GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting

Junseo Lee Seokwon Lee Jungi Lee Junyong Park Jaewoong Sim

Seoul National University

{junseo.lee, seokwon.lee, jungi.lee, junyong.park, jaewoong}@snu.ac.kr

Abstract

This paper presents GSCore, a hardware acceleration unit that efficiently executes the rendering pipeline of 3D Gaussian Splatting with algorithmic optimizations. GSCore builds on the observations from an in-depth analysis of Gaussian-based radiance field rendering to enhance computational efficiency and bring the technique to wide adoption. In doing so, we present several optimization techniques, Gaussian shape-aware intersection test, hierarchical sorting, and subtile skipping, all of which are synergistically integrated with GSCore. We implement the hardware design of GSCore, synthesize it using a commercial 28nm technology, and evaluate the performance across a range of synthetic and real-world scenes with varying image resolutions. Our evaluation results show that GSCore achieves a 15.86× speedup on average over the mobile consumer GPU with a substantially smaller area and lower energy consumption.

CCS Concepts: • **Computer systems organization** → **Architectures**; • **Computing methodologies** → **Rendering**.

Keywords: Accelerators, Gaussian Splatting, Rendering

ACM Reference Format:

Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, Jaewoong Sim. 2024. GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651385

1 Introduction

Recent advances in graphics rendering that build on machine learning and radiance fields are gaining significant attention due to their outstanding quality in synthesizing photorealistic images from novel viewpoints [4, 15, 25, 33, 36].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651385

Figure 1. Rendering with 3D Gaussian Splatting [25] on Jetson Xavier NX and GSCore. The rendered image for GSCore is obtained from our simulator. The model is trained for 30K iterations.

While traditional 3D scene reconstruction methods often struggle to capture the intricate geometric and spatial details of scenes, modern radiance field-based techniques, such as Plenoxels [15] and NeRF [33], enable us to capture the fine details in 3D scenes through the use of *continuous* volumetric radiance fields and *differentiable* rendering primitives.

3D Gaussian Splatting [25] is a recent breakthrough that offers a promising solution for representing the radiance field. Unlike Neural Radiance Fields (NeRF), where scenes are *implicitly* represented, Gaussian Splatting makes use of *explicit* rasterization primitives (i.e., Gaussians), thereby enabling much faster rendering than NeRF, along with scene editing properties. As such, the technique already starts being supported by a variety of popular rendering engines, including Unity [49], Unreal [16], and Blender [13].

While 3D Gaussian Splatting offers remarkable rendering performance with better image quality than prior methods, achieving real-time rendering for several important domains today, such as virtual reality and mobile computing, remains challenging, as depicted in Figure 1. For instance, the Oculus Quest 2, which features the Adreno 650 GPU, is only marginally better than the Jetson Xavier NX in Figure 1. Given that the performance of mobile platforms is primarily constrained by their power budget, it is unlikely that mobile GPUs will become as powerful as desktop or server-class GPUs, such as NVIDIA RTX 3090 or A100. This hinders us from extending the benefits of the technique to broader computing domains.

In this work, we begin by investigating the Gaussian rendering pipeline and conducting an in-depth characterization of the technique on today's computing platform to understand its architectural implications. Our analysis shows that Gaussian sorting and rasterization are two key contributors to the rendering time of Gaussian Splatting. Further investigations, however, reveal that the performance bottlenecks

are partially attributed to the naïve intersection test between Gaussians and pixels in the preprocessing step. In addition, we observe that a significant number of Gaussians do not actually contribute to the pixel color during the alpha blending process due to the nature of Gaussians, early termination, and tile-based lockstep execution on GPUs.

Based on our observations, we propose the optimization techniques—Gaussian shape-aware intersection test, hierarchical sorting, and subtile skipping—to streamline the rendering pipeline of Gaussian Splatting, which we discuss in Section 4. Alongside the optimizations, we present a hardware acceleration unit called Gaussian Splatting Core (GSCore), which specializes in Gaussian-based radiance field rendering. GSCore leverages our key observations to efficiently perform Gaussian sorting and rasterization, which are two time-consuming operations, and delivers substantial improvements in performance and energy efficiency over today's mobile GPUs.

We implement GSCore using SystemVerilog and synthesize it with a commercial 28nm technology. In addition, to evaluate its performance, we implement a cycle-level simulator that models the hardware components of GSCore with a detailed off-chip memory model. Our simulator also produces functional outputs (i.e., pixel colors), which we use to examine the rendered image. We evaluate GSCore across synthetic and real-world scenes that are widely used in the graphics community. Our results show that GSCore achieves a 15.86× speedup on average over the representative mobile Volta GPU, with a small area budget of 3.95mm^2.

In summary, this paper makes the following contributions:

- To our knowledge, this is the *first* work to provide an in-depth analysis of Gaussian-based radiance field rendering on today's computing platforms and identify the root causes of performance inefficiencies.
- We propose optimization techniques to streamline the Gaussian rendering pipeline and perform volume rendering more efficiently than general-purpose computing platforms, complemented by architectural support.
- We present GSCore, a hardware acceleration unit tailored to the needs of Gaussian-based radiance field rendering. GSCore enables real-time rendering of novel view synthesis with substantially small area and energy consumption, thereby extending the benefit of 3D Gaussian Splatting across various computing domains.

2 Background

In this section, we first provide the background of novel view synthesis and modern volume rendering techniques that build on radiance field methods. We then introduce 3D Gaussian Splatting, the state-of-the-art radiance field rendering technique that outperforms other methods in terms of rendering quality and performance.

2.1 Volume Rendering with Radiance Fields

3D Scene Reconstruction and Rendering. Traditional 3D scene reconstructions rely on *explicit* representations such as manually-crafted meshes and point clouds. The emergence of Structure-from-Motion (SfM) [46] paves the way for an entirely new approach, which enables the synthesis of novel views using a collection of photos. Using SfM, one can generate 3D point clouds from a set of 2D images, which are then used to create a mesh structure. With the mesh representation, one can generate a new scene from any viewpoint.

Neural Radiance Fields (NeRF). While traditional scene reconstruction methods produce reasonable images from novel viewpoints, they typically struggle to completely recover from unreconstructed or complex regions. Neural Radiance Fields (NeRF) [33] has recently emerged as a promising method, which allows us to synthesize *novel* views of complex 3D scenes using a partial set of 2D images and neural networks. In NeRFs, scenes are represented *implicitly* through the weights of multi-layer perceptrons (MLPs) that represent the radiance field.[1] The MLP weights are trained directly with a sparse set of 2D images (without using SfM), and the learned weights are used for rendering an image from a specific viewpoint.

The NeRF model generally comprises two MLPs: one for view-independent density and another for view-dependent color. To render an image, we first cast rays from the origin of a viewpoint to each pixel and sample the points along the rays. We then feed the encoded features of the sampled points into the MLPs to obtain the density (σ_i) and color ($\mathbf{c}_i$) values for each point. The density and color values of all sampled points are subsequently accumulated through the volume rendering (α-blending) process to produce the pixel color (C), as shown in Equation 1:

$$\mathbf{C} = \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathrm{T_i}\alpha_\mathrm{i}\mathbf{c}_\mathrm{i}, \tag{1}$$

where $\mathrm{T_i} = \Pi_{\mathrm{j}=1}^{\mathrm{i}-1}(1-\alpha_\mathrm{j})$ and $\alpha_\mathrm{i} = 1 - \exp(-\sigma_\mathrm{i}\delta_\mathrm{i})$. Transmittance $\mathrm{T_i}$ represents the probability of the ray reaching a point without encountering any obstacles, which is calculated using the density (σ_i) and the distance between adjacent samples (δ_i). While NeRF offers impressive rendering quality, it is computationally expensive due to the use of MLPs for *every* sampled point, which is a challenge that several recent works aim to address [7, 17, 36]. In addition, the implicit representation makes it difficult to edit the scene.

2.2 3D Gaussian Splatting

3D Gaussian Splatting [25] is a recent breakthrough that achieves both outstanding rendering quality and performance, along with scene editing capability. It achieves this by leveraging rasterization primitives that are also differentiable

[1] The radiance field is defined by the radiance at every point and direction in 3D space.

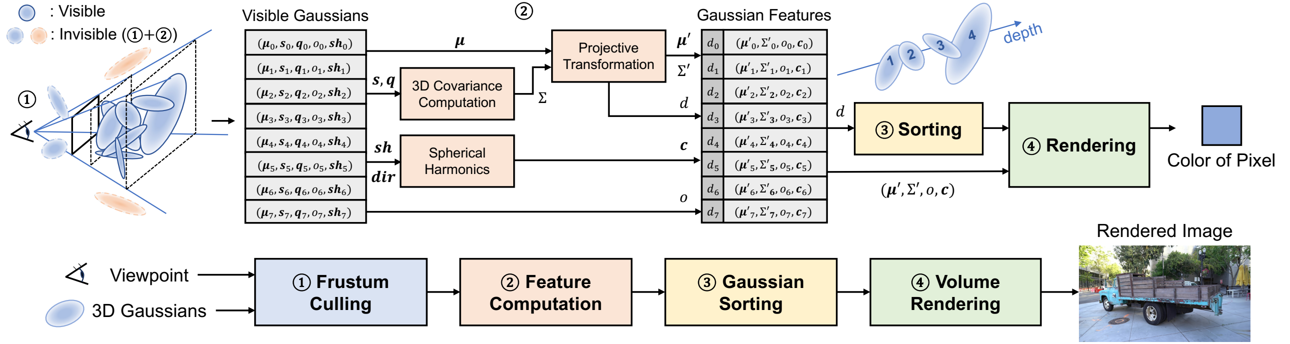

Figure 2. Radiance field rendering pipeline for 3D Gaussian Splatting.

Table 1. Parameters for 3D Gaussian.

Parameter	Symbol	Size	Note
Position (mean)	μ	3	3D vector (x, y, z)
Scale	s	3	3D vector (x, y, z)
Rotation (quaternion)	q	4	scalar + 3D vector (i, j, k)
Opacity	o	1	scalar
SH coefficients	sh	48	4 bands of SH; (1+3+5+7)×3
Total (per Gaussian)		59	

unlike meshes or points. In 3D Gaussian Splatting, a scene is represented by a dense set of 3D *anisotropic* Gaussians, instead of neural networks, each of which has the properties of a position (mean) vector μ, a 3×3 covariance matrix Σ, opacity o, and spherical harmonic (SH) coefficients sh that represent the directional appearance component (color) of the radiance field.[2]

To begin with, a sparse set of points is first constructed in 3D space (i.e., a point cloud) from the 2D images using SfM. Each SfM point is then assigned to a 3D Gaussian, forming an initial sparse set of Gaussians. The features of each Gaussian are *learned* during the training process using conventional gradient-based algorithms in machine learning. It is noted that scenes are *explicitly* represented by 3D Gaussians, leading to much faster rendering than NeRFs while preserving the advantages of volume rendering.

During training, small Gaussians in under-reconstructed regions are cloned, while large Gaussians with high variance are split into smaller ones. This allows for effectively representing the 3D geometry with the control of the Gaussian density. Also, instead of directly optimizing the covariance matrix Σ, the matrix is decomposed into a rotation matrix (R) and a scaling matrix (S), each of which is independently optimized. Then, the covariance matrix is computed using these two matrices; $\Sigma = (RS)(RS)^T$. This facilitates optimizing the covariance features and ensures that the covariance matrix becomes positive semi-definite.[3] Table 1 shows the parameters used for each 3D Gaussian.

Normally, a 3D Gaussian is a *continuous* function defined throughout the entire 3D space. To improve the effectiveness of training and rendering, however, each 3D Gaussian is treated as an *ellipsoid* (i.e., a closed surface) in Gaussian Splatting; that is, a set of *discrete* ellipsoids represents the 3D scene. When the viewpoint and its image plane are given, the ellipsoids are then projected (i.e., "splatted") onto the 2D plane as *ellipses* (*2D splats*). During rendering, we accumulate the color and density of the splats that intersect the pixels. It is noted that NeRFs and Gaussian Splatting share the same volume rendering equation (Equation 1). In the following section, we discuss the rendering pipeline of Gaussian Splatting, which our work targets for optimization.

3 Motivation

In this section, we begin by explaining the rendering pipeline of the 3D Gaussian splatting technique. We then identify the key operations that contribute to the overall rendering time and discuss our observations, which motivate our work.

3.1 Radiance Field Rendering with 3D Gaussians

As discussed in Section 2.2, 3D scenes are reconstructed with a number of Gaussians (N), each of which has the features that are learned during training. Using these Gaussians, we render a 2D image from any viewpoint, which is largely divided into four steps: Frustum Culling, Feature Computation, Gaussian Sorting, and Rasterization (Volume Rendering), as illustrated in Figure 2.

Frustum Culling. To begin with, we first eliminate the Gaussians that are invisible from the camera viewpoint, a process known as *frustum culling*. This involves iterating through all N Gaussians that represent the radiance field and checking whether they are inside the viewing frustum. During this step, only the mean position (μ) of each Gaussian

[2] Spherical harmonics (SH) are a set of functions defined on the surface of a sphere, which are widely used for representing *view-dependent* radiance in computer graphics [43].

[3] The covariance matrix of a multivariate Gaussian *must* be positive semi-definite.

is used for culling after obtaining its depth,[4] so there is no need to read all Gaussian features from off-chip memory. The depth (d) of a Gaussian refers to the z value of its mean position in the 3D view space when the camera is positioned at the origin with its viewing direction aligned along the z-axis. Note that the number of Gaussians to process is reduced to N' after frustum culling.

Feature Computation. Once we obtain (potentially) visible Gaussians (N') for a given viewpoint, we now perform feature computation to obtain the features used for the rest of the pipeline. This step involves both projecting a 3D Gaussian onto the 2D image space and obtaining an RGB color (c) of the Gaussian. For the projection process, we first derive a covariance matrix (Σ) from a scaling (s) vector and a quaternion (q). Then, the Gaussian is projected onto the 2D plane by transforming the 3D mean (μ) and covariance (Σ) into their 2D counterparts (μ' and Σ').

After the projection, we also set a bounding rectangle for the projected Gaussian (i.e., *2D splat*) while computing its extent to determine whether it overlaps with the screen. For the splats that overlap with the screen, the color is computed using SH coefficients (sh) and a viewing direction (dir). Initially, the feature dimension of each Gaussian is quite large (i.e., 59), primarily due to the large number of SH coefficients. This step greatly reduces the feature dimension because the color is now represented as a 3D RGB vector, which helps reduce memory bandwidth consumption when performing volume rendering.

Gaussian Sorting. Once we compute the new features for each Gaussian, we need to obtain a *sorting order* of the Gaussians. This is necessary because volume rendering accumulates the pixel color by traversing the Gaussians front-to-back that overlap the pixel, as shown in Equation 1. This step sorts the Gaussians with respect to the depth (d) for rasterization. Given that the number of Gaussians (N') can be hundreds of thousands or millions, this sorting process takes up a non-negligible portion of rendering time, even when using a well-optimized parallel sorting implementation on GPUs (e.g., a radix sort in the NVIDIA CUB library).

Rasterization. Lastly, we perform rasterization (volume rendering) using the sorted Gaussians to obtain pixel colors. The volume rendering process is inherently parallel because the color computation for each pixel is independent of the others. Additionally, the same volume rendering equation employed in other methods [15, 33, 36] is also used in this step (i.e., Equation 1), as they essentially share the same image formation model. Thus, optimization techniques, such as early ray termination used in volumetric ray-marching [36], can also be employed to reduce the rendering time.

For an effective use of hardware resources, the state-of-art GPU implementation [25] employs conventional *tile-based rendering* (TBR) by dividing the image into several tiles, each of which contains 16×16 pixels. Each thread block then independently renders a 16×16-pixel tile (i.e., 256 threads per 256 pixels) with early termination.

[4] To be precise, only *near-plane culling* is performed in this step; other Gaussians outside the frustum are discarded after image-space projection [25].

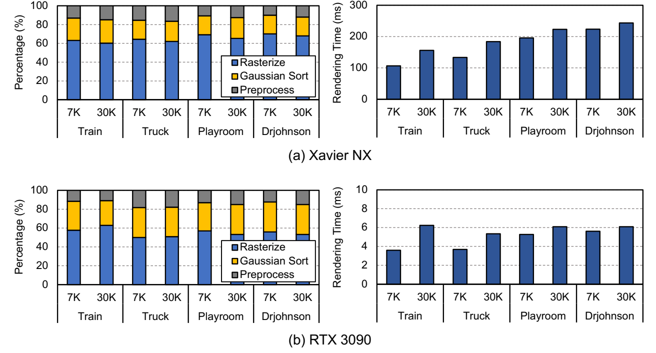

Figure 3. Rendering time breakdown. Each scene is trained with 7K and 30K iterations. We render an image for each scene from a single viewpoint. See Table 3 for the details of each scene.

The rasterization process can be largely divided into five steps: Gaussian fetching, α-computation, α-pruning, early (ray) termination, and α-blending. We first fetch the Gaussian features and compute the α value using Equation 2:

$$\alpha_i = o_i * \exp(-\frac{1}{2}(p' - \mu')^T \Sigma'^{-1} (p' - \mu')), \tag{2}$$

where p' is the pixel position and o_i is the opacity of the Gaussian. The α-computation consumes most of the execution time in rasterization as it involves multiple floating-point computations for vector and exponential operations. In the α-pruning and early termination steps, we skip the Gaussian not to be used for the transmittance and pixel color if α is less than a pre-defined threshold ϵ (e.g., $\epsilon=\frac{1}{255}$ in [25]) for numerical stability. Additionally, we check whether the transmittance (T_i) falls below a threshold (e.g., 10^{-4} in [25]) for early termination. If not, we accumulate the color of the Gaussian to the pixel color in the α-blending step using Equation 1. In the following sections, we investigate the performance aspects of these steps in the rendering pipeline and discuss our observations and optimization opportunities.

3.2 Performance Characterization

To understand the rendering performance of Gaussian Splatting, we run the state-of-the-art implementation [25] with several real-world scenes on the Jetson Xavier NX [39], which is representative of edge platforms for vision systems [2], as well as on a desktop-class RTX 3090 GPU.

Figure 3 decomposes the rendering time into three major components: preprocessing (which combines frustum culling and feature computation), Gaussian sorting, and rasterization. The results show that rasterization and Gaussian sorting are two major performance bottlenecks across different scenes and platforms, collectively occupying around 80% of rendering time. For the same scene and computing platform,

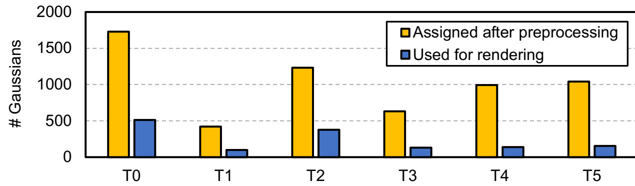

Figure 4. Number of Gaussians assigned to tiles and processed for rendering among the assigned ones in each tile. The results are obtained from randomly selected tiles for Train (30K iterations).

training with 30K iterations generally leads to more Gaussians than with 7K iterations due to the densification process (i.e., splitting/cloning Gaussians), as discussed in Section 2.2. As such, the rendering time for the 30K model generally takes longer than that of the 7K model for the same viewpoint.

The results further demonstrate that today's edge devices would still suffer from slow rendering time. Although the desktop-class GPU offers outstanding rendering performance (> 100 FPS), the edge platform renders an image at only a few frames per second (FPS). The performance of mobile and edge platforms is primarily dictated by their power budget, which is challenging to increase, particularly for battery-powered devices. This motivates us to perform Gaussian-based rendering more efficiently to bring the technique to wide adoption. In doing so, instead of directly accelerating sorting and rasterization steps, we carefully investigate the root cause of the performance bottlenecks, which we discuss in the following section.

3.3 Opportunities for Efficient Gaussian Rendering

We observe that there are several inefficiencies in the state-of-the-art Gaussian-based rendering pipeline.

Observation I: Unnecessary Assignment of Unused Gaussians to Tiles. Figure 4 shows the number of Gaussians assigned to tiles after frustum culling and feature computation, along with the actual number of Gaussians processed for rendering in each tile. In Gaussian Splatting, object shapes are *approximated* using a set of Gaussians. Thus, it is necessary to identify the Gaussians to process for each tile (i.e., 16×16 pixels) for rasterization. In the state-of-the-art implementation, this is achieved by performing an intersection test between the tiles and Gaussians using an *axis-aligned bounding box (AABB)* during the preprocessing step. Note that the same Gaussian can be assigned to *multiple* tiles if it overlaps with a large number of pixels.

While the intersection test with AABBs simplifies the assignment of Gaussians to tiles, which helps reduce the preprocessing time as shown in Figure 3, we observe that it results in a notably high number of *false positives*. To reduce the computation overhead of obtaining the AABB of a Gaussian, the original algorithm sets the AABB as a *square* that circumscribes a circle with a radius equal to the semi-major axis of an ellipse. This is computationally cheap because it only requires the center point of a Gaussian, which we already know (i.e., a Gaussian mean), and the radius of the major axis. However, the majority of the projected Gaussians (i.e., splats) are *anisotropic* (i.e., highly skewed along the major axis), so the AABB leads to considerable empty space between the ellipse and the bounding box. This in turn results in a large number of tiles being *erroneously* assessed as *intersected* (i.e., false positive) despite no actual intersection with the Gaussian.

Table 2. Ratio of unused Gaussians in tiles due to early termination.

Ratio	Train	Truck	Playroom	Drjohnson
Avg.	54.59%	29.24%	25.19%	18.50%
Max.	96.57%	96.20%	98.24%	90.39%

Consequently, this approach significantly increases the number of Gaussians to process per tile for the subsequent steps of the rendering pipeline,[5] adversely affecting both sorting and rasterization time due to unnecessary computation. Considering that these two steps are the key performance bottleneck in the rendering pipeline, avoiding the assignment of unused Gaussians to tiles can substantially reduce the amount of work for each tile and thus rendering time. Section 4.1 discusses the performance impact of the intersection test in Gaussian Splatting and presents our idea of opportunistically performing a more strict intersection test.

Observation II: Unnecessary Sorting for Unused Gaussians. Table 2 shows the percentage of *unused Gaussians* resulting from *early termination* after their assignment to tiles. As discussed in Section 3.1, the transmittance (T) is computed during volume rendering, and if the value falls below a threshold, we do not need to further accumulate the colors of the Gaussians after the current one; this resembles early ray termination in NeRFs.

We observe that a large number of Gaussians assigned to each tile are *not* actually processed due to *early termination* in the rasterization step. This implies that there is no need to spend time *precisely* sorting *all the Gaussians*, as some will not be ultimately used in the subsequent rasterization. Given that Gaussian sorting consumes a non-negligible amount of time in the rendering pipeline, which would become even greater when accelerating the rasterization step, avoiding unnecessary sorting could help reduce rendering time. Section 4.2 discusses our idea of exploiting this opportunity, which not only avoids unnecessary sorting but allows for *further* hiding the sorting overhead by breaking the serialization of Gaussian sorting and rasterization and overlapping their execution.

Observation III: Ineffective Computation in Rasterization. Figure 5 shows the percentage of Gaussians that *actually* contribute to pixel colors during the rasterization process. In the tile-based execution, a thread block is assigned to

[5]The Gaussians assigned to tiles due to false positives will not ultimately be rasterized during alpha computation.

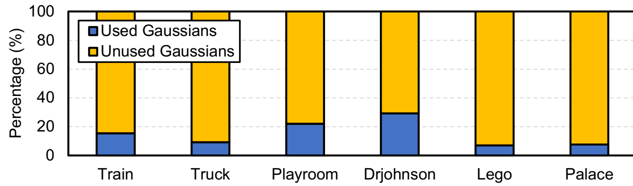

Figure 5. Percentage of Gaussians that actually contribute to the pixel colors among those assigned to tiles.

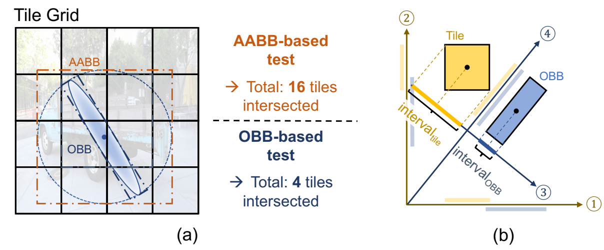

Figure 6. (a) Comparison of Gaussian-tile intersection tests using AABB and OBB. (b) OBB intersection test.

each tile, and all threads in the thread block work on the *same* Gaussian while iterating through the Gaussians assigned to the tile one by one. Although this greatly helps reduce rendering time by avoiding the overhead of *per-pixel* Gaussian sorting as well as thread divergence, as mentioned in [25], it unfortunately leads to the majority of threads performing *ineffective* computation to compute the α value and checking whether the Gaussian under process contributes to its own pixel color (although it turns out it does not).

As shown in the figure, the number of Gaussians that contribute to the pixels is significantly low, which implies that many threads compute something ineffective, thereby wasting compute cores. Ideally, we would like to manage per-pixel (per-thread) information regarding Gaussians, but this is not likely practical due to the overhead of maintaining per-pixel data structures and the lockstep nature of GPU execution. Instead, our key idea is to divide each tile into several *subtiles* and maintain *lightweight* per-subtile information about the pixel influence of the Gaussian under process within a tile. Section 4.3 discusses this *middle ground* approach, which helps reduce the amount of ineffective computation in our volume rendering core.

4 Optimizing Gaussian Splatting Pipeline

By leveraging the observations and opportunities from our analysis, we present the techniques to streamline the Gaussian rendering pipeline, which we employ in our architecture.

4.1 Gaussian Shape-Aware Intersection Test

In the preprocessing step, we need to identify the tiles that each Gaussian intersects for rasterization. As discussed in Section 3.3, however, using a simple intersection test results in a high number of false positives. This not only increases the Gaussian sorting overhead but prolongs the rasterization time due to *unnecessary* alpha computation of Gaussians that do not actually influence the pixels.

To reduce the false positives of the intersection test, we consider employing a more fitted bounding box that aligns well with the *anisotropic* nature of Gaussians, which is an *oriented bounding box* (OBB) [19]. Figure 6 compares the original AABB-based intersection test with the OBB-based one. For simplicity in computation, the state-of-the-art implementation sets an AABB using a circle with a radius equivalent to the length of the *semi-major* axis of each Gaussian. However, this method could *falsely* mark a large number of tiles as *intersected* although they are not, as depicted in Figure 6(a); for instance, 75% of tiles are incorrectly assessed as *intersected* in the figure. In contrast, the OBB, with the flexibility of its edges not constrained to be parallel to the coordinate axes, sets a tighter bounding box, especially when Gaussians become more anisotropic.

Assigning Gaussians to tiles using AABBs is computationally inexpensive, as we can directly identify *all* intersecting tiles by examining the top-left and bottom-right vertices of the AABB rectangle. In contrast, while a more fitted bounding box, such as OBBs, effectively reduces false positives, it is more costly due to the need of *individually* checking each tile with the bounding box for intersection. For example, Figure 6(b) illustrates the procedure of an OBB-based Gaussian-tile intersection test using the Separating Axis Theorem (SAT) [19]. SAT determines intersection by examining overlaps after projecting a tile and an OBB onto potential separating axes. In the figure, for each of the four axes (i.e., ①–④), we check if two projections (intervals) overlap. If an overlap occurs on *every* axis, we can conclude that the Gaussian intersects the tile.

Our key insight is that not all Gaussians need to be checked with a more strict intersection test; some can effectively use a simple bounding box. Moreover, an AABB intersection test is needed beforehand, even when employing an OBB-based test to identify candidate tiles. Hence, we *opportunistically* apply the OBB intersection test based on the characteristics of each Gaussian shape. We use an OBB only when 1) there are multiple intersecting tiles with an AABB, and 2) the ratio of the major axis to the minor axis exceeds a pre-defined threshold. We set the threshold at 2, which provides a reasonable balance between the overhead of the intersection test and false positives of Gaussian-tile assignments in our experiments. Note that we can easily obtain the necessary information during preprocessing.

4.2 Hierarchical Sorting

Based on the observation that we do not need to precisely sort all the Gaussians when considering early termination,

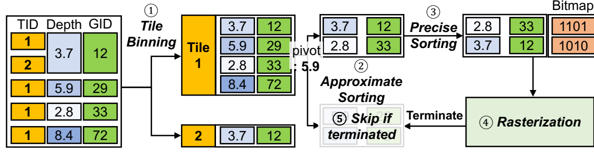

Figure 7. A walking example of hierarchical sorting.

we propose a two-stage sorting process, called *hierarchical sorting*, which allows us to reduce the overhead associated with Gaussian sorting. Our key idea is to initially perform *approximate* sorting of all Gaussians, ensuring that the Gaussians in Group$_i$ have lower depth values than those in Group$_{i+1}$, while the Gaussians within each group are *not* strictly ordered. Subsequently, we *precisely* sort each Gaussian group at the time it is needed for rasterization runtime. In essence, the approximate-sorting stage serves as a preprocessing step aimed at reducing the sorting workload in the subsequent stage.

This provides us with two key benefits. First, we can avoid the sorting overhead for the Gaussian groups that will not be ultimately rasterized due to early termination. In addition, it allows us to hide the precise sorting overhead by overlapping it with the rasterization process. Figure 7 shows an example of hierarchical sorting. After the tile intersection test, we first group the Gaussians that intersect the same tile. In the figure, four Gaussians (Gaussian IDs: 12, 29, 33, 72) are grouped as a set by tile binning, and we perform approximate sorting on the group. To start the sorting, we randomly select the depth of a Gaussian in the set as a pivot. Consequently, it is divided into two *chunks*: one for lower and the other for higher depths. Once the precise sorting of the lower chunk is finished, we can start rasterization. If the rasterization of the tile is finished due to early termination, we can skip the precise sorting for the remaining higher chunks.

4.3 Skipping Ineffective Computation

In rasterization, the alpha values (α) of Gaussians are crucial for determining their influence on a pixel's final color. Originally, if the α falls below a certain threshold, the Gaussian is excluded from rendering for the pixel, which we call *α-pruning*. As discussed in Section 3.3, a significant number of Gaussians are *pruned* after α-computation in the GPU execution, not contributing to the pixel color through volume rendering. Given that α-computation involves multiple floating-point operations (Equation 2), including a costly exponential operation, the compute core consumes a large portion of execution time for the ineffective computation.

Our key insight is that we can actually know the exact boundary of each projected Gaussian because we treat it as an ellipse; although a Gaussian is defined throughout all the points in the 2D/3D space, we only keep it with a 3σ (99.7%) confidence interval. This means that we can pinpoint the *exact* pixel locations to which a Gaussian may contribute.

Ideally, we want to maintain per-pixel information for each Gaussian, but it is prohibitively costly. Instead, to reduce the ineffective α-computation, we divide each tile into smaller sections, which we call *subtiles*. For each Gaussian, we then encode the intersecting information between the Gaussian and subtiles as a *bitmap* during preprocessing. For example, we split a tile into N subtiles, and an N-bit bitmap is assigned to each Gaussian, in which the i-th bit indicates whether the Gaussian intersects with the i-th subtile of the current processing tile (e.g., 0: not intersecting, 1: intersecting). Along with our proposed GSCore architecture (Section 5), this can greatly reduce unnecessary α-computation while skipping the i-th subtile when the bit is zero.

The bitmap information is generated by performing an intersection test between a Gaussian and *subtiles* within a tile. Note that we use the features already computed in the preprocessing step and simply calculate the radius (r) of the major and minor axes for bounding boxes using Equation 3:

$$r_{major} = 3 \times \sqrt{\max(\lambda_1, \lambda_2)} \qquad r_{minor} = 3 \times \sqrt{\min(\lambda_1, \lambda_2)}, \quad (3)$$

where λ_1 and λ_2 are the eigenvalues of a projected 2D covariance matrix of the Gaussian.

4.4 Putting It All Together

Figure 8 shows the Gaussian-based rendering process optimized with our proposed mechanisms: shape-aware intersection test, hierarchical sorting, and subtile skipping. When the

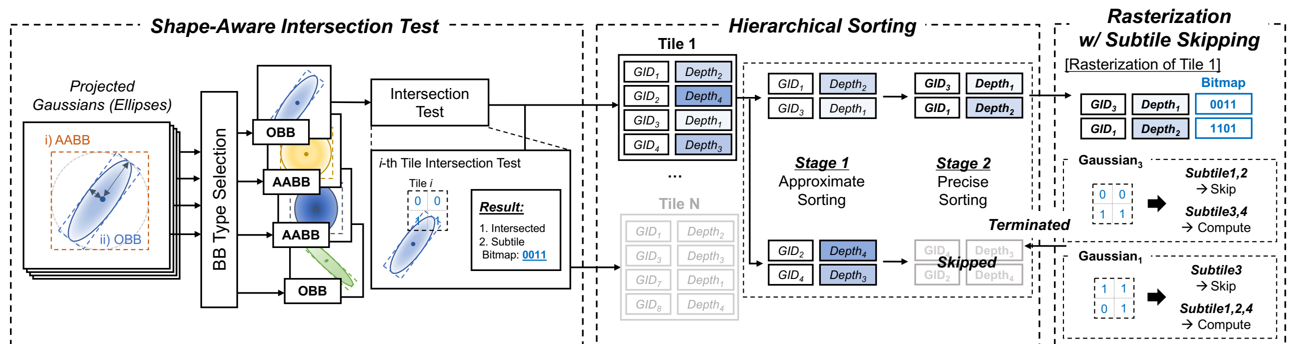

Figure 8. Execution flow of GSCore.

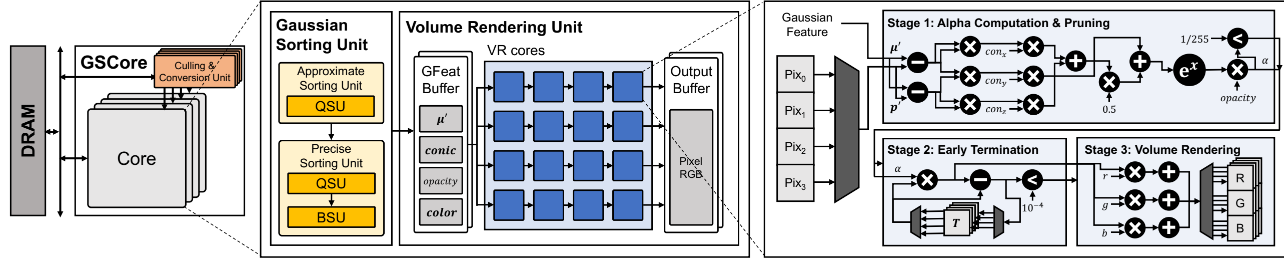

Figure 9. Overall design of GSCore architecture.

projected Gaussians are given, we *opportunistically* choose the type of the bounding box for the intersection test while considering the extent of the Gaussian and the aspect ratio of the major and minor axes. We then identify the intersecting tiles by employing different intersection tests for the AABB and OBB. If the tile is intersected, we also generate a subtile bitmap, which will be used in the rasterization step. After the intersection test is done for all Gaussians, they are fed into the corresponding set of the tile.

To reduce the sorting overhead, we approximately cluster the Gaussians as multiple chunks in the first stage, and we precisely sort each chunk when it is needed. In Figure 8, there are four Gaussians assigned to Tile 1, which are divided into two chunks in Stage 1. From the first chunk, we do precise sorting and start the rasterization. During rasterization, we effectively skip the computation of the subtiles when the corresponding bitmap is zero.

5 GSCore Architecture

In this section, we present GSCore, a specialized rendering engine for 3D Gaussian Splatting.

5.1 Overall Design

Figure 9 shows the overall design of GSCore, which comprises three main hardware modules: Culling and Conversion Unit (CCU), Gaussian Sorting Unit (GSU), and Volume Rendering Unit (VRU), each of which is responsible for preprocessing, Gaussian sorting, and rasterization, respectively.

Given a set of 3D Gaussians with a viewpoint, the CCU first culls invisible Gaussians. It then converts the features of the visible Gaussians into new features (e.g., depth, RGB color, projected 2D mean and covariance matrix) for use in the subsequent modules. Using the new attributes, the AABB/OBB intersection test units in the CCU conduct shape-aware intersection tests to identify the tiles that intersect with the projected Gaussian.

With the depth values and indices of the Gaussians, the GSU performs depth-based sorting for each tile after tile binning. It organizes the Gaussians within the same tile in a hierarchical manner with two stages; i.e., approximate sorting and precise sorting. In the first stage, the Gaussians are divided into several *chunks*, each of which is of a size that can be stored in the Gaussian feature (GFeat) buffer of the VRU. Subsequently, they are precisely sorted starting from the first chunk in the second stage and then sent to the Gaussian feature buffer.

Finally, the VRU performs rasterization with subtile skipping. It processes a single subtile at a time, which is mapped to 4×4 volume rendering cores (VR cores). The VR cores are arranged in a tiled fashion, and each VR core is responsible for computing the colors of 2×2 pixels. In the following sections, we delve into the details of each hardware component.

5.2 Culling and Conversion Unit

The Culling and Conversion Unit (CCU) mainly performs frustum culling, spherical harmonics computation, and shape-aware intersection test, where one of the key components is the OBB intersection test unit (OIU). The OIU is activated to perform a more strict intersection test when an AABB intersection unit finds multiple tiles that a Gaussian intersects while meeting the threshold condition, which we discuss in Section 4.1. We implement the OIU to perform SAT by using an efficient OBB intersection test algorithm [19]. The process involves projecting the centers of two boxes (i.e., a tile and an OBB of the Gaussian) onto an axis and calculating the radii, which are half the length of the intervals (Figure 6). Then, if the distance between the projected centers exceeds the sum of the radii, it indicates that the intervals do not overlap. In principle, we need three dot product operations for each axis: one to project the vector between the centers, and the other two to compute the radii. Along with the algorithm, however, some dot product operations can be eliminated because all vectors are already expressed in the xy coordinate system. By reducing the number of required dot product operations, we can reduce the area overhead and latency for the OBB intersection test. In our design, each CCU also employs two OIUs to perform the test in parallel.

5.3 Gaussian Sorting Unit

Figure 10 compares the execution of a single tile between the GPU with the original algorithm and GSCore. Instead of sorting the entire Gaussians before rasterization, the Gaussian Sorting Unit (GSU) hierarchically sorts Gaussians in two stages. This improves performance by overlapping sorting with rasterization and by reducing the sorting overhead of the chunks after early termination.

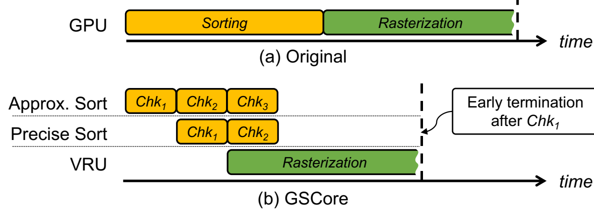

Figure 10. Execution diagram of (a) GPU and (b) GSCore with Gaussian sorting unit.

In the approximate sorting stage, we first divide the Gaussians into several chunks until a chunk fits into the Gaussian feature buffer in the VRU. The chunks are depth-ordered in-between, which means that the Gaussians in Chunk$_i$ (Chk_i) have smaller depth values than those in Chunk$_{i+1}$ (Chk_{i+1}), while the Gaussians within each chunk are not strictly ordered yet. Then, at the precise sorting stage, GSU prioritizes precise sorting of the Gaussians in the chunk with a smaller depth. This is because we need to do α-blending from the Gaussian at the smallest depth. Once the precise sorting of a single chunk is done, VRU begins rasterization for the chunk.

Figure 11 shows two main units of the GSU: a quick sorting unit (QSU) and a bitonic sorting unit (BSU). When the depth values (Depth) and Gaussian indices (GID) are given as key-value pairs, we perform quick sorting and bitonic sorting in the GSU. For this, the GSU employs two 7-pivot QSUs, one per each sorting stage, and one 16-channel BSU for the precise sorting stage. For simplicity, a 4-channel BSU is shown in the figure.

The QSU divides an unsorted list into multiple subsets. Each subset is organized such that all elements are sorted according to the pivot values, which serve as boundaries. To do so, the key is fed into comparators, and a popcount unit counts 1s of the output to find the subset index. We then store the key and value at the corresponding offset of the subset and increment the offset. In the first stage, we perform multiple rounds of quick sorting until the size of each subset is small enough to fit into the Gaussian feature buffer.

In the precise sorting stage, we use the BSU, which typically performs faster than quick sort because of its parallel execution. However, the BSU is generally not scalable as the number of input channels K increases [23]. This is because the number of comparators and interconnects grows proportionally to $K \log^2(K)$, resulting in a significant area overhead. To balance performance and area overhead, we opt for a 16-channel BSU. Then, in the precise sorting stage, the BSU is utilized when the subset size reduces to 16 or fewer after several rounds of quick sorting with the QSU.

5.4 Volume Rendering Unit

The Volume Rendering Unit (VRU) consists of 4×4 VR cores for computing colors through volume rendering (Equation 1). The VRU basically performs rendering at a *subtile* granularity

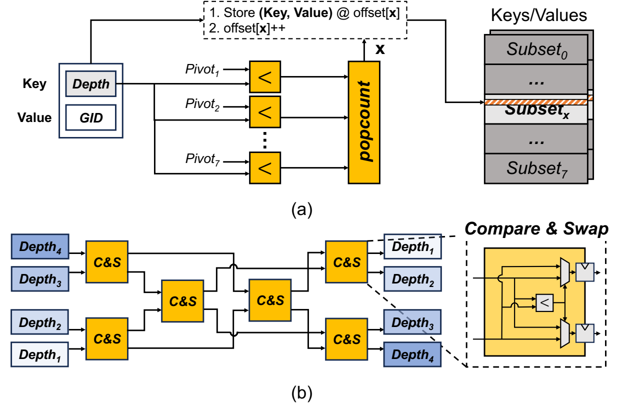

Figure 11. (a) Quick sorting unit. (b) Bitonic sorting unit.

(i.e., 8×8 pixels), and each core in the VRU is responsible for rendering 2×2 pixels. As every pixel in a subtile computes using the same Gaussian feature, the data is broadcast to all cores from left to right. At each cycle, the Gaussian feature is fed into the leftmost cores (i.e., the first column), and each core sends the feature to the right neighbor until it arrives at the rightmost cores. By using this interconnect topology, we can effectively reduce the interconnect complexity for broadcasting.

After loading the Gaussian feature, each core performs three stages: 1) alpha computation and pruning, 2) early termination, and 3) volume rendering, as shown in Figure 9. Each of these stages takes more than a single cycle. In the alpha computation stage, each core computes the alpha value (α) of the Gaussian based on Equation 2. It then checks if the value is smaller than a pre-defined threshold ($\frac{1}{255}$). If so, each core skips the α-blending of the Gaussian for the pixel. In the next stage, each core updates the transmittance (T) of the pixel, which is used to check for early termination, using Equation 4:

$$\mathrm{T_{i+1}} = \mathrm{T_i} \times (1 - \alpha_i) = \mathrm{T_i} - \mathrm{T_i} \times \alpha_i. \tag{4}$$

Each core stops volume rendering for the pixel if T_{i+1} becomes smaller than the threshold (10^{-4}); thus, the rest of the Gaussians assigned to the pixel are not used. For this step, there are two local registers for the transmittance: one for termination checking (T_{i+1}) and the other for the use in the last stage (T_i). In the last volume rendering stage, each core finally computes the attributed color of the Gaussian using the alpha (α_i), transmittance (T_i), and intrinsic color (c_i) and accumulates it to the pixel's color register (C).

As our volume rendering unit is fully pipelined, we need to consider the data dependency of the transmittance because it is sequentially updated, as shown in Equation 4. Within the VR core, transmittance is computed using the multiply-and-accumulate (MAC) unit. In our design, which operates at a 1GHz clock frequency, it takes four cycles for the MAC unit to compute transmittance T_{i+1} from T_i. Hence,

Table 3. Evaluated workloads.

Dataset	Scene (Resolution)	Type
Tanks&Temples [27]	Train (980×545)	Real World & Outdoor
	Truck (979×546)	
Deep Blending [22]	Playroom (1264×832)	Real World & Indoor
	Drjohnson (1332×876)	
Synthetic-NeRF [33]	Lego (800×800)	Synthetic
Synthetic-NSVF [30]	Palace (800×800)	

a straightforward core design would lead to the pipeline stall until the computation of T_{i+1} is finished, leading to severe core underutilization. To address the issue, the VRU employs a *pixel-rotating* pipelined core, where each VR core computes 2×2 pixels by rotating the pixels every cycle. Since there are no dependencies between the transmittances of different pixels, the VR core can proceed to work on other pixels without waiting for the completion of the previous transmittance computation.

6 Methodology

Hardware Implementation. We implement the RTL design of hardware components in GSCore and synthesize it using Synopsys Design Compiler with a commercial 28nm technology node. The SRAMs are generated using a commercial memory compiler with the same 28nm technology. Our design runs at a 1GHz clock frequency and is fully pipelined. We also implement a cycle-level simulator to evaluate the performance of our design with off-chip memory. The simulator models the GSCore architecture with detailed DRAM timing modeled after Micron LPDDR4-3200 with Ramulator [26]. It performs both functional and timing simulation with two primary inputs: a viewpoint and a set of 3D Gaussians; thus, our simulator also produces a final rendered image. We configure the timing parameters of the simulator based on the results from RTL synthesis. Performance metrics are calculated from the cycle count reported by the simulator, which also provides other statistics such as the number of SRAM accesses for estimating the energy consumption of on-chip buffers. The energy consumption of off-chip memory is obtained using DRAMPower [6] and the statistics from the detailed DRAM model.

Workloads. Table 3 presents the workloads that we use to evaluate our rendering engine. To generalize the results, we carefully select several real-world (both indoor and outdoor) and synthetic scenes with varying image resolutions from the datasets published in prior works [22, 27, 30, 33]. Then, we train the Gaussian model for each scene for 30K iterations. Additionally, we observe that using FP16 does not lead to perceptual differences compared to using FP32, except for the exponential function in the alpha computation. Based on the observation, we convert the models trained in FP32 to FP16 and perform rendering based on FP16 to improve compute throughput and area efficiency in our design.

Table 4. Comparison of the baseline GPU and GSCore.

Device	Technology	Area	SRAM	DRAM Bandwidth	Number of Cores
Xavier NX	12 nm	350 mm^2	11.15 MB	LPDDR4X 59.7 GB/s	384 CUDA Cores
GSCore	28 nm	3.95 mm^2	272 KB	LPDDR4 51.2 GB/s	64 VR Cores

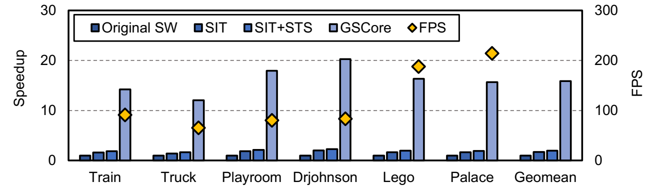

Figure 12. Speedup of GSCore over the baseline GPU.

Baseline. We compare our accelerator with the NVIDIA Jetson Xavier NX [39], which is a representative mobile computing device. The performance and power consumption of the GPU are measured using the built-in hardware counters. Table 4 shows the hardware comparison between Xavier NX and GSCore. GSCore has a smaller SRAM capacity and fewer compute units than the baseline GPU, resulting in a substantially smaller area despite using a technology node that is a couple of generations behind. For software, we use and modify author-released code, which uses the NVIDIA CUB library for Gaussian sorting (fast radix sorting with parallel prefix scans) and implements the CUDA kernel-based rasterizer. We enhance the baseline implementation to avoid computations that do not need to be performed at runtime for rendering (e.g., batched copy for merging SH coefficients) and also implement our optimizations on the codebase to evaluate their performance on the GPU.

7 Evaluation

7.1 Performance

Figure 12 shows the speedup of GSCore over the baseline GPU, along with the performance of software implementations of our optimizations on Xavier NX. In the baseline GPU, applying the shape-aware intersection test (SIT) provides a 1.71× speedup over the original implementation. The benefit comes from mitigating the false positives of the intersection test, which reduces both Gaussian sorting and rasterization time due to the decreased number of Gaussians per tile. Applying subtile skipping (STS) on top of that offers an additional 15% performance improvement. To mitigate thread divergence, we modify the thread indexing such that pixels within the same subtile are mapped to the same warp. The bitmap sparsity (i.e., the ratio of zeros) ranges from

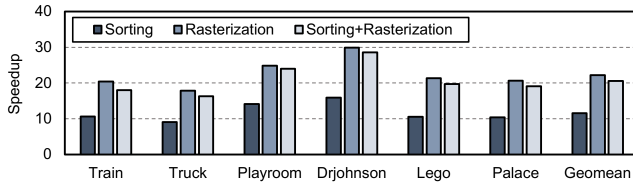

Figure 13. Speedup of sorting and rasterization over GPU.

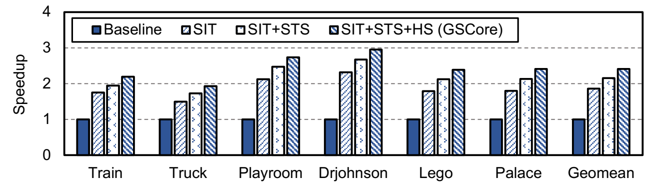

Figure 14. Speedup of variants isolating each optimization.

Table 5. Comparison of rendering quality (PSNR↑/LPIPS↓). Peak signal-to-noise ratio (PSNR) and learned perceptual image patch similarity (LPIPS) are widely used metrics to assess image quality.

Method	Train	Truck	Playroom	Drjohnson	Lego	Palace
Original	24.17	26.46	29.89	35.19	34.47	33.75
	0.17	0.22	0.22	0.18	0.02	0.03
GSCore	24.26	26.49	29.83	35.00	34.40	33.76
	0.17	0.22	0.22	0.18	0.02	0.03

around 35%–52% across the evaluated scenes, which leads to speedups by avoiding unnecessary alpha computation.[6]

In comparison, GSCore achieves an overall 15.86× speedup on average over the baseline GPU with the original implementation. As shown in Figure 13, which further presents the speedup of two main operations (sorting and rasterization) over each of which on the baseline GPU, the performance benefit mainly comes from our specialized hardware design tailored to each operation. The algorithmic optimizations that synergistically work with GSCore also contribute to the increase in performance, which we discuss in Section 7.3. Overall, GSCore allows for real-time rendering across the evaluated scenes, and for the synthetic scenes such as Lego and Palace, it offers a significantly higher FPS.

7.2 Rendering Quality and Compute Efficiency

Table 5 compares the rendering quality between the original implementation and the one employed in GSCore. While we employ several optimizations, they do not lead to any approximation; our intersection test does not produce false negatives, and hierarchical sorting preserves the sorting order in the precise sorting stage. The slight difference between the two is because of using FP16 computation in our design, which leads to a negligible PSNR drop (e.g., a 0.5% PSNR drop at most and even better PSNRs for some scenes) and no loss for LPIPS.

Without sacrificing rendering quality, our optimizations help improve compute efficiency compared to when employing the original algorithm on GSCore, as shown in Table 6. A higher ratio indicates that Gaussians are likely to be used more in the volume rendering stage after alpha computation; in contrast, a lower ratio indicates that hardware performs

Table 6. VRU core utilization when employing the original algorithm and our optimizations.

Method	Train	Truck	Playroom	Drjohnson	Lego	Palace
Original	5.31	3.91	5.76	5.86	2.52	2.73
GSCore	19.96	12.75	24.61	26.31	10.65	11.72

more unnecessary computation. With the shape-aware intersection test and subtile skipping, GSCore achieves higher compute efficiency compared to using the original algorithm.

7.3 Source of Performance Gain

Figure 14 shows the speedup when we apply the algorithmic optimizations—shape-aware intersection test (SIT), subtile skipping (STS), and hierarchical sorting (HS)—to GSCore. We evaluate four variants of GSCore: Baseline, SIT, SIT+STS, and SIT+STS+HS. The baseline is the execution model of the original algorithm in GSCore. The speedup of SIT mainly comes from reducing false positives of Gaussian-tile assignments. This decreases the number of Gaussians that need to be processed in both sorting and rasterization. As a result, it effectively mitigates unnecessary computation and memory traffic, as we discuss in Section 8. STS further improves the performance of rasterization by reducing ineffective computation within a tile using a bitmap. HS reduces Gaussian sorting overheads by overlapping the execution with rasterization and skipping the sorting of Gaussians after early termination. We see that HS synergistically works with STS; when employing STS, the relative portion of sorting in overall rendering time increases, thus reducing the overhead of sorting leads to a non-negligible speedup.

7.4 Area and Energy Efficiency

Table 7 presents the area and power of GSCore. At a 1GHz clock frequency, our design has a total area of 3.95mm^2 with

Table 7. Area and power.

Component	Configuration	Area [mm^2]	Power [W]
Culling and Conversion Unit	4 units	0.82	0.52
Bitonic Sorting Unit	4×(1 unit)	0.06	0.05
Quick Sorting Unit	4×(2 units)	0.01	0.01
Volume Rendering Core	4×(4×4)	1.81	0.25
GFeat Buffer+Others	4×2×(16KB+18KB)	1.25	0.04
Total		**3.95**	**0.87**

[6]For hierarchical sorting, our implementation based on concurrent kernel execution of sorting and rasterization leads to a slowdown due to the overhead of fine-grained synchronizations.

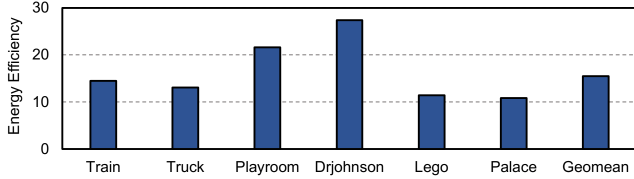

Figure 15. Energy efficiency.

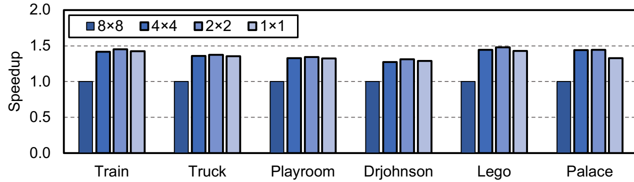

Figure 16. Performance comparison across VRU core sizes.

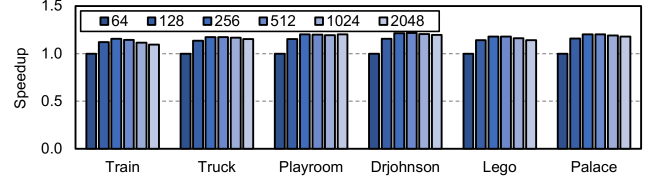

Figure 17. Performance comparison across Gaussian chunk sizes.

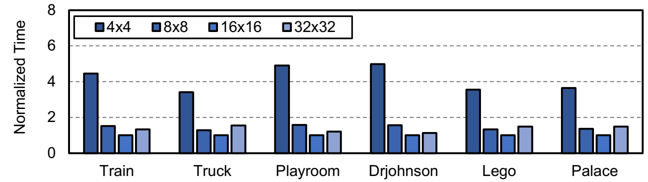

Figure 18. Normalized rendering time across tile sizes on GPU.

a power consumption of 0.87W. For area overhead, the Gaussian feature buffer is one of the key contributors. It is noted that although we opt for a size to accommodate 256 Gaussians (i.e., Gaussian chunk size of 256), we can also employ smaller chunk sizes with minimal impact on performance benefits of GSCore (Section 7.5), which would lead to a reduction in the overall area.

Figure 15 shows the energy efficiency of GSCore compared to the baseline GPU. The benefit comes from our specialized design tailored to the Gaussian-based rendering pipeline, which renders an image faster with low power consumption, along with the reduction in off-chip memory access due to our optimizations. Overall, GSCore achieves a 15.50× improvement in energy efficiency over the baseline.

7.5 Sensitivity Study

VRU Design Space. As the cores in the VRU compute a set of pixel colors by processing the same Gaussian features, using a smaller VRU can lead to a reduction in ineffective computation. Figure 16 shows the performance across different VRU configurations normalized to 8×8. For a fair comparison, we set the total number of VR cores as the same (i.e., 64) across the configurations by using multiple VRUs for small VRU sizes. The performance generally increases from 8×8 to 2×2 as we can skip more unnecessary computation in a smaller VRU. However, using multiple small VRUs leads to off-chip memory contention due to concurrent requests for Gaussian features from a large number of VRUs. Consequently, we observe a slowdown in the core size of 1×1 compared to the 2×2. Furthermore, since each VRU requires its own Gaussian feature buffer, using multiple small VRUs also increases the area overhead. We choose a core size of 4×4, which provides us with a reasonable tradeoff between performance and area overhead.

Gaussian Chunk Size. GSCore employs hierarchical sorting and splits Gaussians into several chunks in the approximate sorting stage. Because VRU starts computation only after the Gaussian features for a chunk are *fully* loaded into the feature buffer, the performance of GSCore varies depending on the chunk size. Figure 17 shows the speedup across the different Gaussian chunk sizes ranging from 64 to 2048, which is normalized to the chunk size of 64. The results show that moderate chunk sizes provide better performance than others, with a reduced speedup at smaller or larger chunk sizes. For larger chunk sizes (e.g., 1024 or 2048), the approximate sorting stage completes earlier because of fewer iterations resulting in a smaller number of chunks. However, they can lead to longer idle cycles of the VRU because it takes more time to load the features for a chunk into the buffer compared to the small and moderate chunk sizes, thereby exposing off-chip memory access latency despite employing double buffering for the feature buffer. Also, the benefit of hierarchical sorting can be reduced as larger chunks are likely to contain more Gaussians that become unnecessary after early termination. Conversely, for smaller chunk sizes (e.g., 64 or 128), the approximate sorting unit consumes more cycles because it needs to iterate more to generate smaller chunks. However, the idle cycles of the VRU can be lower because of the reduction in feature loading time and off-chip memory access. Considering this dueling tension, we choose the chunk size of 256 for our design.

8 Analysis and Discussion

Varying Tile Sizes on GPU. Using smaller tiles reduces the false positives of Gaussian-tile intersection, so rasterization latency can be reduced. However, simply reducing the tile size on GPUs leads to an increase in sorting overheads because Gaussians intersect with more tiles. Figure 18 shows the rendering time of different tile (thread block) sizes normalized to the default configuration (i.e., a 16×16-pixel tile) on Xavier NX. The results show that the overall rendering time increases as the tile size is reduced. For an 8×8 tile, we see that the increase in sorting overheads outweighs the decrease in rasterization latency. For a 4×4 tile, a thread block

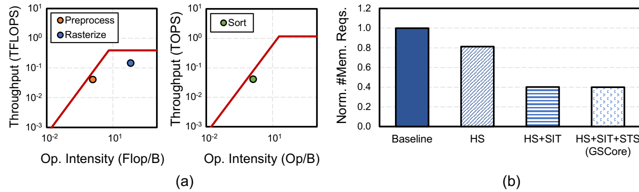

Figure 19. (a) Roofline models for the baseline GPU. (b) Normalized number of memory requests of variants isolating each optimization in GSCore (Train/30K).

only contains 16 threads (i.e., less than the warp size), and we observe that rasterization latency *also* becomes longer than that of 8×8 and 16×16, in addition to more sorting overheads. For a 32×32 tile, the increase in rasterization latency due to more false positives outweighs the decrease in sorting latency. As discussed in Section 4.3, our subtile skipping can help reduce the rasterization latency without increasing the sorting overhead.

Roofline and Memory Requests. Figure 19(a) shows the roofline models of the main operations in Gaussian Splatting on the baseline GPU (for Train/30K): preprocessing, rasterization, and Gaussian sorting. Preprocessing is memory bandwidth-bound as it loads a large number of parameters for each Gaussian with no data reuse. In contrast, rasterization is compute-bound because the feature dimension of loaded Gaussians is greatly reduced in the preprocessing stage and Gaussians are highly reused within a tile (i.e., CUDA thread block) with heavy computation for alpha and volume rendering. Gaussian sorting requires multiple reads and writes of both entire (> 4M) Gaussian indices and depth values of Gaussians, which are memory bandwidth-bound operations. Building on the analysis, we observe ineffective memory access and computation in Gaussian sorting and volume rendering and propose three algorithmic optimizations: hierarchical sorting (HS), Gaussian shape-aware intersection test (SIT), and subtile skipping (STS).

Figure 19(b) shows the number of memory requests for each optimization normalized to the original algorithm in our architecture (baseline). First, HS reduces 19% of memory requests while skipping unnecessary sorting of Gaussians after early termination. Second, SIT effectively removes false positives of Gaussian-tile assignments and reduces unnecessary Gaussian feature loads in both sorting and rasterization. This not only leads to a work reduction but results in an additional 2× reduction in off-chip memory access. Lastly, instead of reducing memory traffic, STS skips ineffective computation in rasterization, which is a compute-bound operation.

Fixed-Function Rasterizer in GPU. 3D Gaussian Splatting is a rasterization-based method, but it is a bit unique in that the primitives are transparent Gaussians (not meshes), so we need to perform volume rendering like NeRFs. While one may use fixed-function rasterizers at least for part of the rendering pipeline, it is likely a bit inefficient compared to the VRU in our work. In general, hardware rasterizers [1, 12] simply produce fragments for given triangular meshes, so alpha computation of Gaussians and blending updates need to be performed in other programmable or fixed-function hardware. All the transparent fragments are also blended without doing early termination, which is a widely used technique for volume rendering. One can probably perform early termination using programmable cores, but it is also expected to be slower compared to dedicated hardware such as VRU.

Using RT Cores for Intersection Test. Instead of producing fragments by conventional rasterization, one may consider using RT cores as they accelerate the ray-AABB intersection test. To do so, we need to build a Bounding Volume Hierarchy (BVH) with the AABBs of 3D Gaussians and perform a BVH traversal for every ray (i.e., pixel) to find intersecting Gaussians. However, achieving better performance compared to conventional rasterization might be challenging. In rasterization, we can *directly* find *all* intersecting pixels with the AABB using simple computation (Section 4.1), whereas for the case of RT cores, we need to perform a *redundant* BVH traversal for *every pixel* that intersects the same Gaussian. In addition, a BVH traversal is computationally more expensive than the simple intersection test for rasterization primitives, as it requires multiple fetching/decoding of the AABBs and ray-AABB intersection tests from the root to a leaf node. It is also noted that accelerating Gaussian-tile intersection tests does not significantly lead to a reduction in overall rendering time, as volume rendering and Gaussian sorting are bigger performance bottlenecks.

9 Related Work

Radiance Field Rendering Acceleration. Recent work proposed software optimizations [10, 15, 36, 48] and hardware accelerators [28, 29, 35, 47] to improve the performance of NeRFs. While the original NeRF [33] only consists of MLPs, which can be accelerated by conventional DNN accelerators [8, 9, 14, 21, 24, 34, 38, 44], the state-of-the-art NeRF models also employ other types of operations, such as multi-resolution hash encoding [36], which become performance bottlenecks. NeuRex [28] proposes restricted hashing, which can eliminate irregular off-chip memory access to hash tables and enable concurrent execution of encoding and MLP operations, along with an accelerator that features a specialized hash encoding engine. Instant-3D [29] presents a training accelerator with specialized hardware units that improve on-chip memory bandwidth utilization and reduce frequent writes to on-chip memory for parameter updates. However, these works focus on NeRFs that employ neural networks, whereas GSCore is the *first* work that targets Gaussian Splatting [25], which builds on differentiable rasterization primitives instead of using neural networks.

Ray Tracing Acceleration. Ray tracing produces high-quality images with realistic lighting effects by tracing the path of individual rays [18]. As it requires costly computation, however, a large number of software and hardware techniques have been proposed to accelerate ray tracing thus far [11, 31, 32, 37, 40, 41]. Several hardware vendors recently also started featuring ray-tracing hardware units in their GPUs, such as NVIDIA RTX series [5], ARM Immortalis-G715 [3], and Qualcomm Adreno GPU [42], which improves the performance of the ray-primitive intersection test. While they can accelerate ray tracing-based rendering, it is challenging to use the hardware units for efficiently performing Gaussian-based radiance field rendering, which GSCore offers with a specialized rendering engine.

Fixed-Function Graphics Pipeline. Traditionally, graphics rendering has been performed on the fixed-function pipeline in graphics hardware. Several prior works have studied the performance and power modeling of the graphics pipeline or implemented GPUs in RTL with fixed-function units such as hardware rasterizers or rendering output units (ROP) [20, 45, 50, 51]. While contemporary GPUs now employ programmable shaders that can be used for graphics rendering, modern rendering methods (e.g., ray tracing) can still benefit from fixed-function hardware such as RT Cores [5]. Gaussian Splatting, which this work targets, is another example of modern rendering methods that can be better supported by specialized hardware units such as GSCore.

10 Conclusion

3D Gaussian Splatting gains significant attention due to its remarkable ability to generate photorealistic images from novel viewpoints, much faster than neural renderings. Nevertheless, we observe that there are opportunities to make the technique more effective and bring its benefits across a wide range of computing segments. In this work, we carefully examine the rendering pipeline of 3D Gaussian Splatting and propose an algorithm-hardware co-design that enables us to efficiently execute the rendering pipeline. Through this co-design approach, GSCore markedly improves the performance of Gaussian-based rendering over existing mobile platforms while maintaining quality standards.

Acknowledgments

We thank the anonymous reviewers and our shepherd Gilbert Bernstein for their valuable feedback. This work was supported in part by a research grant from Samsung Advanced Institute of Technology and by the artificial intelligence semiconductor support program to nurture the best talents (No. RS-2023-00256081) supervised by the Institute for Information & Communications Technology Planning & Evaluation (IITP). The Institute of Engineering Research at Seoul National University provided research facilities for this work. Jaewoong Sim is the corresponding author.

References

[1] Nvidia ada gpu architecture. Technical report, NVIDIA, 2023.

[2] ADLINK. Nvidia jetson xavier nx-based ai vision system, 2022. https://www.adlinktech.com/en/news/nvidia-xavier-nxbased-poe-ai-vision-system.

[3] ARM. Immortalis-g715, 2022. https://www.arm.com/products/silicon-ip-multimedia/immortalis-gpu/immortalis-g715.

[4] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter 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] John Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 2020.

[6] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power and energy estimation tool. http://www.drampower.info.

[7] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision (ECCV)*, 2022.

[8] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[9] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[10] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea 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.

[11] Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. Treelet prefetching for ray tracing. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[12] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. Pomegranate: a fully scalable graphics architecture. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2000.

[13] Blender Foundation. Blender engine. https://www.blender.org.

[14] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[15] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[16] Epic Games. Unreal engine. https://www.unrealengine.com/en-US.

[17] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien P. C. Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.

[18] Andrew S. Glassner, editor. *An introduction to ray tracing*. Academic Press Ltd., 1989.

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

[20] Ayub A. Gubran and Tor M. Aamodt. Emerald: graphics modeling for soc systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[22] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (SIGGRAPH)*, 2018.

[23] M.F. Ionescu and K.E. Schauser. Optimizing parallel bitonic sort. In *Proceedings 11th International Parallel Processing Symposium (IPPS)*, 1997.

[24] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[25] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics (SIGGRAPH)*, 2023.

[26] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters (CAL)*, 2016.

[27] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (SIGGRAPH)*, 2017.

[28] Junseo Lee, Kwanseok Choi, Jungi Lee, Seokwon Lee, Joonho Whangbo, and Jaewoong Sim. Neurex: A case for neural rendering acceleration. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[29] Sixu Li, Chaojian Li, Wenbo Zhu, Boyang (Tony) Yu, Yang (Katie) Zhao, Cheng Wan, Haoran You, Huihong Shi, and Yingyan (Celine) 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.

[30] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. Neural sparse voxel fields. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[31] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. Intersection prediction for accelerated gpu ray tracing. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[32] Daniel Meister, Jakub Boksansky, Michael Guthe, and Jiri Bittner. On ray reordering techniques for faster gpu ray tracing. In *Symposium on Interactive 3D Graphics and Games (I3D)*, 2020.

[33] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020.

[34] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the intel harpv2 xeon+fpga platform: A deep learning case study. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[35] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. Hardware acceleration of neural graphics. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[36] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (SIGGRAPH)*, 2022.

[37] Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics (TOG)*, 2014.

[38] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[39] NVIDIA. NVIDIA Xavier System-on-Chip, HotChips 30, 2018.

[40] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1997.

[41] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (TOG)*, 21(3), 2002.

[42] Qualcomm. Adreno gpu, 2023. https://www.qualcomm.com/news/onq/2023/05/hardware-accelerated-ray-tracing-improves-lighting-effects-in-mobile-gaming.

[43] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2001.

[44] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[45] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 2008.

[46] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. *ACM Transactions on Graphics (SIGGRAPH)*, 2006.

[47] Xinkai Song, Yuanbo Wen, Xing Hu, Tianbo Liu, Haoxuan Zhou, Husheng Han, Tian Zhi, Zidong Du, Wei Li, Rui Zhang, Chen Zhang, Lin Gao, Qi Guo, and Tianshi 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.

[48] Cheng Sun, Min Sun, and Hwann-Tzong 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.

[49] Unity Technologies. Unity engine. https://unity.com/products/unity-engine.

[50] Blaise Tine, Varun Saxena, Santosh Srivatsan, Joshua R. Simpson, Fadi Alzammar, Liam Cooper, and Hyesoon 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.

[51] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. Vortex: Extending the risc-v isa for gpgpu and 3d-graphics. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.