline.

A. Preliminaries on 3D Graphics Rendering

Graphics Pipeline. A 3D scene is rendered into a 2D image through a series of stages, which is referred to as a *graphics pipeline* or a *rendering pipeline*. To run the graphics pipeline, graphics software conventionally builds on standard graphics APIs such as OpenGL [41], Direct3D [31], Vulkan [19], and Metal [2]. Each API defines a set of functions that process the operations in the rendering pipeline on graphics hardware.

Figure 2 illustrates a high-level overview of the OpenGL rendering pipeline. Other graphics APIs also employ a similar pipeline model. The OpenGL pipeline can be largely divided into five stages: vertex shading, vertex post-processing, rasterization, fragment shading, and per-fragment processing. In hardware-based graphics rendering, each pipeline stage maps to either programmable shader cores or fixed-function units in GPUs. Note that in software-based rendering, the operations in each stage are executed entirely on the shader cores without using fixed-function hardware.

When a draw call is invoked with input vertices, the vertex shader[1] transforms the position of each vertex from 3D world space into clip space coordinates, which will be further transformed into 2D screen positions and depth by fixed-function hardware. In the vertex post-processing stage, the vertices are then assembled into primitives (e.g., triangles). In this stage, primitives outside the visible space are excluded through a process known as *view frustum culling*, and only the visible part of a primitive remains if part of the primitive is outside the screen space.

The visible primitives are fed into a hardware rasterizer to identify the pixels that overlap with them. The rasterizer produces *fragments* for each primitive; if a pixel is covered by multiple primitives, there will be more than one fragment for the pixel. Also, vertex attributes computed by the vertex shader are interpolated for each fragment in this stage.

Using the per-fragment data (e.g., pixel position, interpolated features) and shared data (e.g., textures), the fragment

[1]A shader is a small program that runs on the shader cores.

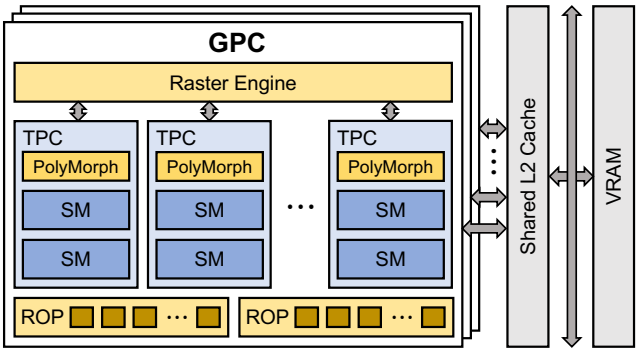

Fig. 3: NVIDIA Ampere GPU architecture [36].

shader computes and outputs a color and an opacity (i.e., an RGBA value) for each fragment. In the final per-fragment processing stage, raster operations perform depth and stencil tests. For the fragments that pass the tests, their RGBA colors are blended or stored into the color buffer to generate the final pixel color. It should be noted that the rendering pipeline can be implemented in hardware with various optimizations, as long as the final pixel colors are correctly produced.

Graphics-Specific Hardware in GPUs. As previously discussed, modern GPUs employ programmable shader cores that execute different types of shader programs. Today, the shader cores are not only accessible from graphics software but are also exposed to run general-purpose programs through software frameworks such as CUDA and OpenCL. Still, GPUs also feature graphics-specific hardware that facilitates the execution of certain parts of the graphics pipeline, which is *not* accessible via general-purpose computing frameworks.

As shown in Figure 3, for example, an NVIDIA GPU includes several special-purpose graphics units in addition to the programmable shaders (i.e., Streaming Multiprocessor; SM). Each Graphics Processing Cluster (GPC) includes a number of Texture Processing Clusters (TPCs), each of which contains a PolyMorph Engine. The PolyMorph Engine performs operations such as vertex fetching and viewport transformation, and forwards the results to the Raster Engine [47].

The Raster Engine (rasterizer) sets up triangle edges using input vertex positions and computes the pixel coverage of each triangle, a process called rasterization. The fragments produced by the rasterizer are sent to the depth (z) test unit (ZROP) if an early z-test is enabled. This unit compares the depth of each fragment with the value in the z-buffer at the same pixel position, discarding fragments that would ultimately fail the late z-test conducted after fragment shading. By doing so, it prevents unnecessary fragment shading computations. After fragment shading in the shader cores (SM), the render output units (ROPs), also known as raster operation units, perform blending or storing operations while ensuring the proper ordering of fragments for the same pixel location.

Tile-Based Rendering. Most contemporary GPUs, including NVIDIA RTX, AMD Radeon, Intel Gen, and ARM Mali, now use some variant of tile-based rendering (TBR). When rendering an image using the hardware graphics pipeline, the screen space is divided into a grid of screen tiles, each containing a block of pixels. These tiles are assigned to the shader cores in the form of warps or thread blocks. For instance, NVIDIA GPUs split the screen space into a grid of 16×16-pixel tiles, each of which is assigned to a specific GPC. This improves cache locality and reduces off-chip memory access during rendering. To achieve this, GPUs perform tile binning in hardware [17], [26]. The fragments produced by the hardware rasterizer are grouped into bins based on their tile IDs. These bins are then flushed to the shader cores when certain conditions are met (e.g., a bin is full, a timeout occurs, or there is a lack of available bins for new fragments with different tile IDs). Section VII provides further analysis and discussion of fixed-function units and tile-based rendering, based on microbenchmarking of modern GPUs.

B. Radiance Field Rendering with Gaussian Splatting

3D Gaussian splatting [18] introduces a novel method that achieves state-of-the-art rendering performance and quality by *explicitly* representing a scene with a set of anisotropic 3D Gaussians. Each Gaussian is characterized by geometric properties, such as a position (mean) coordinate μ and a 3×3 covariance matrix Σ, as well as visual properties, such as opacity o and spherical harmonic (SH) coefficients sh, to represent the view-dependent color of the Gaussian.

For training, given a sparse set of 2D images, an initial set of 3D points is generated using a Structure-from-Motion (SfM) technique. These points serve as the centers for the initial isotropic Gaussians. During the training phase, the features of the Gaussians are updated continuously based on their computed gradients. To better represent the fine geometric details of the scene, the number of Gaussians increases as they are cloned and split into smaller ones.

While a 3D Gaussian is mathematically defined as a continuous function over the entire 3D space, Gaussian splatting models each Gaussian as an ellipsoid for practical purposes. During rendering, these 3D Gaussians are projected onto the 2D image plane as ellipses, referred to as *2D splats*. The splats are sorted by depth, from nearest to farthest relative to the given viewpoint. The final pixel color ($\mathbf{C}$) is then computed using α-blending (Equation 1), which combines the colors ($\mathbf{c}_\mathrm{i}$) of overlapping splats in front-to-back order:

$$\mathbf{C} = \sum_{\mathrm{i}=1}^{\mathrm{N}} \alpha_\mathrm{i} \mathbf{c}_\mathrm{i} \prod_{\mathrm{j}=1}^{\mathrm{i}-1} (1-\alpha_\mathrm{j}), \quad \text{with} \quad \alpha_\mathrm{i} = o_\mathrm{i} \cdot \exp(-\frac{1}{2}(p'-\mu')^T \Sigma'^{-1}(p'-\mu')), \tag{1}$$

where p' denotes the pixel position, and μ' and Σ' represent the mean and the covariance matrix of the 2D splat, respectively.

III. Motivation

A. 3D Gaussian Splatting on Graphics Hardware

To understand the performance implications of exploiting the hardware pipeline for Gaussian splatting, we implement its rendering process using OpenGL and evaluate it across synthetic and real-world scenes on mobile and desktop GPUs.

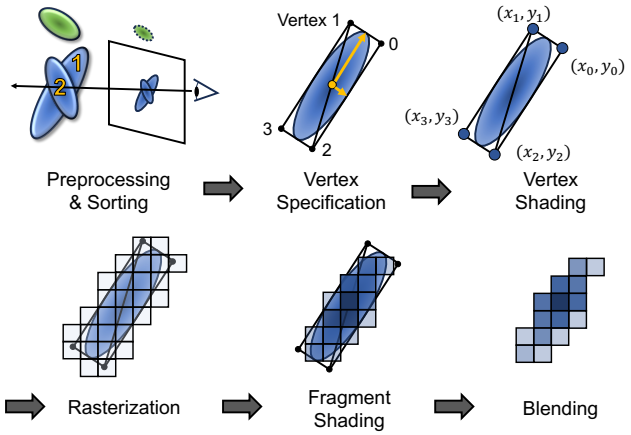

Fig. 4: Gaussian splatting rendering via graphics APIs.

Rendering 3D Gaussians via Graphics APIs. Figure 4 illustrates how 3D Gaussians are rendered using graphics APIs. To begin with, we implement custom CUDA kernels and use the NVIDIA CUB library for the preprocessing and sorting steps, similar to the software-based rendering [18]. For the subsequent steps, we use OpenGL to implement vertex and fragment shaders that execute on graphics hardware.

In the preprocessing and sorting steps, we first perform frustum culling to exclude invisible Gaussians and calculate a depth value, which is the z-value of the center of each Gaussian in camera space. We then project (i.e., "splat") the Gaussians onto screen space and compute an RGB color of each splat using SH coefficients and the viewing direction. The splats are subsequently sorted by the depth values to perform alpha blending front-to-back in a draw call.

Once we obtain a sorting order of the splats, we exploit fixed-function units in graphics hardware for rendering. This requires each splat to be represented as conventional graphics primitives (e.g., triangles). To achieve this, we specify an Oriented Bounding Box (OBB) [14] that surrounds each splat using two triangles with four vertices. By using the center of the splat, two semi-axis vectors of the ellipse, and the vertex indices, the vertex shader computes a 2D coordinate for each vertex in screen space. Each vertex is also assigned the color and opacity values of the corresponding Gaussian, which were previously computed during the preprocessing step.

Subsequently, the hardware rasterizer produces fragments that overlap with the triangles. Afterward, the fragment shader computes the alpha value of each fragment by evaluating a Gaussian function at the pixel position (Equation 1). The fragments whose alpha values are small enough (i.e., $\alpha < \frac{1}{255}$) are excluded from blending, which we call *alpha pruning* in this paper, and the remaining fragments are blended into the corresponding pixels in the ROP units.

Performance Analysis. Figure 5 compares the performance of the state-of-the-art software-based rendering using custom CUDA kernels [18] and our OpenGL-based rendering that exploits fixed-function graphics units. To reduce the amount of

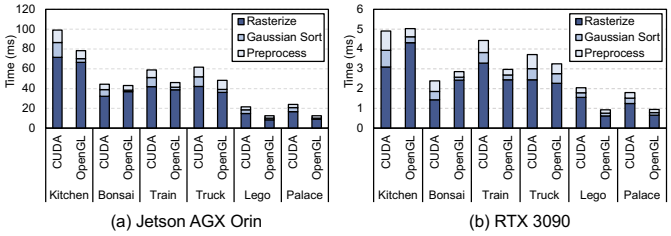

Fig. 5: Performance comparison between software-based (CUDA) and hardware-based (OpenGL) graphics rendering on Jetson AGX Orin and RTX 3090. See Table II for the details of each scene.

work in rasterization, we use a tight OBB[2] that accounts for the opacity of each Gaussian in both CUDA and OpenGL implementations. Specifically, for CUDA, this approach significantly reduces the number of ineffective Gaussian-tile assignments, resulting in a notable speedup in sorting and rasterization compared to the axis-aligned bounding box (AABB)-based method used in the original CUDA implementation, as discussed in previous studies [24], [38]. Note that this optimization does not alter the rendered image, as it only reduces ineffective computations.

Results show that using fixed-function units generally offers better performance than software-based rendering. In software-based rendering, preprocessing is inefficient, requiring per-tile buffers and duplicating depth and index data for Gaussians spanning multiple tiles, which is time-consuming. In contrast, hardware-based rendering eliminates the inefficiencies, as the graphics hardware *automatically manages* tiling and duplication. This also leads to a reduction in sorting time, as we only need to sort the entire Gaussians by depth values without the need of duplication and per-tile sorting. Furthermore, as noted in prior work [24], the lockstep execution of threads in CUDA leads to *ineffective* alpha computation during rasterization. Hardware-based rendering, operating at a finer granularity (e.g., a 2×2-fragment quad) than a warp (i.e., 32 threads), allows for more effective utilization of shader cores and ROPs, thereby improving performance during the rasterization step.

While rendering using graphics hardware in GPUs is generally faster than software-based rendering, it still falls short of real-time rendering, particularly on mobile/edge devices with limited power and resources. As shown in Figure 5(a), hardware-based rendering achieves less than 25 FPS for real-world scenes. In the following section, we further discuss our observations and the inefficiencies of Gaussian-based rendering on the hardware pipeline.

B. Observations and Opportunities

Observation I: ROP Pressure and Inefficient Use of Shader Cores. Figure 6 shows the average throughput utilization of several key graphics hardware units during a draw call. We observe that the rendering performance of Gaussian splatting is dictated by the ROP units (i.e., PROP and CROP) rather than by the shader units (SM). This is because there are a huge number of fragments to blend per pixel in Gaussian

[2]The Gaussian's boundary is defined where the alpha value is equal to $\frac{1}{255}$.

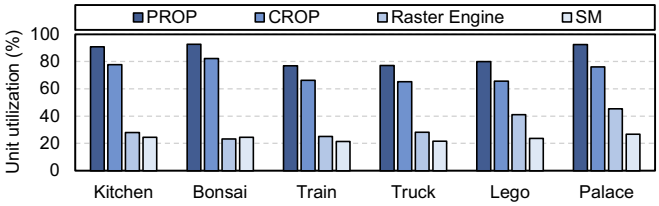

Fig. 6: Throughput utilization of each hardware unit for OpenGL-based rendering, which is computed by $(\frac{\text{Measured Throughput}}{\text{Max Throughput}})\times 100$.

(a) w/o early termination (b) w/ early termination

Fig. 7: Number of fragments per pixel with and without early termination (scene: Bonsai).

splatting. The PROP (Pre-ROP) orchestrates the flow of depth and color fragments for a final pixel, while the CROP (Color ROP) performs blending in the order of fragments received from the PROP.

In addition, the shader units are relatively underutilized because of two reasons. First, the ROP units are the pipeline bottleneck, so the shader units are often not fully utilized due to back pressure. Second, the vertex and fragment shaders used for Gaussian splatting are relatively simple compared to CUDA-based renderers. In detail, the vertex shader only computes the 2D screen coordinate using the center of the splat and axis vectors, and the vertex colors are shared for all vertices of the splat, which are already computed in the preprocessing step. The fragment shader computes the alpha value of each fragment by applying a dot product to a normalized pixel coordinate and performing an exponential operation of the Gaussian function. Compared to other shading programs that require lighting calculations and texture fetching for each fragment [21], the shaders for Gaussian splatting are computationally cheaper.

Observation II: Challenges of Supporting Early Termination. Gaussian splatting is a volume rendering technique that accumulates RGBA colors of fragments at the same pixel location *front-to-back* to produce the final pixel color. Consequently, similar to other volume rendering methods, it can benefit from *early termination*, a widely-used optimization technique in the volume rendering process. Figure 7 illustrates the impact of early termination on Gaussian splatting by comparing the number of fragments per pixel blended with *and* without early termination. With early termination, if the alpha value of a pixel surpasses a predefined threshold after blending, the subsequent fragments are discarded as they do not make a noticeable contribution to the final pixel color. As a result, this technique effectively reduces the amount of computation by avoiding unnecessary shading and blending operations.

Figure 8 shows the speedup of CUDA-based rendering and

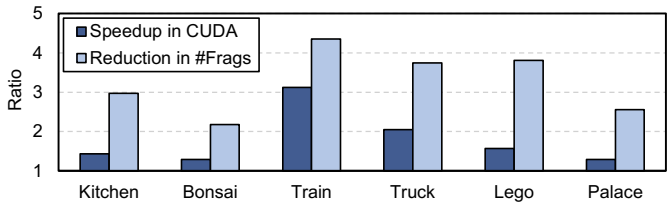

Fig. 8: Speedup of CUDA-based rendering and the reduction in the number of fragments with early termination.

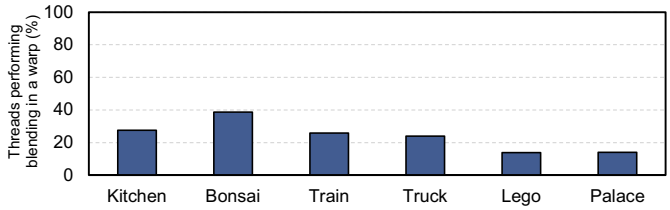

Fig. 9: Average percentage of threads in a warp that participate in blending in software-based (CUDA) rendering.

the reduction in per-pixel fragment count when using early termination. In software-based rendering, while early termination helps improve rendering performance, its full benefits are difficult to realize due to the lockstep execution of threads working on different pixels. In the worst case, even if only one thread (pixel) in a warp is not terminated, all other threads in the warp still ineffectively consume shader cores until the active thread finishes. Figure 9 shows the percentage of threads in a warp performing blending operations for the evaluated scenes. With the combined effects of alpha pruning and early termination, less than 40% of threads perform effective work (i.e., blending) in a warp across all scenes.

On the other hand, hardware-based rendering currently does *not* natively support early termination. However, adding this capability to graphics hardware would significantly improve rendering performance for two reasons while better exploiting its benefits compared to software-based rendering. First, the hardware graphics pipeline performs fragment blending at a finer granularity in ROPs (i.e., a quad; 2×2 fragments) than a warp, thereby enabling more effective utilization of compute units when early termination is applied than software-based rendering. Second, since volume rendering places substantial pressure on ROPs, reducing the number of fragments sent to them would greatly streamline the hardware pipeline by alleviating their processing burden. To harness this benefit in the hardware pipeline, Section V-B discusses our proposed architecture, which minimally extends the graphics hardware by *repurposing* fixed-function units, such as stencil test hardware.

In the following section, we discuss the performance of in-shader blending and software-based early termination via graphics APIs to demonstrate the need for architectural support for volume rendering.

IV. Software Optimizations and Limitations

A. Performing In-Shader Blending

One approach to reduce the ROP pressure might be performing raster operations, particularly pixel blending, in the

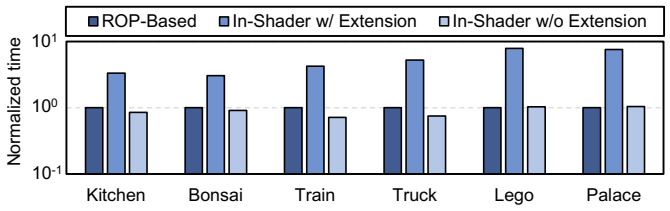

Fig. 10: Normalized rasterization time of the ROP-based blending and in-shader blending with an OpenGL extension (log scale).

fragment shader. While one may think that this could take advantage of the high parallelism of shader cores, we cannot naïvely perform blending in the fragment shader because we need to ensure the blending order of fragments to obtain accurate pixel colors. Even though graphics hardware forms thread blocks and dispatches them to the shader cores in order, the fragment threads corresponding to the same pixel can run *out-of-order* due to warp scheduling policy and memory systems (e.g., cache hits/misses, memory controller). As a result, performing in-shader blending without additional mechanisms does not prevent fragment threads from violating the blending order and atomicity, thereby producing an incorrect pixel color.

Recently, several standard graphics APIs offer an extension that enables the use of a critical section in the fragment shader, such as `GL_ARB_fragment_shader_interlock` in OpenGL and `VK_EXT_fragment_shader_interlock` in Vulkan, where an order of lock acquisition can be configured to follow the order of fragments. We implement in-shader blending by modifying the fragment shader with the OpenGL extension. After discarding the fragments with alpha values below the threshold (i.e., ϵ=$\frac{1}{255}$) through alpha pruning, alive threads (fragments) attempt to acquire a lock by calling `beginInvocationInterlockARB`.[3] Inside the critical section, the pixel (RGBA) load, blend, and store operations are performed in an atomic manner. By releasing the lock after the store, the next fragment thread in line enters the critical section and reads the valid pixel color.

We observe that in-shader blending for Gaussian splatting performs substantially worse than ROP-based blending, as shown in Figure 10. The significant drop in performance is mainly due to the overhead of the locking mechanism rather than the raster operations. As we can see, in-shader blending *without* interlocking, which allows fragment threads to run in parallel to access and update the same pixel in non-deterministic order, performs close to or faster than ROP-based rendering. However, it does not guarantee the exact blending results. In summary, we conclude that it is necessary to use ROPs for blending in terms of performance and preciseness in volume rendering.

B. Software-Based Early Termination

To demonstrate the effectiveness of hardware-based early termination, we first implement software-based (OpenGL) early termination via conventional multi-pass rendering approaches [21]. The key concept of the multi-pass approach for early termination is to render an image with *multiple* draw calls. If a pixel is terminated on the i-th draw call due to an early termination condition (i.e., $\alpha \geq 0.996$), we skip processing the fragments of the terminated pixel from the $(i+1)$-th draw call using a stencil test.[4] As the original algorithm [21] was devised for texture-based volume rendering, we begin by describing our algorithm for Gaussian splatting.

Algorithm 1 Multi-Pass Rendering with Early Termination

Input: N; Number of passes, G; Gaussians, View; Viewpoint
Output: Pixels; RGBA pixel colors

```
Pixels ← 0; Stencils ← 0
Splats, Depths ← Preprocess(G, View)
SortedIdx ← SortGaussians(Depths)
Batches ← SplitIntoNBatches(Splats, SortedIdx)
for each i from 1 to N do
    // Draw call with the i-th batch of splats
    Pixels ← DrawBatch(Batches[i], Pixels, Stencils)
    if i < N then
        // Stencil buffer update for terminated pixels
        Stencils ← EarlyTerminate(Pixels.A, Stencils)
    end if
end for
```

Algorithm 1 outlines the procedure for multi-pass rendering with early termination. Initially, the colors and stencil values of pixels are set to 0. A stencil value of zero indicates that the corresponding pixel has not yet been terminated. For N-pass rendering, the algorithm first divides the depth-ordered splats equally into N batches after preprocessing and sorting. It then iterates through multiple passes, with each pass consisting of two draw calls. In the first call, we *partially* update the colors of the pixels that pass the stencil test (i.e., stencil value == 0). In the second call, we update the stencil values for the pixels that meet the termination condition in the first call.

To update the stencil value, the second call renders a screen-sized rectangle composed of two triangles using a different fragment shader and stencil test. The rasterizer produces a fragment for every pixel, which is sent to the fragment shader. Each fragment in the shader loads the alpha of the pixel and is discarded if the pixel is not terminated yet (i.e., $\alpha <$ 0.996). After the shading, the stencil test is performed for the non-discarded fragments, which represent the terminated pixel locations up to this pass, and updates the stencil values to 1. Consequently, the fragments of early-terminated pixels are not shaded or blended in subsequent passes, reducing the ROP pressure.

Figure 11 shows the speedup of software-based early termination according to the number of passes. The baseline is the original OpenGL-based rendering, where the number of passes (N) is 1. The performance is influenced by two factors: the reduction in the number of fragments due to early termination and the overhead from additional draw

[3]We configure the extension so that entering the critical section follows the order of the fragments.

[4]The stencil test is one of the raster operations, which controls fragment processing by comparing the per-pixel stencil value in a stencil buffer with the predefined reference value.

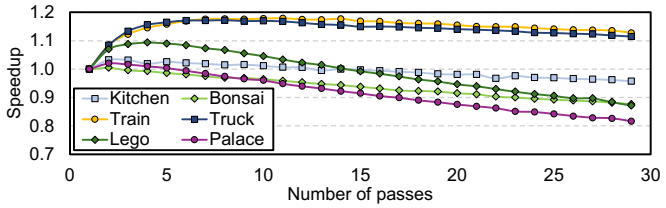

Fig. 11: Performance of software-based early termination (OpenGL) with multi-pass rendering across the number of passes.

calls for stencil updates. While using a larger N can reduce fragments more by checking early termination in a finer-grained manner, it also increases the overall overhead of stencil updates. As a result, for the scenes with low fragment reduction (e.g., Bonsai, Kitchen) or already short rendering times (e.g., Lego, Palace), the performance improvement is marginal, or it performs worse than the baseline due to the overhead. In large scenes with more fragment reduction (e.g., Train, Truck), we can observe a speedup, but it does not reach its full potential due to the stencil update overhead, which hardware-based early termination helps avoid. In addition, the optimal number of passes varies depending on the scene and viewpoint, making the software-based approach less practical. Support for hardware-based early termination can also help address this issue.

V. VR-Pipe: Streamlining Graphics Hardware

A. Overview of Architecture and Execution Flow

Figure 12 provides an overview of the baseline GPU architecture along with our hardware extensions for volume rendering. Because graphics hardware is designed to support the standard graphics APIs, the high-level microarchitecture is broadly similar across different hardware vendors, though each may implement minor optimizations. Accordingly, we build our extensions on a general graphics architecture commonly found in contemporary GPUs and described in prior literature [15], [26], [37], [40], [44], combined with our analysis of real graphics hardware. Note that the hardware extensions are generally applicable as they do not significantly alter the standard hardware pipeline.

When input vertices are assigned to a SIMT cluster, the shader cores fetch vertex data from memory, perform vertex shading, and output the resulting vertices with attributes. Instead of passing the actual attribute data, a pointer to the attributes is sent to the next pipeline stage after the attributes are stored in a designated memory region in the L2 cache.[5]

The vertices are then assembled as triangle primitives in the vertex processing and operations (VPO) unit. For each primitive, the VPO unit also identifies intersecting *screen* tiles (e.g., a block of 16×16 pixels) by computing the bounding box of each primitive. Since each SIMT cluster is exclusively responsible for a set of screen tiles, the primitives are then distributed across SIMT clusters through a crossbar to run the subsequent pipeline stages. If a primitive intersects multiple

[5]In NVIDIA terminology, this memory region is called the Circular Buffer (CB), and the corresponding cache lines are marked as no-evict [40].

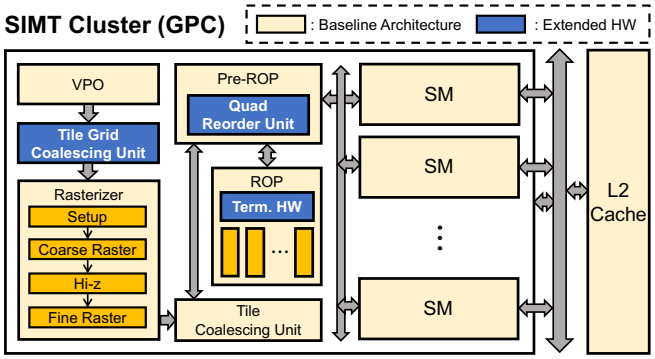

Fig. 12: VR-Pipe microarchitecture.

tiles, it is sent to multiple different clusters and processed independently.

The tile grid coalescing (TGC) unit then manages these primitives at a coarser granularity than a screen tile, which we call a *tile grid* (e.g., 4×4 screen tiles). Each bin in the TGC unit manages a tile grid and collects primitives that intersect with any of the cluster's screen tiles within that grid. When a bin has collected a sufficient number of primitives for the tile grid, the TGC unit flushes the bin.

Once a TGC bin is flushed, the rasterizer performs rasterization for every primitive in four sequential steps: setup, coarse raster, hierarchical-z (Hi-z) test, and fine raster. The setup unit first computes edge equations of the primitive using vertex coordinates. The coarse raster then identifies intersecting *raster* tiles (e.g., 8×8 pixels) within a screen tile. The Hi-z test is performed on each raster tile, and only the raster tiles that pass the test are sent to the next pipeline stage. Finally, the fine raster unit checks if each pixel in the raster tile is covered by the primitive. The pixels within the primitive are then assembled as 2×2-fragment quads.

The tile coalescing (TC) unit manages a set of TC bins, each collecting the quads for the *same* screen tile. It aggregates the quads from the fine raster unit into the corresponding bin and flushes them to the Pre-ROP (PROP) hardware, which controls the blending order between multiple quads. The flush occurs when one of three conditions is met. First, the bin is full. Second, all bins are occupied and a quad from a new tile arrives (the oldest bin is flushed). Third, pre-determined cycles have elapsed after the last incoming quad.

When the bin is flushed, the quads are sent to the depth-stencil ROP (ZROP) for an early termination check before fragment shading. This hardware support for early termination operates at the *fragment* level during a *single* draw call, thereby realizing most of its potential compared to CUDA-based or OpenGL-based implementations. The quads with at least one fragment that passes the early termination test are sent back to the PROP. A quad reorder unit in the PROP then identifies the overlapping quads that can be *merged* in the shader cores and sets the necessary flags. These quads are then dispatched to the shader cores as warps for fragment shading, where quad merging is performed via warp shuffling.

After the shading, the surviving fragments from alpha pruning and quad merging are sent to the color ROP (CROP)

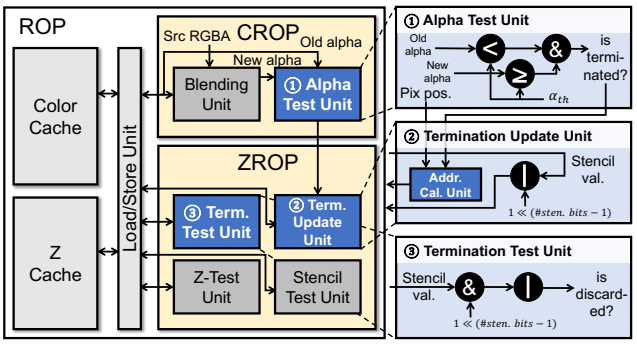

Fig. 13: Early termination hardware in ROP.

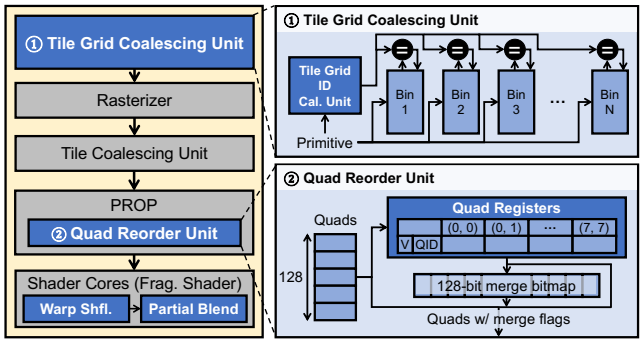

Fig. 14: Overall hardware pipeline for quad merging.

for pixel blending. The termination hardware in the ROP checks if the termination condition is met after blending and updates termination information as needed. In the following subsections, we explain the architectural details of hardware-based early termination and multi-granular tile binning with quad merging.

B. Hardware Support for Early Termination

Figure 13 shows hardware extensions to enable early termination in the graphics pipeline. Early termination and stencil test share a similar purpose: eliminating fragments that do not affect the final rendered output before shading and blending. In this sense, the early termination unit can be architected in the same stage where the stencil test unit is placed.

Our key insight is that multi-bit (e.g., 8-bit) stencil data per pixel is already used for the stencil test, and we only need one bit data for checking if the pixel is early terminated. In most cases, only a few bits of the stencil value are used for the stencil test by masking (e.g., `glStencilMask(0x01)`), and the remaining bits can be used for the early termination test. By repurposing the most significant bit (MSB) of the stencil value for the termination check, both early termination and the stencil test can be supported harmonically.

For example, when an 8-bit stencil value is used, the MSB serves as a termination flag, while the remaining 7 bits are used for the stencil test. Initially, the MSB is set to 0, allowing fragments of the pixel to pass the termination test and blend with the pixel color. Once the pixel's alpha value is sufficiently accumulated, the MSB is set to 1, marking the pixel as terminated. All subsequent fragments for the pixel are then discarded before shading. As such, the early termination unit can be implemented with negligible overhead by leveraging the existing stencil buffer.

We add three lightweight computational units to the ROP. First, after blending the pixel color (destination RGBA) with the shaded fragment color (source RGBA) in CROP, the alpha test unit determines if the early termination condition is met by checking if the alpha exceeds a predefined threshold by this fragment. Instead of comparing only the accumulated alpha value, we also check if the previous alpha does not exceed the threshold. This is because using only the former condition would unnecessarily generate more termination bit update requests to ZROP, which leads to bandwidth contention to the z-cache and an increase in latency for the termination test. If the pixel is newly terminated, the alpha test unit then sends a termination signal to ZROP with a pixel coordinate.

In ZROP, the termination update unit is triggered by the termination signal and updates a termination bit to 1. After the address calculation with the pixel coordinate, it first loads the stencil value and sets the termination bit using a bitwise OR operation. The updated stencil value of the terminated pixel is then written back to the z-cache.

The early termination test is performed for the fragments that are flushed from the TC unit. By conducting the test, we can effectively discard fragments of the pixels that are already early-terminated and no longer need to be blended. Consequently, this early termination test reduces the amount of work for shader cores and ROPs, easing the burden of fragment shading and blending, respectively.

C. Reducing ROP Workload by Quad Merging

It is costly and challenging to increase ROP throughput by adding more blending units to the SIMT cluster and increasing ROP cache bandwidth. Instead, we leverage shader cores for blending, which offer high compute throughput yet are underutilized in Gaussian rendering. Our key insight is that volume rendering has an *associative* property, so we can alter the order of fragment blending—though not arbitrarily.

$$\mathbf{C} = \sum_{\mathrm{i}=1}^{3} \alpha_{\mathrm{i}} \mathbf{c}_{\mathrm{i}} \prod_{\mathrm{j}=1}^{\mathrm{i}-1} (1-\alpha_{\mathrm{j}}) = f_{fb}(f_{fb}(\mathbf{c}_{\mathrm{pm},1}, \mathbf{c}_{\mathrm{pm},2}), \mathbf{c}_{\mathrm{pm},3}), \qquad (2)$$

where $\mathbf{c}_{\mathrm{pm,i}} = (\alpha_{\mathrm{i}}\mathrm{r}_{\mathrm{i}}, \alpha_{\mathrm{i}}\mathrm{g}_{\mathrm{i}}, \alpha_{\mathrm{i}}\mathrm{b}_{\mathrm{i}}, \alpha_{\mathrm{i}})$, $f_{fb}(\mathbf{c}_1, \mathbf{c}_2) = \mathbf{c}_1 + (1-\alpha_1)\mathbf{c}_2$.

Equation 2 shows an example of blending two fragments $\mathbf{c}_2$ and $\mathbf{c}_3$ into a (pre-multiplied alpha) pixel RGBA color $\mathbf{c}_{\mathrm{pm},1}$. In the hardware pipeline, fragments are sent from PROP to CROP in front-to-back order. Since the front-to-back alpha blending equation f_{fb} is associative, i.e., $f_{fb}(f_{fb}(\mathbf{c}_1, \mathbf{c}_2), \mathbf{c}_3) = f_{fb}(\mathbf{c}_1, f_{fb}(\mathbf{c}_2, \mathbf{c}_3))$, we can *partially* change the computation order of volume rendering. This allows for *opportunistically* blending fragments in shader cores, thereby reducing the number of blending operations by ROPs without affecting the final pixel color; note that we cannot obtain the correct color with $f_{fb}(\mathbf{c}_2, f_{fb}(\mathbf{c}_1, \mathbf{c}_3))$ though.

Figure 14 shows the hardware pipeline for quad merging. Since the conventional hardware graphics pipeline processes quads as the smallest unit of granularity, our goal is to identify

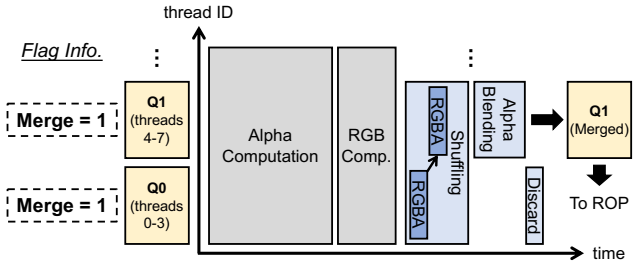

Fig. 15: Execution timeline of fragment shader with quad merging.

overlapping quads within a warp and *merge* them into a single quad by performing partial blending in the shader cores. To do so, we propose hardware and software extensions designed to merge the quads as much as possible.

On the hardware side, we introduce a *tile grid* coalescing (TGC) unit and a quad reorder unit. Without the TGC unit, the TC unit simply gathers quads that belong to the same screen tile into the TC bin, and the subsequent quad reorder unit attempts to find overlapping quads. However, the limited number of TC bins causes them to be frequently flushed before collecting a sufficient number of quads to be merged, particularly when primitives are large or spatially distributed. To increase the opportunity to merge quads, the TGC unit first gathers the primitives that intersect the same *tile grid* into the TGC bin. Note that the *tile grid* is larger than a screen tile but is still smaller than the region that all TC bins in the TC unit can cover. When the TGC bin is flushed, the rasterizer processes the primitives within the *tile grid* and generates quads that can be handled by a set of TC bins, thereby mitigating the premature flushing of the TC bins.

When the TC bin is flushed, the quad reorder unit (QRU) identifies the overlapping quads and reorders them to launch together in the same warp. The input quads are stored in the buffer and assigned sequential quad IDs (QIDs) from 0 to 127. For overlap detection, the QRU maintains 64 8-bit (consisting of a valid bit and 7-bit quad ID) registers, each corresponding to a relative quad location in a screen tile from position $(0, 0)$ to $(7, 7)$. The unit sequentially examines each quad starting from QID 0 and stores its QID in the register corresponding to its location. If the register already contains a valid QID, this indicates an overlap between two quads. Upon detecting overlapping quads, the QRU adds the pair into a warp with a merge flag indicating that they need to be merged in the fragment shader. The unit also manages a 128-bit bitmap where each bit represents whether the quad will be merged. After examining all quads, the QRU fills the warps with the quads that will not be merged using the bitmap.

On the software side, we propose an extension that can be added to the original shader for quad merging. Figure 15 illustrates the execution flow of the fragment shader with the proposed software extension. After completing the original fragment shading operations, the threads marked with a merge flag engage in partial blending. In the figure, for instance, two quads at quad offset 0 (threads 0-3) and 1 (threads 4-7) need to be merged. Since two quads to be merged are always adjacent in the warp by the quad reorder unit, the quad at $(2n + 1)$ quad offset retrieves the fragment colors from the quad at $2n$ offset using warp shuffling and blends them with its own. Ultimately, a single merged quad is produced and sent to the ROP for final blending into the color buffer. By leveraging the high bandwidth of warp shuffling and the high compute throughput of shader cores, we can increase effective blending throughput, which would otherwise be limited by the number of ROP units.

TABLE I: Simulation configuration.

GPU	
# GPC	1
# SIMT Cores	16 (1024 CUDA Cores)
SIMT Core Freq.	612 MHz
Lanes per SIMT Core	64 (4 warp schedulers)
L1D/T	48 KB, 128B line
Shared L2	4 MB, 128B line (sectored)
CROP Cache Size	16 KB, 128B line (sectored)
Raster Tile Size	8×8 pixels
Tile Grid Size	64×64 pixels (4×4 tiles)
# of TGC Bins TGC Bin Size	128 16 primitives
# of TC Bins TC Bin Size	32 128 quads
ROP Throughput	2 quads/cycle (RGBA16F)
DRAM	LPDDR3-1600 (16-channel)

TABLE II: Evaluated workloads.

Dataset	Scene (Resolution / #Gaussians)	Type
Mip-NeRF 360 [5]	Kitchen (1552×1040 / 1.85M)	Real World & Indoor
	Bonsai (1552×1040 / 1.24M)	
Tanks&Temples [20]	Train (980×545 / 1.03M)	Real World & Outdoor
	Truck (979×546 / 2.54M)	
Synthetic-NeRF [32]	Lego (800×800 / 358K)	Synthetic
Synthetic-NSVF [27]	Palace (800×800 / 327K)	

VI. Evaluation

A. Experimental Methodology

Simulation Infrastructure. To evaluate the rendering performance of graphics workloads, we use the Emerald simulator [15] that builds on the gem5 and GPGPU-Sim simulators. Emerald models an SoC system that comprises both CPU and GPU cores. We use *standalone* (GPU) mode and implement our hardware extensions on the baseline GPU architecture that models the hardware stages in contemporary GPUs. Because Emerald does not implement the hardware units that we need to build on, we make extensive modifications to the codebase, based on our analysis on graphics hardware, which we discuss in Section VII. We set the GPU frequency and DRAM bandwidth to match those measured using Jetson AGX Orin in 30W power mode. We measure the execution cycles between the start of a draw call and the end of raster operations, which captures the performance of an end-to-end graphics pipeline. Table I shows the system configuration used in this work.

Workloads. Table II shows the workloads that we use to evaluate VR-Pipe. We carefully select six widely-used scenes from published datasets [5], [20], [32] to evaluate real-world and synthetic scenes with varying complexity. For each scene,

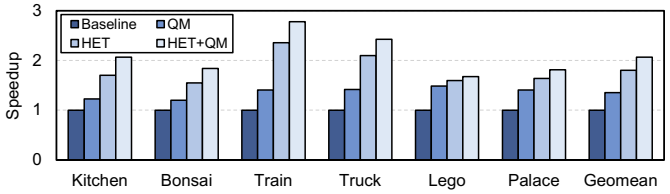

Fig. 16: Speedup of VR-Pipe over the baseline GPU.

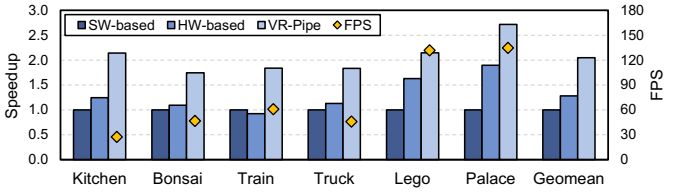

Fig. 17: Overall end-to-end speedups of VR-Pipe over software-based (CUDA) and hardware-based (OpenGL) rendering.

we train the model for 30K iterations, which results in approximately 300K to 2.5M Gaussians per scene. We generate the trace of OpenGL ES API calls using apitrace [1] and simulate the rendering at the full image resolution specified in the table. For the evaluated viewpoints, up to 900K Gaussians are within the viewing frustum.

B. Performance Results

Figure 16 shows the speedups from two hardware extensions in VR-Pipe: hardware-based early termination (HET) and quad merging (QM). By applying each component to the baseline graphics hardware, we evaluate four variants: Baseline, QM, HET, and HET+QM. QM reduces the amount of blending work for ROPs by partially blending the overlapping quads within shader cores. This improves rendering performance by up to 1.49× over the baseline. HET, on the other hand, excludes the fragments associated with terminated pixels before fragment shading. This effectively reduces the number of fragments to be processed in the hardware pipeline, thereby mitigating unnecessary computations and memory accesses for both shading and blending, providing a 1.80× speedup on average. Overall, by combining HET and QM, VR-Pipe provides an average speedup of 2.07× over the baseline GPU.

The performance gain varies across the scenes, as the benefits of early termination and quad merging are influenced by the scene size and viewpoint. For large, real-world outdoor scenes such as Train and Truck, we can achieve greater speedups with early termination since a relatively large number of Gaussians exist beyond the surface. In contrast, Bonsai exhibits a lower speedup among real-world scenes due to its inherent scene structure. With the object (i.e., the bonsai) positioned at the center and surrounded by a background room, the benefit of early termination is naturally concentrated in the central pixel region.

For high-resolution scenes like Bonsai and Kitchen, the performance improvement from quad merging is slightly lower compared to other scenes. This is because successive primitives span a larger number of tile grids, causing TGC bins to be flushed more frequently before being fully utilized. Note that in practice, the primitives of these high-resolution scenes are distributed across GPCs, which reduces the frequency of bin flushing. Thus, the benefit of quad merging would likely be higher than observed in our experiment. In summary, VR-Pipe consistently demonstrates performance improvements over the baseline GPU, highlighting its broad applicability across diverse scene types.

Overall End-to-End Rendering. Figure 17 compares the overall end-to-end performance estimation, which includes preprocessing and sorting, between different scenarios of 3D Gaussian rendering.[6] For a fair comparison, software-based (CUDA) rendering uses early termination, while hardware-based (OpenGL) rendering does not, as the baseline architecture lacks native support for early termination. On average, VR-Pipe can provide a 2.05× and 1.60× speedup over the SW-based (CUDA) and HW-based (OpenGL) rendering, respectively. Since the rasterization step, which is our optimization target, occupies more than 70% of total rendering time, VR-Pipe can effectively accelerate the overall end-to-end rendering process, achieving high frame rates (FPS).

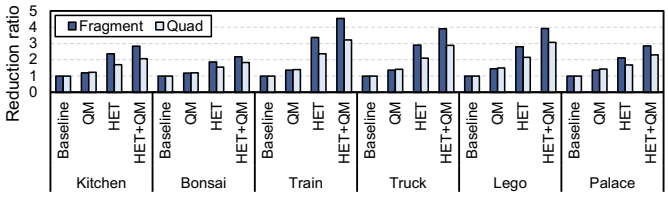

Fig. 18: Reduction ratio of quads and fragments blended by ROP.

C. Source of Performance Gain

Figure 18 shows the reduction in quads and fragments blended by ROPs, a key source of performance gains for both HET and QM. HET reduces the number of fragments by 2.52×, significantly lowering the amount of work by eliminating fragments before shading. QM further reduces the number of fragments by 1.30× on top of HET, increasing the overall blending throughput by utilizing shader cores for blending. For quads, HET achieves a 1.90× reduction, with QM providing an additional 1.32× reduction. The quad reduction of HET is smaller than its fragment reduction because quads are eliminated only when all their fragments are terminated. QM effectively reduces the number of quads by merging two into one. Consequently, by reducing ROP pressure, VR-Pipe achieves speedups in relation to the reduction in quads and fragments across the evaluated scenes.

D. Energy Efficiency

To estimate the energy efficiency of a draw call when VR-Pipe is implemented in mobile/embedded GPUs, we imitate

[6]As our evaluated GPU is configured similarly to AGX Orin, we observe a high correlation in execution time between the AGX Orin and our simulation for the preprocessing kernel. However, due to the limitations of Emerald, it is challenging to obtain correlated execution time for the sorting kernel that employs the NVIDIA CUB library. To better estimate the overall rendering performance, we use the execution time of preprocessing and sorting kernels on AGX Orin in this experiment.

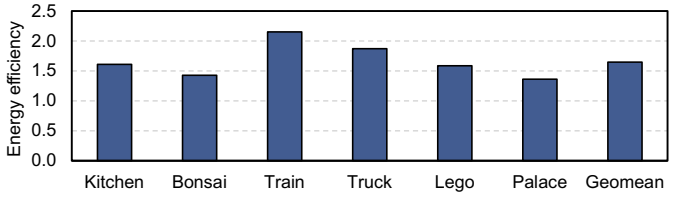

Fig. 19: Energy efficiency of VR-Pipe over the baseline GPU.

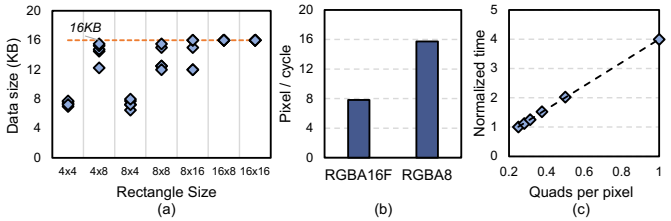

Fig. 20: (a) Data sizes fitting within the CROP cache across different rectangle sizes, measured by rendering rectangles at randomized positions. The scatter in data points shows variability in cache usage due to randomized positions tested. (b) Pixels per cycle when rendering the same amount of pixels using RGBA16F or RGBA8 color format. (c) Normalized rendering time with varying quads per pixel when using RGBA16F color format. The quads per pixel value is controlled by using a stencil test and adjusting the shape and position of primitives.

the effect of HET and QM in the AGX Orin. For HET, we first count the number of fragments until early termination for each pixel and store the numbers in the stencil buffer. During rendering, we then decrease the stencil value by 1 for each fragment until the value reaches zero. Once the value becomes zero, we discard subsequent fragments. We also manually add warp shuffling, blending, discard operations in the fragment shader, and make them operate as the same reduction ratio as QM in our simulator. As a result, our approach provides 1.65× higher energy efficiency on average (up to 2.15×) compared to the baseline GPU, as shown in Figure 19.

E. Implementation Cost

Table III shows the hardware cost of our proposed hardware extensions for volume rendering. We do not account for the cost of computational units because the bitwise operators used for alpha and early termination tests, the comparators in the tile grid coalescing unit, and the two floating point comparators in the alpha test unit would be considerably cheaper than the storage overhead. Our hardware extensions require only 25KB of storage, which is negligible considering it is for each GPC, not for a single SM.

VII. Analysis and Discussion

A. Analysis on Real Graphics Hardware

Emerald [15] does not implement, or approximately models, the graphics hardware units that we need to build upon (e.g., ROP, tile binning hardware). To address this, we created OpenGL-based microbenchmarks to explore the characteristics of these special-purpose units in real graphics hardware and properly model them in our simulation framework. The microbenchmarks render rectangles or triangles by adjusting various parameters, including positions, color formats, the number of involved screen tiles and rectangle overlaps. They are carefully designed and configured to stress the *unit of interest* while minimizing the impact on or from other units. For example, we restrict our analysis to a single GPC by positioning primitives within the screen tiles mapped to a specific GPC, as each GPC contains its own special-purpose units. Also, as modern GPUs employ color compression, we attempt to bypass it by generating different colors for each pixel via hashing. We run them on NVIDIA Ampere GPUs and measure the data using Nsight Graphics. OpenGL extensions including `GL_NV_shader_thread_group` and `GL_NV_shader_thread_shuffle` are also used to collect data from running warps formed through the graphics pipeline.

TABLE III: Hardware cost of VR-Pipe.

Hardware	Size
Tile Grid Coalescing Unit	(4B CBE pointer * 3 vertices * 16 entries + 2B tile grid ID) * 128 bins = 24.25KB
Quad Reorder Unit	(4B CBE pointer + 6-bit quad pos.) * 128 + 64 * 1B register + 16B bitmap = 688B
Total	24.92KB

Render Output Unit (ROP). The ROP units are now part of the GPC from the Ampere architecture to increase the performance of raster operations [36]. A single GPC has two ROP partitions, each containing eight ROP units. A single ROP unit can process a single fragment by fetching the corresponding pixel color of the image. We observe that ROP units do not fetch pixel colors directly from the L2 cache but first access another cache-like structure in each GPC, and we refer to it as a CROP cache. If a cache miss occurs, the color values are fed from the L2 cache.

We first measure the size of the CROP cache by rendering multiple rectangles while inspecting the L2 cache bandwidth consumption from CROP. Figure 20(a) shows the size of pixel color data before CROP starts accessing the L2 cache while increasing the number of rectangles. For example, we first draw an 8×16-pixel rectangle using the RGBA16F[7] (FP16) pixel format, which corresponds to 1KB. We then add more rectangles in random positions until the CROP starts accessing the L2 cache. The results show that 8×16-pixel rectangles can be drawn at 16 locations without L2 cache access. Over various rectangle sizes and positions, the CROP cache has never held more than 16KB of data, suggesting its size is likely 16KB.

Figure 20(b) shows that the number of pixels per cycle that CROP can process depends on the color format of the rendered image. For example, the pixels-per-cycle throughput is higher when using the RGBA8 (UNORM8) pixel format compared to RGBA16F. For RGBA8, which is 32 bits per pixel (bpp), a GPC with 16 ROP units can process 16 pixels per cycle, whereas it can process only 8 pixels per cycle when using

[7]Each red, green, blue, and alpha channel is a 16-bit floating-point number.

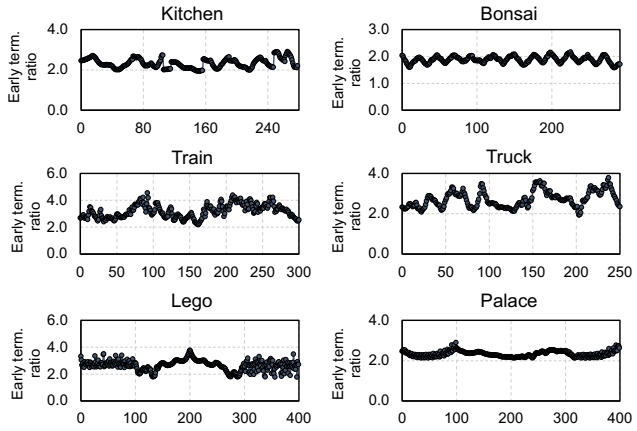

Fig. 21: Early termination ratio of varying viewpoints.

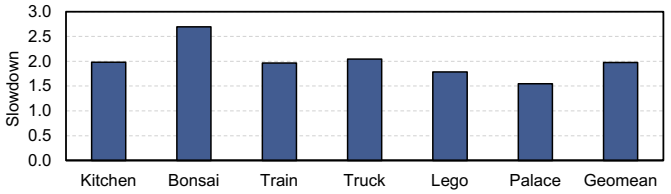

Fig. 22: Performance comparison with GSCore [24].

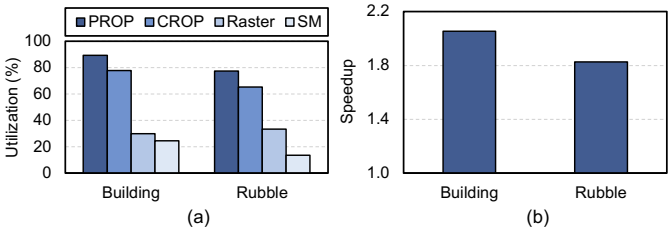

Fig. 23: (a) Unit utilization and (b) speedup for large-scale scenes.

the RGBA16F format, which is 64 bpp. This implies that the effective read bandwidth of the CROP cache is likely to be larger than 64 bytes per cycle. We also observe that ROPs cannot achieve the maximum pixels-per-cycle throughput if some pixels in a quad are partially discarded, as shown in Figure 20(c). This implies that the ROP units operate at a quad granularity; i.e., four ROP units operate together to process a 2×2-fragment quad.

Tile Binning. We analyze the behavior of tile binning while varying the number of screen tiles (N) involved in rendering. To investigate the number of tile bins, we draw *distinct* primitives (2×2-pixel rectangles) that are distributed across N screen tiles and measure the number of warps launched. We arrange the rectangles to visit the screen tiles in a round-robin manner (i.e., a repeating sequence of 1, 2, ..., N, 1, 2, ...), which helps us clearly observe the flushing effect. Note that each bin collects only the quads within the same screen tile, as previously discussed.

We observe that quads from distinct rectangles at the same pixel position but from different rounds are binned and launched together mostly as a single warp if N is less than or equal to 32. However, once N exceeds 32, rectangles within the same screen tile but from different rounds are launched as separate warps. For example, drawing 320 rectangles across 32 screen tiles results in 67 warps being launched. In contrast, drawing 330 rectangles across 33 screen tiles leads to the launch of 330 warps (i.e., each warp contains only a single quad). This occurs because, after the 32nd primitive, the binning of the 33rd primitive—rendered on the 33rd screen tile—triggers the flushing of one of the bins, resulting in the launch of an underutilized warp. This pattern continues for the remaining primitives, which indicates that each GPC in the evaluated GPUs has 32 tile bins.

B. Early Termination Ratio of Varying Viewpoints

Figure 21 shows the early termination ratio across different viewpoints. We evaluate all the viewpoints that are provided in the dataset. The early termination ratio is the ratio between the number of blended fragments processed with and without early termination. A higher ratio indicates a greater potential for performance improvements through early termination. As discussed in Section VI-C, outdoor scenes exhibit higher average ratios than indoor or synthetic scenes because there are more Gaussians beyond the surface in larger scenes. For example, up to nearly 4.4 times more fragments could be unnecessarily blended without early termination in the Train scene. While the ratio varies depending on the scenes, every scene shows an average ratio greater than 1.5, indicating that more than 33% of the fragments can be eliminated by early termination. This demonstrates that supporting early termination in hardware is a key factor for efficient volume rendering.

C. Discussions

Comparison to Accelerators for 3D Gaussian Splatting. Figure 22 compares the performance of VR-Pipe with a specialized accelerator for 3D Gaussian splatting, GSCore [24]. The results show that GSCore performs better than VR-Pipe due to its tailored design for Gaussian splatting. As a dedicated accelerator, however, GSCore is inherently limited to running graphics workloads involving Gaussian splatting and requires custom compilers and runtime. VR-Pipe, on the other hand, extends the capabilities of existing graphics hardware and runs standard graphics APIs, offering greater flexibility for rendering both traditional rasterization and volume rendering tasks. We expect that future acceleration efforts will involve both building specialized accelerators and enhancing existing graphics hardware, depending on deployment requirements.

Scalability for Larger or More Complex Scenes. As mentioned in Section II-A, Gaussian primitives are rendered in a tile-based manner, and thus the benefit of VR-Pipe can be easily extended to very large-scale scenes as long as they fit within the GPU memory. For larger or more complex scenes, such as Building (9.06M Gaussians) and Rubble (5.21M Gaussians) used in Mega-NeRF [45] and CityGaussian [29], there is a significantly higher number of Gaussians that need blending, which makes ROPs a persistent bottleneck, as shown in Figure 23(a). Consequently, VR-Pipe helps improve rendering performance for these large-scale scenes, as shown in Figure 23(b).

Potential for Future Deployments in GPUs. Both hardware early termination (HET) and quad merging (QM) could potentially be adopted in GPUs, as the main hardware overhead is only tens of kilobytes of storage for the bins in the TGC unit and a few registers in the quad reorder unit, as discussed in Section VI-E. Between HET and QM, we envision that HET may have great potential for direct deployment, as it offers solid performance benefits across different scene types, as shown in Figure 16, and is relatively simple to implement without much hardware overhead. As for QM, it is noted that the storage overhead added for QM is per GPC, not per SM. Given that the baseline GPU (Jetson AGX Orin) contains at least 3.6MB of SRAM in a single GPC, the additional overhead for QM might also be acceptable.

VIII. Related Work

Efficient Radiance Field Rendering. The introduction of Neural Radiance Fields (NeRF) [32] has generated significant interest in efficient 3D scene representation and rendering for radiance fields. Over the past years, there has been a large amount of research aimed at accelerating NeRFs through algorithmic or software optimizations [7], [13], [34], [43], and the development of hardware accelerators [12], [23], [25], [33], [42]. The state-of-the-art method, 3D Gaussian splatting [18], has further fueled interest in accelerating radiance field rendering [16], [22], [24], [35], [38] as it employs rasterization primitives that can be rendered much faster than NeRFs. However, previous research focused on software graphics rendering on programmable cores or building dedicated hardware accelerators. In contrast, VR-Pipe investigates the potential of efficient radiance field rendering while utilizing fixed-function units in graphics hardware. To our knowledge, this is the first work that assesses the performance implications of rendering Gaussian-based radiance fields on the hardware graphics pipeline with software and hardware optimizations.

Enhancing Graphics Rendering Hardware. The performance advantage of executing graphics rendering on either programmable shader cores or fixed-function units varies depending on the rendering methods and hardware designs. Previous studies have explored the performance implication of graphics hardware design by developing simulation infrastructures for graphics workloads [3], [9], [15], [44]. Additionally, several studies have aimed to improve the performance of special-purpose hardware such as ray tracing units in graphics hardware [8], [28] and proposed hardware accelerators for graphics applications [30], [39]. In contrast to these works, which primarily evaluate traditional graphics workloads, our work focuses on improving the performance of volume rendering workloads, such as Gaussian splatting, which require blending a huge number of fragments per pixel.

In the context of multi-sample anti-aliasing, prior work proposed reducing the amount of redundant shading by merging fragments from adjacent triangles in a mesh at the quad granularity [11]. While both our work and quad-fragment merging (QFM) [11] aim to reduce operations by merging quads, our proposed technique differs from QFM in many aspects. Our method aims to blend *overlapping primitives* along the depth direction and applies to quads from any primitive. In contrast, QFM merges quad fragments from small (e.g., pixel-sized) triangles that *share* an edge (i.e., *connected*, *non-overlapping* triangles). As such, QFM is not applicable to the scenes consisting of a number of unconnected transparent triangles, such as those in 3D Gaussian splatting. In addition, our method computes the *exact* color for each pixel by offloading blending operations from ROPs to shader units, whereas QFM *approximates* pixel colors by using the color from one triangle when multiple triangles are merged into a single quad.

IX. Conclusion

Gaussian splatting is becoming one of the key techniques in graphics rendering, yet its performance implication on the hardware graphics pipeline has remained unexplored thus far. Through hardware graphics rendering and software optimization using a standard graphics API, we show that hardware-based volume rendering can match or surpass the performance of software-based rendering that relies solely on shader cores. Nonetheless, we also observe that there is potential for *native* hardware support for volume rendering workloads in graphics hardware. Based on the analysis on modern graphics hardware, we introduce VR-Pipe, which integrates hardware-based early termination and multi-granular bins with quad merging into the special-purpose units in GPUs. These features greatly improve the performance of rendering scenes via volume rendering such as Gaussian splatting. We anticipate that these features will also benefit graphics and game engines that traditionally run on graphics-specific hardware.

Acknowledgment

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by the Institute for Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT) (RS-2023-00256081, RS-2024-00395134) and a research grant from Samsung Electronics Co., Ltd. (0418-20230064, A0342-20200002). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

References

[1] apitrace, "apitrace," 2018. [Online]. Available: https://github.com/apitrace/apitrace

[2] Apple, "Metal." [Online]. Available: https://developer.apple.com/metal/

[3] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems," in *Proceedings of the 27th International Conference on Supercomputing (ICS)*, 2013.

[4] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[5] 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.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[7] 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.

[8] 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.

[9] V. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and E. E, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2006.

[10] Epic Games, "Unreal engine." [Online]. Available: https://www.unrealengine.com/en-US

[11] K. Fatahalian, S. Boulos, J. Hegarty, K. Akeley, W. R. Mark, H. Moreton, and P. Hanrahan, "Reducing shading on gpus using quad-fragment merging," *ACM Transactions on Graphics (TOG'10)*, 2010.

[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] S. Fridovich-Keil, A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance fields without neural networks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[14] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: a hierarchical structure for rapid interference detection," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1996.

[15] A. A. Gubran and T. M. Aamodt, "Emerald: graphics modeling for soc systems," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[16] A. Hamdi, L. Melas-Kyriazi, J. Mai, G. Qian, R. Liu, C. Vondrick, B. Ghanem, and A. Vedaldi, "Ges : Generalized exponential splatting for efficient radiance field rendering," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[17] Intel Corporation, "Intel processor graphics gen11 architecture." [Online]. Available: https://cdrdv2-public.intel.com/686065/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf

[18] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics (SIGGRAPH)*, vol. 42, no. 4, July 2023.

[19] Khronos Group, "Vulkan." [Online]. Available: https://registry.khronos.org/vulkan

[20] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and temples: benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics (SIGGRAPH)*, 2017.

[21] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *Proceedings of IEEE Visualization Conference (VIS)*, 2003.

[22] J. C. Lee, D. Rho, X. Sun, J. H. Ko, and E. Park, "Compact 3d gaussian representation for radiance field," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[23] 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.

[24] 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.

[25] 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.

[26] J. E. Lindholm, H. P. Moreton, J. S. Montrym, and S. R. Whitman, "Apparatus and method for raster tile coalescing," U.S. Patent 7 564 456, Jul. 21, 2009. [Online]. Available: https://patents.google.com/patent/US7564456

[27] L. Liu, J. Gu, K. Z. Lin, T.-S. Chua, and C. Theobalt, "Neural sparse voxel fields," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[28] 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.

[29] Y. Liu, C. Luo, L. Fan, N. Wang, J. Peng, and Z. Zhang, "Citygaussian: Real-time high-quality large-scale scene rendering with gaussians," in *Proceedings of European Conference on Computer Vision (ECCV)*, 2024.

[30] Y. Lü, L. Huang, L. Shen, and Z. Wang, "Unleashing the power of gpu for physically-based rendering via dynamic ray shuffling," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.

[31] Microsoft, "Direct3d." [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d

[32] 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.

[33] 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.

[34] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Transactions on Graphics (SIGGRAPH)*, 2022.

[35] S. Niedermayr, J. Stumpfegger, and R. Westermann, "Compressed 3d gaussian splatting for accelerated novel view synthesis," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

[36] NVIDIA, "Nvidia ampere ga102 gpu architecture," 2021. [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.1.pdf

[37] T. J. Purcell and S. E. Molnar, "Rasterization tile coalescer and reorder buffer," U.S. Patent 8 605 102, Dec 10, 2013. [Online]. Available: https://patents.google.com/patent/US8605102

[38] 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.

[39] K. Ramani, C. P. Gribble, and A. Davis, "Streamray: a stream filtering architecture for coherent ray tracing," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[40] J. S. Rhoades, S. E. Molnar, E. M. Kilgariff, M. C. Shebanow, Z. S. Hakura, D. L. Kirkland, and J. D. Kelly, "Distributing primitives to multiple rasterizers," U.S. Patent 8 704 836, Apr. 22, 2014. [Online]. Available: https://patents.google.com/patent/US8704836

[41] M. Segal and K. Akeley, "The opengl graphics system: A specification." [Online]. Available: https://www.opengl.org/registry/doc/glspec46.core.pdf

[42] 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.

[43] 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.

[44] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper, and H. Kim, "Skybox: Open-source graphic rendering on programmable risc-v gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

[45] H. Turki, D. Ramanan, and M. Satyanarayanan, "Mega-nerf: Scalable construction of large-scale nerfs for virtual fly-throughs," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[46] Unity Technologies, "Unity engine." [Online]. Available: https://unity.com/products/unity-engine

[47] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi gf100 gpu architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.