This CVPR paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the accepted version;
the final published version of the proceedings is available on IEEE Xplore.

FlashGS: Efficient 3D Gaussian Splatting for Large-scale and High-resolution Rendering

Guofeng Feng[2,3,*,†] Siyan Chen[7,*,†] Rong Fu[1,✉] Zimu Liao[1] Yi Wang[1] Tao Liu[1] Boni Hu[1]
Lining Xu[4] Zhilin Pei[1] Hengjie Li[1] Xiuhong Li[6] Ninghui Sun[2,3] Xingcheng Zhang[1] Bo Dai[5]

[1]Shanghai Artificial Intelligence Laboratory, [2]Institute of Computing Technology, CAS, [3]University of Chinese Academy of Sciences
[4]The Chinese University of Hong Kong, [5]The University of Hong Kong, [6]Peking University, [7]Independent Researcher

* Equal contribution, † Work done during an internship at Shanghai Artificial Intelligence Laboratory, ✉ Corresponding author

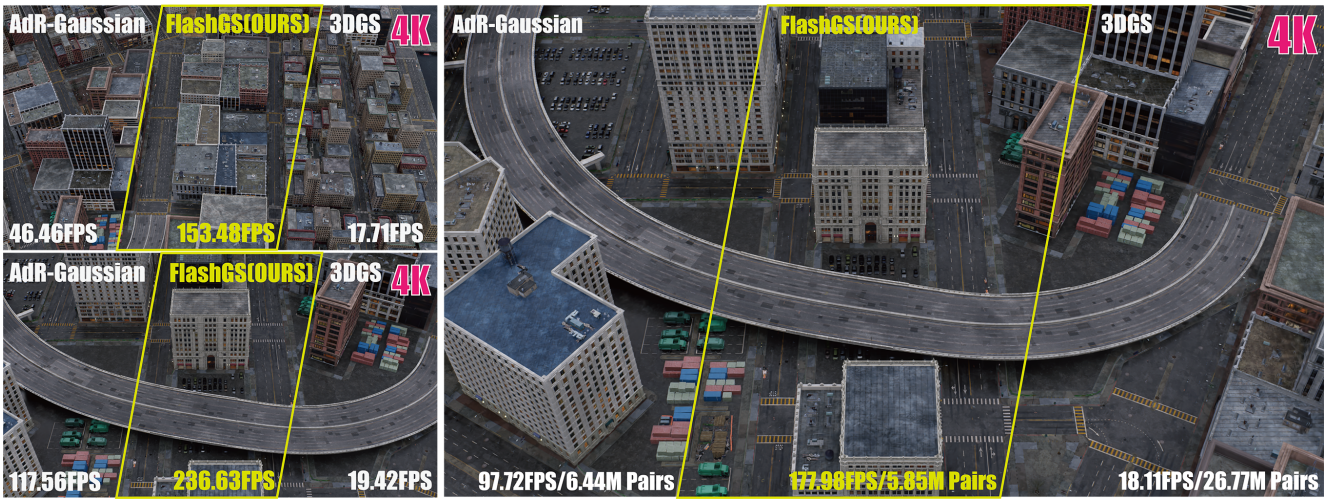


Figure 1. Our method significantly improves the rasterization performance of 3D Gaussian Splatting (3DGS), enabling real-time rendering in large-scale, high-resolution scenes, such as MatrixCity [32] at 4K. The key innovation lies in mitigating redundancy and workload imbalance inherent in tile-based rasterization. To achieve this, we introduce a precise intersection algorithm that eliminates redundant Gaussian-tile pairs and propose a Gaussian size-aware adaptive task partitioning strategy. Additionally, we design a pipelined rendering process, which further enhances performance, leading to substantial speedups.

Abstract

Recent advances in 3D Gaussian Splatting (3DGS) have demonstrated significant potential over traditional rendering techniques, attracting widespread attention from both industry and academia. However, real-time rendering with 3DGS remains a challenging problem, particularly in large-scale, high-resolution scenes due to the presence of numerous anisotropic Gaussian representations, and it has not been extensively explored. To address this challenge, we introduce FlashGS, an open-source CUDA library with Python bindings, featuring comprehensive algorithm design and optimizations, including redundancy elimination, adaptive scheduling, and efficient pipelining. First, we eliminate substantial redundant computations through precise Gaussian intersection tests, leveraging the intrinsic mechanism of the 3DGS rasterizer. During task partitioning, we propose an adaptive scheduling strategy that accounts for variations in Gaussian size and shape. Additionally, we design a multi-stage pipelining strategy for color computation in the rendering process, further accelerating performance. We conduct an extensive evaluation of FlashGS across a diverse range of synthetic and real-world 3D scenes, encompassing scene sizes of up to 2.7 km^2 cityscape and resolutions of up to over 4K. Our approach improves 3DGS rendering performance by an order of magnitude, achieving an average speedup of 7.2×, and rendering at a minimum of 125.9 FPS, setting a new state-of-the-art in real-time 3DGS rendering. https://github.com/InternLandMark/FlashGS.

1. Introduction

3D reconstruction and neural rendering, driven by recent breakthrough 3D Gaussian Splatting (3DGS)[21], have demonstrated significant utility acc applications such as autonomous driving [20, 51, 53], virtual reality (VR) gaming [12, 19], augmented reality (AR) [31], and virtual worlds [2]. While 3DGS has opened up a new paradigm, enabling fast rendering while maintaining image quality, it has not yet reached the level required to meet the growing demand for large-scale and high-resolution scenes in real-world applications, such as city-level reconstruction. 3DGS employs Gaussian representations for scenes and utilizes tile-based software rasterization [26] for rendering. This approach inherently involves significant computational overhead, as both the number of Gaussians and the number of Gaussian-tile interactions scale with scene complexity and resolution [29, 45]. As a result, rendering large-scale, high-resolution scenes requires managing a growing number of Gaussians and tiles, which poses a critical challenge for practical deployment.

To mitigate computational overhead while ensuring high rendering quality, we conduct an in-depth analysis of the rasterization process and identify key inefficiencies and redundancies: (1) **Redundant Gaussian-tile intersections:** 3DGS checks the overlap between ellipses (projected Gaussians) and tiles for rendering. However, the commonly used bounding box approximation fails to accurately capture Gaussian attributes such as pose and shape, leading to significant redundancy. (2) **Diversity in Gaussian sizes:** 3DGS reconstructs scenes at high quality with Gaussians of varying sizes. During intersection checks, this results in an imbalanced workload—some Gaussians intersect only a single tile, while others span multiple tiles. (3) **Uneven Gaussian distribution of scenes:** The number of Gaussians processed per tile varies significantly due to the inherent distribution of scene elements. This imbalance in workload can lead to performance bottlenecks and inefficient resource utilization.

To further enhance the performance of 3DGS, subsequent research has focused on reducing the computational overhead associated with the large number of Gaussian representations required for scene reconstruction. Several approaches apply compression or pruning methods to minimize the storage and generation of excessive Gaussians [10, 14, 29, 34]. These approaches may lead to performance degradation, and the effectiveness may depend on the characteristics of the original scene. Another category of methods focuses on minimizing computational redundancies during the tiled-base software rasterization, mainly arising from determining the contribution of each Gaussian to individual tiles. AdR-Gaussian [45] and GScore [27] have identified that the axis-aligned bounding box (AABB) used in vanilla 3DGS for determining Gaussian-tile overlap is not sufficiently tight. They introduced improved bounding box methods to reduce incorrect intersection tests. However, an even tighter bounding box used to approximate the Gaussian projected ellipse may still reserve substantial redundancy, at least a non-negligible 21.5% (i.e. 1-π/4) redundancy still exists, even in the ideal case where the bounding box tightly encloses the ellipse.

To address these performance issues, we propose FlashGS, an efficient 3DGS rasterization pipeline, which eliminates redundant tasks in the rendering process without modifying the underlying scene representation. FlashGS introduces a precise intersection algorithm that directly computes the relationship between ellipses and tiles, avoiding approximations and ensuring that all Gaussian-tile pairs involved in rendering are accurately identified without introducing invalid objects. Additionally, we design a dynamic task partitioning strategy that adapts to the size of Gaussians at runtime. Specifically, for small Gaussians covering a single tile, one thread is assigned, while for larger Gaussians covering multiple tiles, the workload is re-distributed across multiple threads to balance the computation. We aim to minimize task execution time while preserving the Gaussian distribution, ensuring high rendering quality even for heavy-load tasks. To further optimize performance, we introduce a pipelined rendering process to mitigate performance degradation caused by data dependencies. By overlapping memory access for Gaussian features and pixel color calculations during alpha blending in a multi-stage pipeline, we reduce the total execution time.

Our main contributions are as follows:

- We design a precise intersection algorithm that tightly adapts to Gaussian shapes, incorporating algebraic simplifications and geometric transformations to reduce additional overhead.
- We propose a dynamic parallel scheduling strategy that adaptively maps the workload based on the Gaussian size, enabling efficient task partitioning.
- We introduce a software pipelining tile-based renderer to mitigate the load imbalance caused by the uneven distribution of Gaussians, ensuring both computational efficiency and rendering accuracy.

Extensive evaluation demonstrates that FlashGS consistently achieves real-time rendering at a minimum of 125.9 FPS. Compared to 3DGS [21] and current state-of-the-art 3DGS renderer ADR-Gaussian [45], FlashGS achieves an average of 8.7× and 2.2× speedup on large-scale and high-resolution scenes.

2. Related Work

2.1. Novel View Synthesis

Novel view synthesis (NVS) represents a crucial yet challenging task in computer graphics, with Neural Radiance

Fields (NeRF)[36] and 3DGS [21] emerging as two prominent approaches for addressing NVS effectively.

Neural Radiance Field NeRF employs a multi-layer perceptron(MLP) to derive color and opacity based on spatial coordinates and camera direction, utilizing volume rendering technique [8] to synthesize images from novel views. However, the rendering process is hindered by frequent queries of the MLP and ray sampling, resulting in slow performance. Although recent works [13, 38, 46, 47] leverage discrete structures like voxel grids and point clouds to mitigate the computational cost of MLP inference, they still face limitations in sampling speed.

3D Gaussian Splatting 3DGS adopts 3D Gaussians and a differentiable tile-based rasterization technique to bypass the MLP and sampling used in NeRF, enabling real-time and photorealistic rendering of scenes. Recent advancements have focused on improving reconstruction and rendering quality by optimizing projection process [15, 18], eliminating popping artifacts [15, 40], anti-aliasing [50], regularizing with the geometry [30, 33, 49], incorporating ray-tracing [37], etc.

2.2. High-Performance 3DGS

Model Compression 3DGS comes with a substantial memory usage caused by millions of Gaussians with spherical harmonic (SH) coefficients, constraining the scale and the Level of Detail (LOD) of scenes. To address this, recent works have reduced the model size while maintaining rendering quality by reducing the number of Gaussians [11, 35] and storing them in compact structures [6, 7, 34, 39, 41].

Model Scaling-up To render large-scale scenes beyond the memory constraints, multi-GPU parallelism has been applied. Grendel-GS [52] introduces sparse communication and dynamic load balance mechanisms. DoGS [5] decomposes a scene into K blocks and applies the alternating direction method of multipliers (ADMM) [4] for distributed 3DGS training.

Fast Rendering Pipeline Real-time rendering is essential for handling large-scale, high-resolution scenes. Recent studies [9, 16, 42] have focused on improving rendering speed. For example, gsplat [48] is a CUDA differentiable rasterization library for 3DGS, designed for memory-efficient training and rendering. GScore [27] proposes a dedicated hardware acceleration unit, that efficiently supports the 3DGS rendering pipeline on mobile devices. However, hardware acceleration approaches often lack generality across 3DGS variants. AdR-Gaussian [45] approximates ellipses using tight AABB to reduce intersects of tiles and Gaussians, accelerating the rendering process. However, its modification of Gaussian distribution to balance computational load results in precision loss.

Current optimizations for 3DGS often target specific aspects, such as rendering speed [45], memory efficiency [28], or rendering quality improvements with secondary performance benefits [15, 34, 40]. These efforts frequently overlook the computational bottlenecks inherent in 3DGS. In contrast, our work provides a systematic analysis, design, and implementation of a high-performance 3DGS rendering pipeline, FlashGS, focusing on efficiently supporting its core computational operations. Moreover, our approach can be seamlessly integrated with existing 3DGS-related works, offering performance enhancements.

3. Method

Background 3D Gaussian Splatting (3DGS) [21] models the scene with a set of Gaussian ellipsoids, enabling fast rendering through rasterizing these Gaussians into high-quality images. The rasterizer can be coarsely divided into three steps: preprocessing (preprocessCUDA and DuplicateWithKeys), sorting, and rendering, as illustrated in Figure 2. In the preprocessing stage, each Gaussian is processed in parallel to obtain perspective-dependent data required for rendering. DuplicateWithKeys identifies the tiles covered by each Gaussian and generates intersected key-value pairs. In the sorting phase, the start and end indices of Gaussians within each tile are determined for rendering, using a key that combines view space depth and tile ID. In the rendering stage, pixel colors are computed in parallel, and Gaussians in each tile are rendered by alpha blending. We observe significant computational redundancy in the rasterization pipeline, with only a limited number of key-value pairs actually contributing to the rendering result. The preprocessing phase assigns unnecessary tiles that do not overlap the projected Gaussians, leading to redundant Gaussian-tile pairs and inefficiencies in sorting and rendering. Additionally, the varying size of Gaussian distributions and the latency in rendering slow tiles create an imbalance between computation and memory access, potentially underutilizing computing resources.

Overall Architecture Our primary objectives are to 1) eliminate Gaussian redundancy and 2) resolve or mitigate load imbalance without compromising rendering quality. To achieve this, we propose FlashGS, a novel algorithm that integrates redundancy elimination, adaptive scheduling, and efficient pipelining. These components are meticulously designed to enhance the performance of the rasterization process, as shown in Figure 2. Section 3.1 briefly outlines the process of determining the valid range of Gaussian-projected ellipses, which is similar to AdR-Gaussian [45]. Section 3.2 presents our precise intersection algorithm in detail, which tightly adapts to the shape of Gaussians to

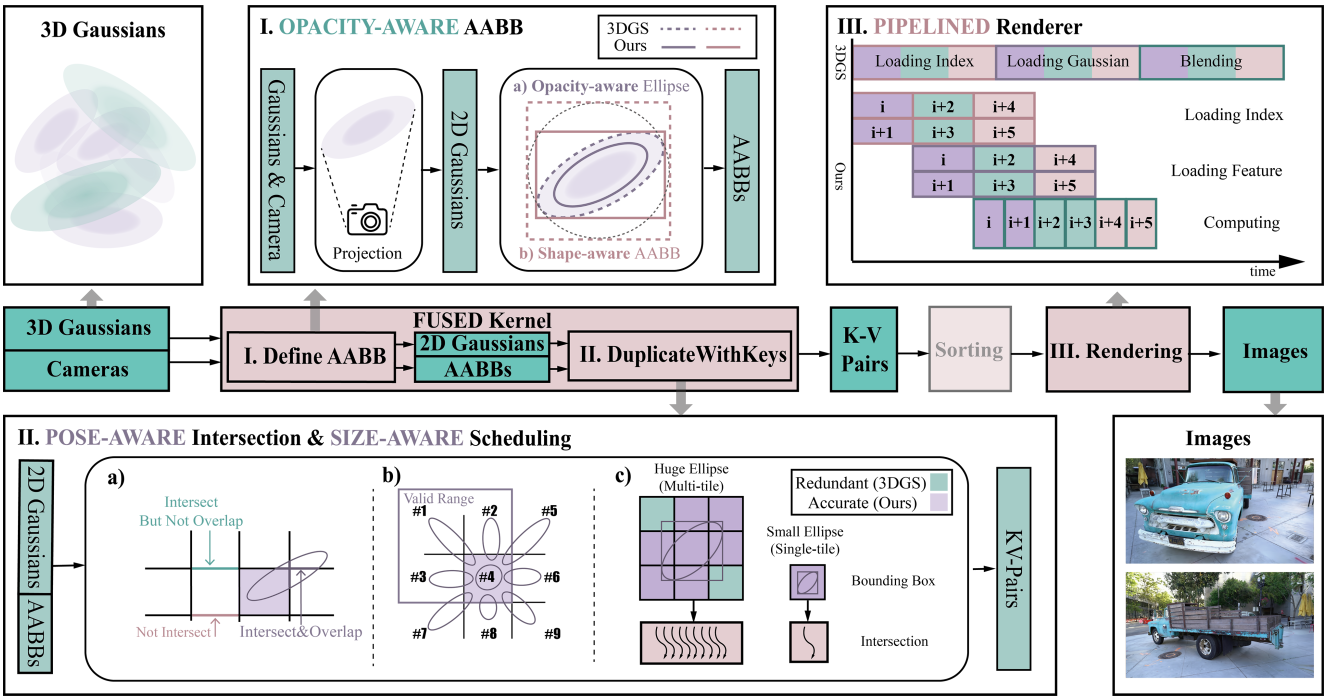


Figure 2. The detailed pipeline of FlashGS, incorporates precise intersection testing, balanced task partitioning, and a pipelined renderer.

effectively eliminate invalid intersections. Section 3.3 describes our dynamic parallel scheduling strategy, designed to address load imbalances caused by the varying sizes of Gaussians. Section 3.4 introduces a three-stage pipeline algorithm, to mitigate load imbalances resulting from the uneven Gaussian distribution.

3.1. Opacity-aware AABB Definition

We need to define the valid range of the ellipse projected by the Gaussians before performing the intersection test. While 3DGS applies the common 3-σ rule to set the range, it ignores the influence of Gaussian features such as opacity. We incorporate opacity α to obtain a more precise range. Without loss of generality, we assume that the center (mean) of the 2D Gaussian distribution is located at the origin, and the decline of the opacity from the center is given by $\alpha_r = \alpha \times \exp(-\frac{r^2}{2\sigma})$, where α is the opacity at the Gaussian center, r is the distance vector from the center to the current point, and σ is the covariance. When α decays to a predefined threshold τ, we consider this point to define the boundary of the effective ellipse: $r = \sqrt{2ln(\frac{\alpha}{\tau})\sigma}$. Given that RGB values are typically represented using 256 discrete levels, we set $\tau = \frac{1}{255}$. We derive that when $\alpha \leq 0.35$, our method outperforms the three-sigma rule, otherwise, we default to using the three-sigma rule.

Directly computing the intersection between a Gaussian projected ellipse and all tiles on the screen is expensive. To reduce this overhead, we first employ a compact AABB (defined as $r = \sqrt{2\ln(\frac{\alpha}{\tau})\sigma}$, where σ_x and σ_y correspond to projection along the X and Y axes). It serves to efficiently identify candidate tiles that require further intersection tests. Subsequently, we refine the Gaussian-tile intersection process using geometric and algebraic simplifications, significantly reducing computational costs.

3.2. Pose-aware Precise Gaussian-tile Intersection

We propose a precise intersection algorithm to better handle Gaussians with varying poses, as they are often not axis-aligned. Directly solving the system of equations for the projected ellipse's conic curve and the tile's edges would incur significant computational overhead. We reduce the solving complexity through carefully designed algebraic transformations and geometric simplifications.

Geometric Transformation For each tile intersection, we apply geometric transformations to avoid the cost of directly solving multiple quadratic equations. First, we determine whether the ellipse and the rectangle intersect by checking if the ellipse's center lies within the rectangle or if the ellipse intersects its edges. While the former is straightforward, the latter case is more complex. Our key insight is that determining whether the ellipse intersects with a specific edge of the rectangle can be transformed into another problem: checking whether the segment of the ellipse's intersection with the line of the rectangle's edge overlaps with the line segment representing the edge, as illustrated in Fig-

ure 3(a). If an intersection exists, we compute the endpoints of the segment on the line that is enclosed by the ellipse. These endpoints are then used to determine whether the segment overlaps with the corresponding edge of the rectangle. The intersection of the ellipse with any edge of the rectangle determines the overall intersection relationship between the ellipse and the rectangle. In the worst case, we may need to check all four edges of the tile to make a final determination (if none of the first three checked edges intersect with the ellipse). This could lead to significant overhead. To optimize this, we leverage the spatial relationship and symmetry between the ellipse's center and the rectangle. This ensures that, in the worst case, we only need to check two edges instead of all four, as illustrated in Figure 3(b). The four edges of the tile divide the space into nine regions, and the ellipse's center may lie in any of them. First, we check whether the ellipse's center lies within the rectangle. Then, we examine the relative position of the ellipse's center with respect to the rectangle's center. For example, in the case shown in the 'valid range', where the center lies in the 'top-left' region relative to the center, we only need to check the two corresponding edges in that region. Other positions, such as 'bottom-right' and 'top-right' can also be equivalently solved using central/axial-symmetry, respectively, following the 'top-left' solving process.

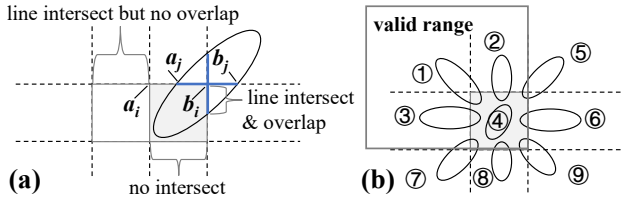


Figure 3. Geometric simplification for precise intersection. (a) A tile is considered intersected if the segment of the ellipse intersecting the line of the tile's edge ($[a_j, b_j]$) coincides with the edge ($[a_i, b_i]$). (b) The relative location between the ellipse's center and the tile can be categorized into nine types. Leveraging symmetry, we can reduce this to only four cases to determine which tile edges need intersection checks.

Algebraic Simplification We use algebraic equivalence transformations to avoid high-cost computations like square roots. Given two line segments $[a_i, b_i]$ and$[a_j, b_j]$ in geometric transformation, they overlap if their intersection is not empty as shown in Equation 1:

$$min(b_i, b_j) - max(a_i, a_j) \geq 0 \tag{1}$$

Specifically, this is equivalent to checking if the right endpoint (end) of one line segment is after the left endpoint (start) of the other line segment, and if the left endpoint (start) of the line segment is before the right endpoint (end) of the other line segment, i.e., $a_i \leq b_j \wedge b_i \geq a_j$. Computing the endpoints of intersected segments ($[a_j, b_j]$) requires solving quadratic equations formed by the ellipse and tile edge, as shown in Equation 2, where $\Delta = B^2 - 4AC$, and A, B, C are coefficients of the equation $AX^2 + BX + C = 0$, when $\Delta \geq 0$:

$$a_i \leq \frac{-B + \sqrt{\Delta}}{2A} \wedge \; b_i \geq \frac{-B - \sqrt{\Delta}}{2A} \tag{2}$$

The key insight is that we only need to check the intersection condition, not the exact root values. Assuming $A > 0$, this condition can be transformed into Equation 3 to avoid costly operations like division and square roots, where $e_1 = 2Aa_i + B$ and $e_2 = 2Ab_i + B$:

$$\Delta \geq 0 \wedge (e_1 \leq 0 \; \vee \; e_1^2 \leq \Delta) \wedge (e_2 \geq 0 \; \vee \; e_2^2 \leq \Delta) \tag{3}$$

3.3. Size-aware Adaptive Gaussian Scheduling

Imbalanced workloads arise from the varying number of tiles covered by Gaussians of different sizes, as shown in Figure 4. This can degrade the performance of both the intersection phase and the Gaussian-tile pair key-value binding stage.

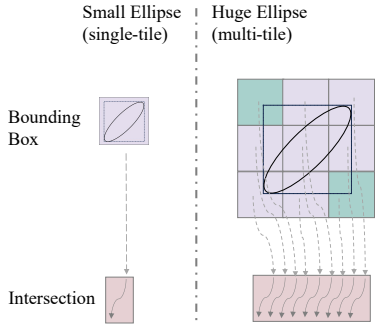


Figure 4. Adaptive intersection task partitioning for Gaussians of varying sizes. For large ellipses covering multiple tiles, the intersection test is distributed across multiple threads.

To optimize scheduling, we dynamically select processing methods based on Gaussian size and employ adaptive mapping for balanced workload distribution. For small ellipses intersecting a single tile, the same thread handles both intersection and key-value generation. For larger ellipses spanning multiple tiles, we assign a warp (32 threads) to leverage GPU hardware. This is dictated by the GPU's hardware scheduler, where a warp is the smallest scheduling unit and a thread is the smallest computational unit. Using a single warp for all cases would limit the efficiency of Gaussians intersecting only one tile to 1/32, while assigning multiple warps to large Gaussians would introduce unnecessary scheduling overhead.

Our final preprocessing workflow (Algorithm 1) maps tasks based on Gaussian size and shape. By merging intersection and key-value generation (preprocessCUDA and

Algorithm 1 Size-aware Scheduling Strategy

Input: $Gaussians$ and candidate tiles
Output: Key-value (intersected Gaussian-tile) pairs

if bounding box covers a single tile **then**
  if tile intersects ellipse or contains center **then**
    $key \leftarrow compute_tile_key$
    $offset \leftarrow atomicAdd(offset, 1)$
    $value \leftarrow gaussian_idx$
    Store key and value
for each thread in warp **do**
  if $multi_tiles$ **then**
    $parameters \leftarrow multi_thread_values$
    for tiles in bounding box **do**
      if $valid$ **then**
        Compute and store the keys and values

DuplicateWithKeys) within our adaptive scheduling framework, we reduce data dependencies and reuse intermediate results, further improving efficiency.

3.4. Latency-aware Pipelined Tile Rendering

Tile-level Imbalance The precise intersection reduces the number of tile-Gaussian pairs in rendering. However, achieving low-latency rendering for these pairs remains challenging. While tile-based algorithms leverage GPU thread-level parallelism, computational workloads vary significantly across tiles [45]. Overall latency is often dictated by the slowest tiles, which may contain significantly more overlapping Gaussians. Unlike Gaussian size variations, this imbalance cannot be mitigated by simply remapping tasks to more threads, as each tile encompasses numerous pixels (e.g., a 16×16 tile contains 256 pixels). Each task must utilize at least one warp's resources, and assigning additional thread groups would incur extra overhead.

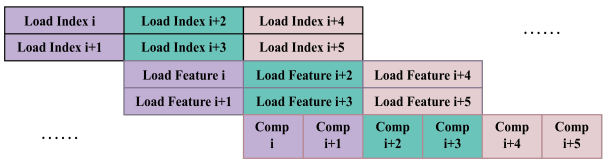


Figure 5. Software pipelining to achieve better overlap between computation and memory access stages, with three stages and two sub-blocks within an iteration.

Software Pipeline to Hide Latency Each tile computes the contribution of relevant Gaussians loaded from global memory, based on a sorted Gaussian-tile list. This involves reading Gaussian indices, retrieving the corresponding features, and computing colors. To mitigate the performance degradation from these dependencies, our pipeline overlaps the computation of different Gaussians within a tile. Specifically, in the i-th iteration: we first access the sorted key-value list to obtain the Gaussian IDs for the $i+2$-th iteration; then, using buffered indices from the $i+1$-th iteration, we load the corresponding Gaussian features; finally, we perform alpha blending using the Gaussian features buffered from previous steps. Additionally, we prefetch the indices and features of two tiles at once. Since memory access latency generally exceeds computation latency for a single tile, we overlap the computation of two tiles with the prefetch latency of the subsequent two tiles.

Notably, unlike the previous section where we address load imbalance by increasing parallelism, we cannot adopt this approach here due to data dependencies. Thread-level parallelism alone is insufficient to effectively hide this latency. Moreover, while parallelism across multiple warps on the GPU can mask some latency through overlap, it does not resolve the issue of prolonged computation latencies in individual tasks, which may still bottleneck overall performance. Our goal is to minimize the computational latency of high-load tasks, preventing them from becoming end-to-end performance bottlenecks.

4. Evaluation

4.1. Experimental Setup

We conduct comprehensive tests on datasets of varying scales and resolutions, summarized in Table 1. Among these, Rubble [43] and MatrixCity [32] are large-scale, high-resolution datasets, which exceed the real-time rendering capabilities of existing 3DGS-based systems. Additionally, we evaluate commonly used 3DGS datasets. Truck and Train are outdoor scenes from the Tanks & Temple [24]. Playroom and DrJohnson are two indoor scenes in DeepBlending [17]. All evaluations use the same pre-trained models, except for AdRGS-Full in Table 1, which requires additional training to balance the Gaussian distribution [45]. Due to memory constraints, we were unable to retrain AdRGS-Full for MatrixCity and Rubble and directly use pre-trained models.

We measure rendering speed using FPS and evaluate image quality with PSNR, SSIM, and LPIPS. To assess performance stability, we report both the average FPS across all frames and the FPS of the slowest frame in each scene. All experiments are conducted on an NVIDIA RTX 3090 GPU (24GB VRAM).

FlashGS is implemented as a high-performance CUDA library with Python bindings, incorporating low-level optimizations, including assembly-level optimizations. Further details are provided in the supplementary materials.

4.2. Performance Comparison

We compare FlashGS with latest 3DGS [21], gsplat (v1.4.0) [48], and AdR-Gaussian [45] (abbreviated as

Table 1. Quantitative evaluation of rendering speed for FlashGS and related works, including large-scale and high-resolution scenes as well as other commonly used datasets. The same pre-trained model is used for different rasterizers, except that AdRGS-Full requires retraining to achieve performance gains (cell marked with*).

AvgFps ↑ / MinFPS ↑	Truck		Train		Playroom		DrJohnson		MatrixCity		Rubble
	1080p	4k	1080p	4k	1080p	4k	1080p	4k	1080p	4k	4608*3456
3DGS	121.8 / 98.2	41.3 / 33.5	127.9 / 78.0	40.2 / 23.4	146.4 / 93.1	47.2 / 31.1	109.8 / 59.1	34.8 / 17.3	47.9 / 24.6	15.0 / 7.1	20.7 / 4.7
gsplat	99.6 / 80.6	35.0 / 29.0	105.9 / 65.6	34.1 / 19.9	118.1 / 72.5	39.7 / 26.1	88.2 / 48.4	29.2 / 14.8	35.5 / 19.5	12.2 / 6.1	17.5 / 14.4
AdRGS-AABB	286.6 / 179.5	131.9 / 106.8	330.6 / 237.5	138.0 / 104.3	354.6 / 167.6	149.2 / 83.4	319.0 / 141.4	139.2 / 78.6	142.0 / 102.7	68.8 / 47.5	59.0 / 43.3
AdRGS-Full	411.1 / **307.4** *	166.2 / 133.3 *	471.5 / **316.7** *	176.5 / 140.5 *	552.0 / 335.5 *	211.0 / 134.0 *	486.6 / **229.9***	196.1 / 116.5*	142.0 / 102.7	68.8 / 47.5	59.0 / 43.3
FlashGS-Pre	360.2 / 209.2	190.8 / 156.3	406.8 / 266.5	203.8 / 158.9	453.0 / 250.7	223.6 / 138.6	428.7 / 163.6	204.8 / 100.3	239.5 / 169.2	117.1 / 80.2	97.6 / 69.1
FlashGS-Full	**450.5** / 225.2	**289.0 / 215.1**	**518.1** / 287.4	**301.2 / 237.0**	**694.4 / 342.5**	**367.7 / 213.2**	**613.5** / 202.0	**334.4 / 144.7**	**310.6 / 209.6**	**207.4 / 139.6**	**183.9 / 125.9**
FlashGS-w/o	267.1 / 161.8	103.0 / 73.6	267.5 / 179.1	94.8 / 68.9	364.2 / 209.8	145.1 / 81.3	312.5 / 152.2	113.2 / 63.9	185.4 / 126.4	70.0 / 50.9	60.4 / 32.8

Table 2. Quantitative results of the rendered image quality across all datasets (at the original resolution of the ground truth). Results marked with gray color indicate that the model was retrained rather than using the pre-trained model by vanilla 3DGS.

Quality	Truck	Train	Playroom	Drjohnson	MatrixCity	Rubble
	PSNR↑ / SSIM↑ / LPIPS↓	PSNR↑ / SSIM↑ / LPIPS↓	PSNR↑ / SSIM↑ / LPIPS↓	PSNR↑ / SSIM↑ / LPIPS↓	PSNR↑ / SSIM↑ / LPIPS↓	PSNR↑ / SSIM↑ / LPIPS↓
3DGS	27.24 / 0.9073 / 0.0576	25.10 / 0.8619 / 0.1111	35.39 / 0.9511 / 0.0626	37.16 / 0.9524 / 0.0439	33.78 / 0.9315 / 0.0696	22.72 / 0.6266 / 0.3570
gsplat	27.22 / 0.9069 / 0.0575	24.84 / 0.8492 / 0.1140	35.31 / 0.9507 / 0.0625	37.13 / 0.9521 / 0.0439	33.65 / 0.9310 / 0.0697	22.72 / 0.6264 / 0.3557
AdRGS-AABB	27.20 / 0.9070 / 0.0575	25.07 / 0.8618 / 0.1109	35.35 / 0.9509 / 0.0626	37.14 / 0.9523 / 0.0439	33.74 / 0.9315 / 0.0696	22.68 / 0.6262 / 0.3573
AdRGS-Full	26.83 / 0.9010 / 0.1053	24.22 / 0.8310 / 0.1477	34.77 / 0.9443 / 0.0740	36.49 / 0.9477 / 0.0504	33.74 / 0.9315 / 0.0696	22.68 / 0.6262 / 0.3573
FlashGS-Full	27.20 / 0.9063 / 0.0599	25.06 / 0.8607 / 0.1129	35.24 / 0.9505 / 0.0639	36.98 / 0.9511 / 0.0449	33.68 / 0.9308 / 0.0704	22.74 / 0.6273 / 0.3568

AdRGS). The rendering speed and image quality are shown in Table 1 and Table 2. AdRGS-AABB represents the optimization with tighter AABB in AdRGS, and AdRGS-Full includes further optimizations with the load-balanced retrained model. FlashGS-Pre represents our optimizations related to preprocessing (precise intersection and scheduling), FlashGS-Full further incorporates optimizations for the render operator, and FlashGS-w/o refers to FlashGS-Full without Gaussian size-aware scheduling.

Rendering Speed FlashGS achieves real-time rendering ($\geq$ 125.9FPS) across all tested scenes, as shown in Table 1. The fully optimized FlashGS-Full consistently delivers the highest average FPS in all scenarios, achieving a 7.2× speedup over 3DGS on average and an 8.7× speedup on the 7 large-scale or high-resolution tests. Compared to the current state-of-the-art, AdRGS-Full, FlashGS-Full achieves an average of 1.8× speedup and 2.2× speedup on average and on 7 large-scale or high-resolution scenes. FlashGS-Pre, which implements our precise intersection algorithm, also achieves an average 1.5× speedup over AdRGS-AABB while maintaining real-time rendering, highlighting the importance of precise intersection. For the slowest frames (MinFPS), FlashGS-Full also shows superior performance. FlashGS achieves 125.9 FPS on the slowest frame of the Rubble dataset, while AdRGS achieves only 43.3 FPS, and gsplat and vanilla 3DGS only achieve 14.4 FPS and 4.7 FPS, respectively. However, in relatively low-resolution outdoor scenes such as Truck-1080p and Train-1080p, AdRGS outperforms FlashGS in MinFPS. This may be due to the uneven distribution of objects in these scenes, where lower resolutions amplify the effect (with coarser tiles, Gaussians tend to cluster within single tiles). By retraining, AdRGS can influence the densification process in vanilla 3DGS, leading to a more balanced Gaussian distribution. Additionally, gsplat shows the lowest rendering performance. This is partly due to its design with richer features than one focused on rendering acceleration. Rich feature components may lead to a complex implementation with many different interfaces, potentially reducing hardware utilization.

Image Quality Table 2 presents the image quality metrics of different implementations, evaluating whether the rendering speedup compromises visual fidelity. We can observe that the image quality of vanilla 3DGS, gsplat, AdRGS-AABB, and FlashGS-Full is very similar. In contrast, the re-trained model of AdRGS-Full (gray-shaded cells in Table 2) exhibits relatively higher rendering errors compared to other methods. This discrepancy likely arises because AdRGS modifies the Gaussian distribution during scene reconstruction. In scenes where objects are inherently unevenly distributed, enforcing a more uniform reconstruction may reduce accuracy. FlashGS-Full maintains high image quality, with only a slight accuracy drop in some scenes. This may be attributed to the use of more efficient assembly-level optimizations in our implementation. Further details are provided in the supplementary materials.

4.3. Ablation Study

Precised Intersection As shown in Table 1, our precise intersection algorithm (FlashGS-Pre) significantly improves performance, outperforming vanilla 3DGS by 4.5× on average and AdRGS-AABB by 1.5× on average. This

Table 3. Quantitative results of the maximum number of intersections for rendering, i.e., the overlapped Gaussian-tile pair.

Max Intersections	Truck 1080p	Truck 4k	Train 1080p	Train 4k	Playroom 1080p	Playroom 4k	DrJohnson 1080p	DrJohnson 4k	Matrixcity 1080p	Matrixcity 4k	Rubble 4608*3456
3DGS	14,199,983	47,273,716	18,943,297	65,482,520	13,377,659	45,646,667	24,797,099	85,451,242	56,148,670	207,330,839	91,690,486
gsplat	14,604,417	48,044,861	19,318,054	66,374,552	13,651,326	46,164,736	25,211,011	86,245,982	56,996,305	208,933,575	92,773,080
AdRGS	4,198,279	11,612,374	4,449,273	13,008,020	4,577,658	12,160,724	6,662,625	15,492,906	8,454,262	24,564,554	30,346,782
FlashGS	**3,805,372**	**8,655,273**	**3,729,928**	**9,490,331**	**3,660,958**	**7,966,705**	**6,388,792**	**13,615,019**	**6,108,576**	**13,549,417**	**19,905,064**

demonstrates our algorithm's effectiveness in eliminating false Gaussian coverage—the core source of rasterization workload in 3DGS. The results confirm that our algebraic and geometric optimizations reduce equation-solving overhead without compromising overall performance.

Pipelined Rendering Building on the workload reduction from the precise intersection, FlashGS-Full achieves an additional 1.5× speedup through pipelined rendering. We overlap Gaussian feature memory access with alpha blending computations to minimize data dependency stalls. This optimization proves crucial for scenes with numerous Gaussians, where computational load becomes the bottleneck. Notably, we preserve the natural Gaussian distribution rather than artificially balancing loads.

Adaptive Scheduling Removing adaptive scheduling (FlashGS-w/o) reveals its critical importance, as shown in Table 1. Without size-aware scheduling, workload imbalance from varying Gaussian sizes significantly impacts performance—an issue overlooked in prior work. FlashGS-w/o underperforms AdR-AABB because our precise intersection increases per-thread workload, exacerbating the imbalance in Gaussian-tile pair construction.

4.4. Redundancy Reduction

Table 3 shows the maximum Gaussian-tile intersection counts across 11 scenes using identical pre-trained models, where intersection counts directly determine rasterization performance through storage and computation overhead. FlashGS achieves the fewest intersections with 81.4% average redundancy reduction versus 3DGS, and an additional 23.3% reduction versus AdRGS's adaptive AABB—particularly evident in MatrixCity-4K (93.5% and 44.9% reductions versus 3DGS and AdRGS respectively). GSplat shows increased intersections versus vanilla 3DGS (unexplained in their original work). Our precise intersection algorithm demonstrates superior efficiency by perfectly integrating with 3DGS's tile-based rasterizing mechanism.

4.5. Compared to Hardware Rasterization Methods

Some approaches have integrated hardware rasterization into 3DGS rendering, utilizing APIs such as OpenGL [22], Vulkan [23], and WebGPU [44]. We compared these hardware-accelerated methods with the CUDA-based vanilla 3DGS and our optimized FlashGS. Table 4 presents the performance of two representative hardware rasterization methods—web-splat [39] and fast-gaussian-rasterization (fast_gauss) [1]—alongside our approach. FlashGS achieves over a 2.5× speedup compared to these methods. This performance gap primarily stems from the inherent inefficiencies of rendering Gaussian primitives using traditional hardware rasterization pipelines: (1) Representing Gaussian ellipses with planar quads introduces an inherent redundancy of $1 - \pi/4$, leading to unnecessary computations. (2) A large number of Gaussians results in many small triangles, significantly lowering the pixel-to-point ratio and reducing GPU efficiency [3, 25]. These limitations highlight the advantages of our optimized CUDA-based approach in efficiently handling Gaussian splatting.

Table 4. Rendering speed of software/hardware rasterizations (web-splat failed on MatrixCity with over 20 million Gaussians).

AvgFPS ↑	Truck		Train		Playroom		DrJohnson		MatrixCity		Rubble
Resolution	1080p	4k	1080p	4k	1080p	4k	1080p	4k	1080p	4k	4608*3456
3DGS	121.8	41.3	127.9	40.2	146.4	47.2	109.8	34.8	47.9	15.0	20.7
fast_gauss	47.6	39.4	78.7	54.1	73.2	56.5	28.1	17.3	22.6	19.5	23.4
web-splat	323.5	114.6	297.1	96.1	401.1	133.9	317.1	102.4	failed	failed	44.14
FlashGS	**450.5**	**289.0**	**518.1**	**301.2**	**694.4**	**367.7**	**613.5**	**334.4**	**310.6**	**207.4**	**183.9**

5. Conclusion

3DGS encounters significant challenges on large-scale and high-resolution scenes, hindering its further application. In this work, we propose FlashGS, enabling real-time rendering of large-scale and high-resolution scenes. We achieve efficient rasterization by integrating comprehensive optimizations on algorithm and operations, rendering at over 125.9 FPS on real-world scale high quality datasets. Our future work will focus on enhancing more specific applications. The performance is foreseeable, as our approach does not heavily rely on assumptions from a certain algorithm.

Acknowledgement This work is supported by the National Key R&D Program of China (No. 2022ZD0160201), Shanghai Artificial Intelligence Laboratory, and Shanghai Municipal Science and Technology Major Project.

References

[1] Fast gaussian rasterization. GitHub, 2024. 8

[2] Yara Jamil Alkhatib, Anna Forte, Gabriele Bitelli, Roberto Pierdicca, and Eva Malinverni. Bringing back lost heritage into life by 3d reconstruction in metaverse and virtual environments: The case study of palmyra, syria. In *International Conference on Extended Reality*, pages 91–106. Springer, 2023. 2

[3] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus. In *11th Pacific Conference on Computer Graphics and Applications, 2003. Proceedings.*, pages 335–343. IEEE, 2003. 8

[4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3 (1):1–122, 2011. 3

[5] Yu Chen and Gim Hee Lee. Dogs: Distributed-oriented gaussian splatting for large-scale 3d reconstruction via gaussian consensus. *Advances in Neural Information Processing Systems*, 37:34487–34512, 2024. 3

[6] Yihang Chen, Qianyi Wu, Mengyao Li, Weiyao Lin, Mehrtash Harandi, and Jianfei Cai. Fast feedforward 3d gaussian splatting compression. *arXiv preprint arXiv:2410.08017*, 2024. 3

[7] Yihang Chen, Qianyi Wu, Weiyao Lin, Mehrtash Harandi, and Jianfei Cai. Hac: Hash-grid assisted context for 3d gaussian splatting compression. In *European Conference on Computer Vision*, 2024. 3

[8] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4):65–74, 1988. 3

[9] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, and Nandita Vijaykumar. Distwar: Fast differentiable rendering on raster-based rendering pipelines. *arXiv preprint arXiv:2401.05345*, 2023. 3

[10] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, Zhangyang Wang, et al. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *Advances in neural information processing systems*, 37: 140138–140158, 2024. 2

[11] Guangchi Fang and Bing Wang. Mini-splatting: Representing scenes with a constrained number of gaussians. In *European Conference on Computer Vision*, pages 165–181. Springer, 2024. 3

[12] Linus Franke, Laura Fink, and Marc Stamminger. Vr-splatting: Foveated radiance field rendering via 3d gaussian splatting and neural points. *arXiv preprint arXiv:2410.17932*, 2024. 2

[13] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5501–5510, 2022. 3

[14] Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. Eagles: Efficient accelerated 3d gaussians with lightweight encodings. In *European Conference on Computer Vision*, pages 54–71. Springer, 2024. 2

[15] Florian Hahlbohm, Fabian Friederichs, Tim Weyrich, Linus Franke, Moritz Kappel, Susana Castillo, Marc Stamminger, Martin Eisemann, and Marcus Magnor. Efficient perspective-correct 3d gaussian splatting using hybrid transparency. *arXiv preprint arXiv:2410.08129*, 2024. 3

[16] Abdullah Hamdi, Luke Melas-Kyriazi, Jinjie Mai, Guocheng Qian, Ruoshi Liu, Carl Vondrick, Bernard Ghanem, and Andrea Vedaldi. Ges: Generalized exponential splatting for efficient radiance field rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19812–19822, 2024. 3

[17] 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 (ToG)*, 37(6):1–15, 2018. 6

[18] Letian Huang, Jiayang Bai, Jie Guo, Yuanqi Li, and Yanwen Guo. On the error analysis of 3d gaussian splatting and an optimal projection strategy. In *European conference on computer vision*, pages 247–263. Springer, 2024. 3

[19] Ying Jiang, Chang Yu, Tianyi Xie, Xuan Li, Yutao Feng, Huamin Wang, Minchen Li, Henry Lau, Feng Gao, Yin Yang, et al. Vr-gs: a physical dynamics-aware interactive gaussian splatting system in virtual reality. In *ACM SIGGRAPH 2024 Conference Papers*, pages 1–1, 2024. 2

[20] Nikhil Varma Keetha, Jay Karhade, Krishna Murthy Jatavallabhula, Gengshan Yang, Sebastian Scherer, Deva Ramanan, and Jonathon Luiten. Splatam: Splat, track & map 3d gaussians for dense rgb-d slam. *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21357–21366, 2023. 2

[21] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3D gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics (TOG)*, 42:1–14, 2023. 2, 3, 6

[22] Khronos Group. OpenGL Specification. Technical report, Khronos Group, 2025. 8

[23] Khronos Group. Vulkan API Specification. Technical report, Khronos Group, 2025. 8

[24] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)*, 36 (4):1–13, 2017. 6

[25] Samuli Laine and Tero Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, 2011. 8

[26] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1440–1449, 2021. 2

[27] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong 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, Volume 3*, pages 497–511, 2024. 2, 3

[28] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21719–21728, 2024. 3

[29] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. Compact 3d gaussian representation for radiance field. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21719–21728, 2024. 2

[30] Jiahe Li, Jiawei Zhang, Xiao Bai, Jin Zheng, Xin Ning, Jun Zhou, and Lin Gu. DNGaussian: Optimizing sparse-view 3D gaussian radiance fields with global-local depth normalization. *arXiv preprint arXiv:2403.06912*, 2024. 3

[31] Sixu Li, Chaojian Li, Wenbo Zhu, Boyang Yu, Yang Zhao, Cheng Wan, Haoran You, Huihong Shi, and Yingyan 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*, pages 1–13, 2023. 2

[32] Yixuan Li, Lihan Jiang, Linning Xu, Yuanbo Xiangli, Zhenzhi Wang, Dahua Lin, and Bo Dai. Matrixcity: A large-scale city dataset for city-scale neural rendering and beyond. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3205–3215, 2023. 1, 6

[33] Yanyan Li, Chenyu Lyu, Yan Di, Guangyao Zhai, Gim Hee Lee, and Federico Tombari. Geogaussian: Geometry-aware gaussian splatting for scene rendering. In *European Conference on Computer Vision*, pages 441–457. Springer, 2025. 3

[34] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20654–20664, 2024. 2, 3

[35] Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. Taming 3dgs: High-quality radiance fields with limited resources. In *SIGGRAPH Asia 2024 Conference Papers*, pages 1–11, 2024. 3

[36] 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. *Communications of the ACM*, 65(1):99–106, 2021. 3

[37] Nicolas Moenne-Loccoz, Ashkan Mirzaei, Or Perel, Riccardo de Lutio, Janick Martinez Esturo, Gavriel State, Sanja Fidler, Nicholas Sharp, and Zan Gojcic. 3d gaussian ray tracing: Fast tracing of particle scenes. *ACM Transactions on Graphics (TOG)*, 43(6):1–19, 2024. 3

[38] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)*, 41(4):1–15, 2022. 3

[39] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10349–10358, 2024. 3, 8

[40] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. Stopthepop: Sorted gaussian splatting for view-consistent real-time rendering. *ACM Transactions on Graphics (TOG)*, 43(4):1–17, 2024. 3

[41] Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. Octree-GS: Towards consistent real-time rendering with lod-structured 3D gaussians. *arXiv preprint arXiv:2403.17898*, 2024. 3

[42] Jiakai Sun, Han Jiao, Guangyuan Li, Zhanjie Zhang, Lei Zhao, and Wei Xing. 3dgstream: On-the-fly training of 3d gaussians for efficient streaming of photo-realistic free-viewpoint videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 20675–20685, 2024. 3

[43] Haithem Turki, Deva Ramanan, and Mahadev 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*, pages 12922–12931, 2022. 6

[44] W3C. WebGPU Specification, 2025. 8

[45] Xinzhe Wang, Ran Yi, and Lizhuang Ma. Adr-gaussian: Accelerating gaussian splatting with adaptive radius. In *ACM SIGGRAPH Asia 2024 Conference Proceedings*, pages 1–10, 2024. 2, 3, 6

[46] Linning Xu, Yuanbo Xiangli, Sida Peng, Xingang Pan, Nanxuan Zhao, Christian Theobalt, Bo Dai, and Dahua Lin. Grid-guided neural radiance fields for large urban scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8296–8306, 2023. 3

[47] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5438–5448, 2022. 3

[48] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, and Angjoo Kanazawa. gsplat: An open-source library for Gaussian splatting. *arXiv preprint arXiv:2409.06765*, 2024. 3, 6

[49] Mulin Yu, Tao Lu, Linning Xu, Lihan Jiang, Yuanbo Xiangli, and Bo Dai. Gsdf: 3dgs meets sdf for improved rendering and reconstruction. *arXiv preprint arXiv:2403.16964*, 2024. 3

[50] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3d gaussian splatting. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 19447–19456, 2024. 3

[51] Cheng Zhao, Su Sun, Ruoyu Wang, Yuliang Guo, Jun-Jun Wan, Zhou Huang, Xinyu Huang, Yingjie Victor Chen, and Liu Ren. Tclc-gs: Tightly coupled lidar-camera gaussian splatting for surrounding autonomous driving scenes. *arXiv preprint arXiv:2404.02410*, 2024. 2

[52] Hexu Zhao, Haoyang Weng, Daohan Lu, Ang Li, Jinyang Li, Aurojit Panda, and Saining Xie. On scaling up 3d gaussian splatting training. *arXiv preprint arXiv:2406.18533*, 2024. 3

[53] Xiaoyu Zhou, Zhiwei Lin, Xiaojun Shan, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. Drivinggaussian: Composite gaussian splatting for surrounding dynamic autonomous driving scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21634–21643, 2024. 2