SLGS: A Structure-Aware Scanline Renderer for Efficient 3D Gaussian Splatting

Seunghyun Lee
Yonsei University
Seoul, Republic of Korea
seunghyun.lee@yonsei.ac.kr

Mingu Jung
Yonsei University
Seoul, Republic of Korea
mingu.jung@yonsei.ac.kr

Sungwoo Kim
Yonsei University
Seoul, Republic of Korea
sungwoo.kim@yonsei.ac.kr

Sungbin Kim
Yonsei University
Seoul, Republic of Korea
sungbin.kim@yonsei.ac.kr

Dongho Ha*
Meta Platforms
Sunnyvale, CA, USA
dongho9601@gmail.com

Hyunwuk Lee
Gwangju Institute of Science and Technology
Gwangju, Republic of Korea
hyunwuklee@gist.ac.kr

Yingyan (Celine) Lin
Georgia Institute of Technology
Atlanta, GA, USA
ylin715@gatech.edu

Won Woo Ro
Yonsei University
Seoul, Republic of Korea
wro@yonsei.ac.kr

Abstract

3D Gaussian Splatting (3DGS) has become a practical representation for high-quality novel view synthesis by rendering scenes as explicit Gaussian primitives. However, its forward pipeline is still dominated by sorting and rendering, and prior accelerators largely optimize these stages within the same square-tile-based execution framework. Our analysis shows that this organization is poorly matched to three structural properties of Gaussians. First, neighboring regions share strong candidate-set and ordering continuity, but square tiles weaken this locality through large tile strides. Second, each Gaussian's effective projected area is available before rendering, yet exploiting it under two-dimensional tiles is expensive, leaving unnecessary pixel-level work and fewer opportunities to process non-overlapping Gaussians together. Third, projected Gaussians are often highly anisotropic with locally correlated dominant orientations, whereas square tiles remain isotropic work units.

To address this mismatch, we propose a *scanline-oriented rendering* scheme and a corresponding in-GPU accelerator, *SLGS*. The scanline rendering pipeline partitions the image into macro-tiles and processes them as direction-aligned scanlines selected according to local Gaussian orientation. This organization enables ordering-state reuse across neighboring scanlines, low-overhead effective-area handling, packetized execution of non-overlapping Gaussian work, and 32-lane front-to-back compositing with lane-level feedback. Overall, SLGS achieves 12.43× speedup over original 3DGS, 2.36× speedup over the strongest prior accelerator, and 52.6–78.5% lower rendering energy than prior accelerators.

CCS Concepts

• **Computing methodologies** → **Rendering**; • **Computer systems organization** → **Special purpose systems**.

*This research was done in the author's personal capacity, and the views in the paper do not represent those of Meta.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846836

Keywords

Novel View Synthesis, 3D Gaussian Splatting, Scanline Rendering, Hardware Acceleration, GPU Architecture, Neural Rendering

1 Introduction

Novel view synthesis (NVS) renders images from unseen viewpoints given a set of observed views, enabling applications such as virtual and augmented reality, autonomous driving, robotics, and free-viewpoint media [10, 18, 20, 23, 32, 33]. Neural Radiance Fields (NeRF) [24] demonstrated photorealistic NVS, but dense ray sampling and volume integration make both training and inference expensive [5, 8, 19, 29, 31]. 3D Gaussian Splatting (3DGS) [12] addresses this cost by representing a scene as a set of explicit, learnable 3D Gaussians and rendering them directly in screen space, making explicit primitive-based rendering a practical alternative to implicit MLP-based neural rendering.

Despite this efficiency, 3DGS inference is still dominated by sorting and rendering [4, 9, 14, 16, 28]. In the original 3DGS pipeline, Gaussians projected into screen space are first assigned to 16×16 square tiles through conservative proxy regions, then sorted independently within each tile, and finally composited through tile-cooperative, front-to-back execution. Prior accelerators [11, 15, 17, 26, 27, 35–37] have identified several inefficiencies that arise from this tile-based organization, including candidate inflation, redundant tile-local sorting, and ineffective raster work. However, they primarily address these inefficiencies by adding optimizations within the tile-based pipeline, such as tighter tile admission, temporal sorting reuse or sorting reduction, and subtile-level skipping. As a result, the underlying square-tile or tile-derived rendering organization remains largely unchanged.

This raises two more fundamental questions. Is tile-based rendering itself the most efficient execution organization for 3DGS? More importantly, is a square-tile work unit well matched to the structural properties of Gaussian primitives? To answer these questions, we revisit three rendering-relevant Gaussian properties and their interaction with square tiles. First, neighboring image-space regions often share a large fraction of Gaussian candidates and preserve similar depth ordering, implying strong spatial locality and ordering continuity [4, 6, 21]. Square tiles, however, group pixels into 16×16 units and move with a stride of at least 16 pixels, which

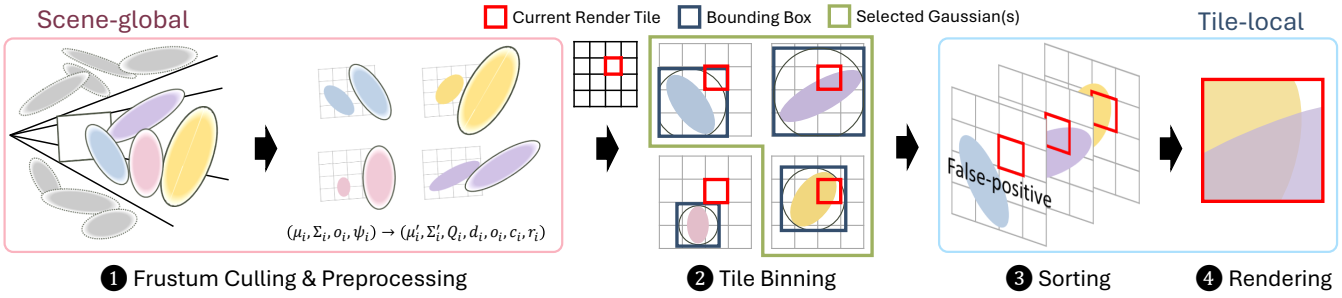


Figure 1: Forward pipeline of 3D Gaussian Splatting.

weakens the finer-grained locality already present at pixel level and limits reuse across neighboring regions. Second, after projection, each Gaussian has a well-defined screen-space region in which it can meaningfully affect pixels [38, 39]; we refer to this region as its effective projected area. Square-tile execution, however, handles this region conservatively because checking its exact overlap with a two-dimensional tile is expensive. This leads to substantial computation on pixels outside the region and reduces the opportunity to execute non-overlapping Gaussians together. Third, projected Gaussians are often highly anisotropic, and nearby Gaussians frequently share similar dominant orientations [6, 21, 39]. Square tiles, however, are isotropic work units and therefore do not naturally reflect this directional structure.

These observations motivate a different execution organization. In this work, we propose a *scanline-oriented rendering* scheme for 3D Gaussian Splatting and a corresponding in-GPU accelerator, *SLGS*. Instead of rendering independent square tiles, SLGS partitions the image into macro-tiles and processes each macro-tile as orientation-selected horizontal or vertical scanlines. This enables ordering-state reuse, exact support generation before compositing, packetized non-overlap execution, and 32-lane front-to-back compositing with lane-level feedback.

This reorganization provides three concrete advantages. First, scanline traversal preserves spatial locality and ordering continuity more effectively than square-tile execution, making reuse more natural across neighboring work units. Second, effective projected area is exploited at low overhead, enabling efficient elimination of unnecessary pixel-level work and exposing more opportunities to process non-overlapping Gaussians concurrently. Third, scanline direction can be aligned with dominant local orientation, improving unit utilization for highly anisotropic Gaussians. As a result, the proposed scheme achieves 12.43× speedup over original 3DGS and 2.36× over the strongest evaluated prior accelerator on the common workload; after normalizing accelerator designs to 28 nm and 1 GHz, SLGS reduces average per-frame rendering energy by 52.6–78.5% relative to the evaluated prior accelerators.

The contributions of this work are as follows.

- We identify a structural mismatch between square-tile execution and Gaussian structure in 3DGS, showing that spatial locality, effective projected area, and anisotropy are only weakly exposed under tile-based organization.
- We propose a scanline-oriented rendering scheme that combines direction-aligned macro-tile decomposition, scanline-local ordering-state reuse, pixel-level exact support generation, and conflict-aware packetized compositing.
- We design SLGS, an in-GPU accelerator with bitmap-native scheduling and done-aware runtime optimization, enabling Gaussian rendering to exploit exact support and lane-level feedback more directly than conventional square-tile pipelines.

2 Background & Prior Work

2.1 Novel View Synthesis

Novel view synthesis (NVS) renders unseen viewpoints from observed images and underpins applications such as AR/VR, robotics, and autonomous systems. Neural Radiance Fields (NeRF) [24] achieved photorealistic NVS with implicit neural fields, but their dense ray sampling and volumetric integration make inference costly. 3D Gaussian Splatting (3DGS) [12] instead represents a scene as explicit learnable Gaussians. To render a new view, it projects these Gaussians onto the image plane, orders them by depth, and accumulates their opacity-weighted color contributions from front to back. This shifts NVS from ray-based field integration to ordered processing of projected Gaussian primitives, making the rendering pipeline amenable to architectural acceleration.

2.2 3D Gaussian Splatting

The key shift introduced by 3D Gaussian Splatting is to represent a scene not as an implicit radiance field but as a set of learnable explicit Gaussian primitives,

$$\mathcal{G} = \{g_i\}_{i=1}^{N}.$$

Each Gaussian g_i is parameterized by a 3D mean μ_i, covariance Σ_i, opacity o_i, and appearance parameter ψ_i, such as view-dependent color coefficients. During training, these parameters are optimized and density control refines the Gaussian set. As a result, the forward renderer of 3DGS can be understood as a screen-space rendering pipeline that processes explicit primitives projected to the current view, rather than directly sampling an implicit field along rays.

As shown in Figure 1, the original forward pass of 3DGS for a given camera view v proceeds as follows.

1 Frustum Culling & Preprocessing. First, Gaussians that do not participate in rendering for the current view are removed, yielding

$$\mathcal{G}_v^{\text{vis}} \subseteq \mathcal{G}.$$

Each $g_i \in \mathcal{G}_v^{\text{vis}}$ is then projected onto the current view to compute the projected mean, projected 2D covariance, and inverse conic:

$$\mu_i' = \pi(\mu_i), \qquad \Sigma_i' = J_i W \Sigma_i W^\top J_i^\top, \qquad Q_i = (\Sigma_i')^{-1},$$

where W denotes the world-to-camera transform and J_i is the Jacobian of the camera projection evaluated at Gaussian g_i. The renderer also materializes depth d_i, opacity o_i, color c_i, and radius r_i. Later stages therefore operate on the view-dependent screen-space state

$$\phi_i \triangleq (\mu_i', \Sigma_i', Q_i, o_i, d_i, c_i, r_i),$$

which captures the projected position, geometry, and appearance of each Gaussian.

❷ **Tile binning.** The image plane is partitioned into a set of 16×16 square tiles $\mathcal{T}$. Each Gaussian is assigned to tiles using a conservative proxy bounding box, rather than its effective projected area. In the original implementation, a conservative square axis-aligned proxy A_i^{bbox} is derived from the projected center μ_i' and a projected extent proxy r_i, and the Gaussian is duplicated to every overlapping tile. As a result, each tile $t \in \mathcal{T}$ obtains the candidate set

$$C_t = \{\, g_i \in \mathcal{G}_v^{\text{vis}} \mid t \cap A_i^{\text{bbox}} \neq \emptyset \,\}.$$

Since A_i^{bbox} is only a bounding proxy, a Gaussian can be included as a candidate even when its effective projected area does not intersect the tile. In this paper, we refer to such Gaussian-tile assignments as false-positive candidates. A Gaussian with a large screen-space extent may also be replicated across many tiles, after which each tile processes it through its own sorting and compositing stages.

❸ **Tile local depth sorting.** The candidates in each tile are sorted in front-to-back order according to depth d_i, yielding an ordered stream σ_t:

$$d_{\sigma_t(1)} \leq d_{\sigma_t(2)} \leq \cdots \leq d_{\sigma_t(|C_t|)}.$$

This lets pixels in a tile share one compositing order, but sorting is reused only within that tile. A Gaussian spanning multiple tiles therefore appears in multiple ordered streams, preventing direct reuse across tile boundaries.

❹ **Tile cooperative rasterization.** Finally, pixels inside a tile consume the shared ordered stream σ_t and perform front-to-back alpha compositing. For a pixel p, we first define the quadratic distance term

$$q_i(p) = (p - \mu_i')^\top Q_i (p - \mu_i')$$

which measures the distance from p to the projected Gaussian center under the inverse-covariance metric Q_i. The corresponding per-pixel opacity contribution is

$$\alpha_i(p) = o_i \exp\left(-\frac{1}{2} q_i(p)\right).$$

Color C and transmittance T are then accumulated as

$$C(p) = \sum_k T_{k-1}(p)\, \alpha_{\sigma_t(k)}(p)\, c_{\sigma_t(k)}, \;\; T_k(p) = T_{k-1}(p)\big(1 - \alpha_{\sigma_t(k)}(p)\big).$$

Practical implementations additionally prune negligible opacity contributions and terminate compositing once the remaining transmittance falls below a threshold. In the actual GPU implementation, one 16×16 tile corresponds to one thread block, and 256 threads correspond to the 256 pixels within the tile. These threads traverse the tile's ordered Gaussian stream together. At each iteration, all threads fetch the same Gaussian, but each thread computes the Gaussian's opacity contribution $\alpha_i(p)$ for its own pixel based on the pixel's distance from the projected Gaussian center, and then updates that pixel's color and transmittance. This improves fetch reuse and order sharing, but forces pixels with little or no overlap to participate in tile-level execution.

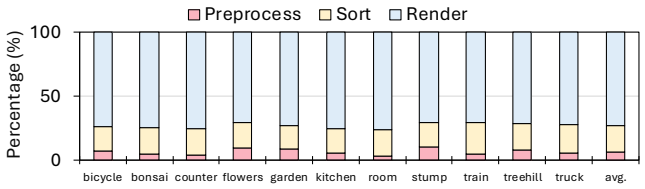


Figure 2: Latency breakdown of 3DGS across pipeline stages for various scenes.

2.3 Accelerating 3DGS

Figure 2 shows the stage-wise latency breakdown of the original 3DGS forward pipeline. In our profiling, rendering is the dominant contributor to total latency, followed by sorting, while preprocessing contributes only marginally. On average, rendering accounts for 72.96% of total latency, sorting for 20.49%, and preprocessing for the remaining 6.55%. This indicates that accelerating 3DGS requires reducing the cost of both rendering and sorting.

Prior work has addressed these costs through tighter candidate admission, reduced sorting overhead, and more efficient raster work. GSCore [15] combines shape-aware tile admission, hierarchical sorting, and subtile skipping; AdR-Gaussian [34] reduces each Gaussian's rendering range with opacity-aware boundaries; GBU [37] accelerates blending with an intra-tile row-wise dataflow; Speedy-Splat [7] improves Gaussian-to-tile admission through row/column traversal; GauRast [17] accelerates Gaussian rasterization using an enhanced GPU rasterizer; GS-TG [11] shares sorting across spatial tile groups; and Neo [26] reuses ordering across frames. While these techniques optimize different stages in diverse ways, they retain square tiles or tile-derived regions as the underlying spatial execution organization. This leaves open whether the tile-based organization itself is well matched to Gaussian structure, which we examine next.

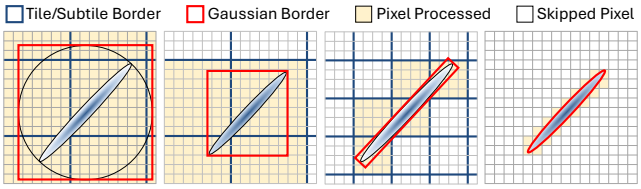
3 Motivation

In this section, we examine the rendering-relevant characteristics of Gaussians and how they are handled under square-tile execution. Because 3DGS explicitly represents a scene with Gaussians, these primitives exhibit several structural properties that directly affect rendering, including spatial placement, projected area, and orientation.

3.1 Spatial Distribution

After training, Gaussians tend to lie on continuous scene surfaces rather than appearing as random primitives. Once projected, neighboring image regions therefore share many Gaussians and often preserve similar relative depth order, forming strong spatial locality. In principle, both the set of contributing Gaussians and their ordering should be reusable across neighboring pixels or fine-grained linear work units.

However, the 16×16 square-tile pipeline breaks this continuity into isolated tile-level subproblems. Moving to an adjacent tile already shifts the work unit by at least 16 pixels, weakening fine-grained locality and ordering coherence at tile boundaries. As a



(a) Original 3DGS (b) AdR-Gaussian (c) GSCore (d) SLGS (ours)

Figure 3: Illustration of how different optimization methods bound and cover the projected area of a Gaussian.

	Orig. 3DGS	AdR-G.	GSCore	SLGS
Boundary cue	Shape only (3σ)	Op.-aware	Shape only (3σ)	Op.-aware
Coverage selection	Tile ∩ AABB	Tile ∩ AABB	Tile ∩ OBB + Subtile	Exact pixel interval
Avg. coverage (px)	1961.49	743.25	505.77	120.31
Avg. coverage (tile)	7.70	2.95	1.99	-

Table 1: Quantitative comparison of projected-area handling across optimization methods.

result, candidate streams and ordering state are reconstructed independently per tile.

3.2 Effective Projected Area

Another important property is that each Gaussian has an effective projected area, the finite screen-space footprint that remains after negligible opacity contributions are pruned. This area is already available after preprocess, rather than being discovered late inside the render loop. Using the projected mean μ_i', inverse conic Q_i, and opacity o_i, all computed during preprocess, together with the same opacity threshold used in original 3DGS, $\alpha_{\min} = 1/255$, the effective projected area of Gaussian g_i can be written as

$$A_i^{\text{eff}} = \left\{ (x, y) \;\middle|\; ([x, y]^\top - \mu_i')^\top Q_i ([x, y]^\top - \mu_i') \le -2 \log\left(\frac{\alpha_{\min}}{o_i}\right) \right\}.$$

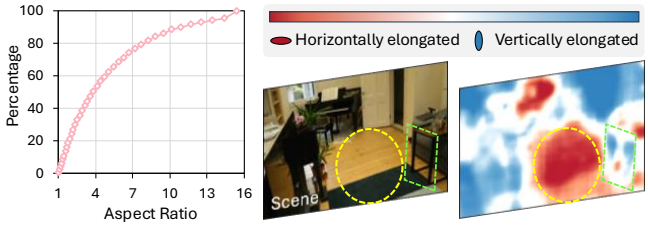
The issue is that square-tile execution does not use this area directly. For a tile $T = [x_0, x_1] \times [y_0, y_1]$, deciding whether Gaussian g_i overlaps the tile reduces to testing

$$\exists (x, y) \in T \text{ such that } (x, y) \in A_i^{\text{eff}}.$$

Because A_i^{eff} is quadratic, this is a two-dimensional constrained overlap test rather than a simple AABB intersection. Tile-based pipelines therefore fall back to conservative proxy admission.

This leaves two sources of inefficiency. First, false positives remain: tiles and pixels that are not actually affected by a Gaussian can still propagate into later stages. Second, the true conflict relation among Gaussians is obscured: under a conservative proxy, two Gaussians may be treated as conflicting simply because they fall into the same tile or subtile, even when their effective projected areas are disjoint. Thus, exact handling of the effective projected area matters not only because it tightens the admitted region, but also because it reveals true overlap more faithfully than proxy-based admission.

Figure 3 and Table 1 illustrate this trend. As prior methods tighten admission through opacity-aware boundaries, OBB selection, and subtile skipping, average tile coverage drops from 7.70 in original 3DGS to 2.95 in AdR-Gaussian and 1.99 in GSCore, while average pixel coverage drops from 1961.49 to 743.25 and 505.77. This shows



(a) Cumulative distribution (b) Tile-level dominant-orientation heatmap

Figure 4: (a) CDF of Gaussian aspect ratios in an example scene. (b) Example view and its corresponding tile-level Gaussian direction heatmap. Red regions denote tiles dominated by horizontally oriented Gaussians, whereas blue regions denote tiles dominated by vertically oriented Gaussians.

that tighter projected-area handling reduces false positives and exposes more true non-overlap. The remaining challenge is to expose this area in a form that the renderer can use directly, rather than only tightening tile admission.

3.3 Anisotropy and Dominant Orientation

After preprocess, each Gaussian already carries its projected shape and direction through Σ_i', or equivalently Q_i, without additional runtime estimation. Let the eigenvalues of Σ_i' be $\lambda_{i,1} \ge \lambda_{i,2}$. Then the aspect ratio of the projected ellipse can be written as

$$\text{aspect ratio}_i = \sqrt{\lambda_{i,1}/\lambda_{i,2}} = a_i^{\text{maj}}/a_i^{\text{min}},$$

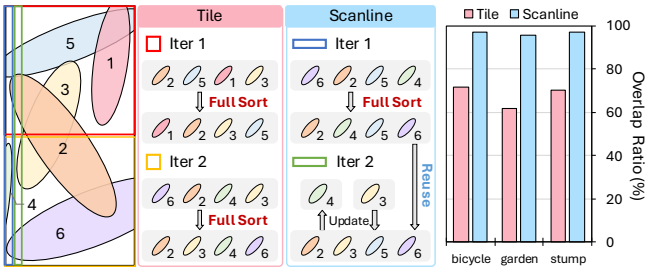
where a_i^{maj} and a_i^{min} denote the major and minor axis lengths of the projected ellipse. The same eigenstructure also determines the dominant orientation of the Gaussian, that is, the direction along which its projected footprint spreads in image space.

In practice, this anisotropy is often substantial. Perspective distortion, thin structures, boundary geometry, and densification during training all contribute to strongly elongated projected Gaussians. In the example scene shown in Figure 4(a), 49.55% of projected Gaussians have an aspect ratio of at least 4, with a long tail toward highly elongated footprints. This illustrates that anisotropic Gaussians can constitute a substantial fraction of the rendering workload.

This directional structure is also spatially correlated. Because nearby Gaussians often lie on the same underlying surface, they tend to share similar dominant orientations. Figure 4 (b) illustrates this clearly: floor regions are dominated by horizontally elongated Gaussians, shown in red regions, whereas vertical structures such as legs are dominated by vertically elongated Gaussians, shown in blue regions. The current square tile, however, is an isotropic work unit and therefore cannot directly reflect this directional structure at the level of execution organization. As a result, Gaussian anisotropy is handled only coarsely under tile-level cooperative execution, while execution opportunities aligned with dominant orientation remain hidden.

4 Observations

The discussion above suggests that square tiles are not the most natural execution unit for Gaussian rendering. A better organization should preserve local ordering continuity, expose effective projected area at low overhead, and reflect the dominant orientation



(a) Sorting reuse in Tile & Scanline (b) Candidate overlap ratio

Figure 5: (a) Illustration of candidate and ordering-state reuse across neighboring work units. (b) Candidate overlap ratio with adjacent rendering units for tiles and scanlines across representative scenes.

of projected Gaussians. From this perspective, scanline-oriented rendering provides a more suitable execution structure.

4.1 Maximizing Spatial Locality

The spatial locality and ordering coherence discussed in Section 3.1 are preserved more directly when neighboring work units change only slightly. Scanlines provide such an organization because they advance by one row or column, whereas square tiles move in coarse tile-sized strides.

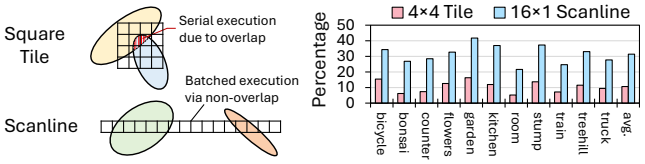
Figure 5 illustrates this effect. Under tile-based rendering, adjacent work units show limited candidate overlap, which often makes independent sorting necessary for each tile. Under scanline-based rendering, most candidates and their relative depth order remain stable across neighboring scanlines, and only a small number of entries need to be inserted or removed. Quantitatively, tile-based rendering preserves only 67.94% average candidate overlap across neighboring units, whereas scanline-based rendering preserves 96.59%. Thus, scanlines expose the locality of Gaussian placement more directly than square tiles.

4.2 Effective-Area Representation and Batching Opportunity

As discussed in Section 3.2, exploiting effective projected area under square tiles requires a two-dimensional constrained overlap test. By contrast, scanlines reduce the same operation to a row-local one-dimensional support test, making effective-area filtering substantially cheaper.

Our oracle-style geometry microbenchmark on an i9-14900K CPU server confirms this difference. Testing the intersection between a conservative bounding box and a tile takes 2.54 ns on average, whereas testing the intersection between a Gaussian's effective projected area and a tile takes 9.05 ns. The scanline case reduces this cost to 5.75 ns. Moreover, because Gaussian support changes smoothly across neighboring scanlines, scanline-level support can be updated incrementally, further reducing the cost to 0.83 ns. Thus, line-shaped units make effective-area handling practical by reducing a two-dimensional overlap problem to a one-dimensional span that can be updated and later materialized at pixel granularity.

This one-dimensional representation also translates into batching opportunity. Figure 6 (a) compares equal-area work units: a 4×4 subtile and a 16-pixel scanline. In a 4×4 subtile, candidates



(a) Opportunity of batching (b) Ratio of non-overlapping Gaussians

Figure 6: (a) Illustration of how projected Gaussian overlap within a work unit affects serial versus batched execution. (b) Non-overlap fraction of adjacent Gaussian pairs within 16-pixel work units. Higher fractions indicate greater batching opportunity.

are confined to a small two-dimensional region, making true overlap likely and often forcing serial execution. A scanline constrains candidates along only one axis, allowing their supports to remain separated along the 16-pixel span. Thus, line-shaped units preserve more true non-overlap for batched execution.

Figure 6 (b) confirms this trend with an oracle-style geometry test over a representative 10-view sample per scene. The adjacent-pair non-overlap fraction is consistently higher for line-shaped units than for 4×4 square subtiles: 27.20% vs. 6.86% on *counter*, 37.94% vs. 15.19% on *garden*, and 37.03% vs. 15.39% on *bicycle*. These results show that line-shaped units not only remove false positives at lower cost, but also expose more scheduler-visible non-overlap for packetized batching.

4.3 Aligning with Dominant Orientation

The anisotropy and dominant orientation discussed in Section 3.3 are also directly related to the choice of rendering unit. When the unit direction matches the dominant local orientation, a Gaussian is less fragmented across units and covers a denser span within each unit, improving lane utilization. Square tiles cannot exploit this alignment because they are isotropic, whereas scanline-oriented units can choose a horizontal or vertical direction per macro-tile.

Taken together, these observations suggest that scanline-oriented rendering offers three advantages over square-tile organization: it preserves locality and ordering coherence more effectively, exposes effective projected area in a lower-overhead and more usable form, and aligns execution units more naturally with Gaussian anisotropy and dominant orientation.

5 Proposal

Motivated by the observations in the previous section, we propose a *scanline-oriented rendering* scheme for 3D Gaussian Splatting and its corresponding in-GPU accelerator, *SLGS*. Instead of optimizing within square tiles, SLGS exposes Gaussian structure at the execution-unit level by replacing square-tile rendering with scanline execution. It combines scanline-local ordering reuse, recurrence-based support extraction, bitmap-native packetization, and 32-lane front-to-back compositing.

5.1 Scanline Rendering Pipeline

Figure 7 shows the scanline rendering pipeline. The pipeline keeps the GPU-side preprocess but replaces square-tile rendering with scanline-oriented execution in the SLGS HW-tail. It consists of four stages: GPU front-end setup, scanline-local sort and reuse, effective

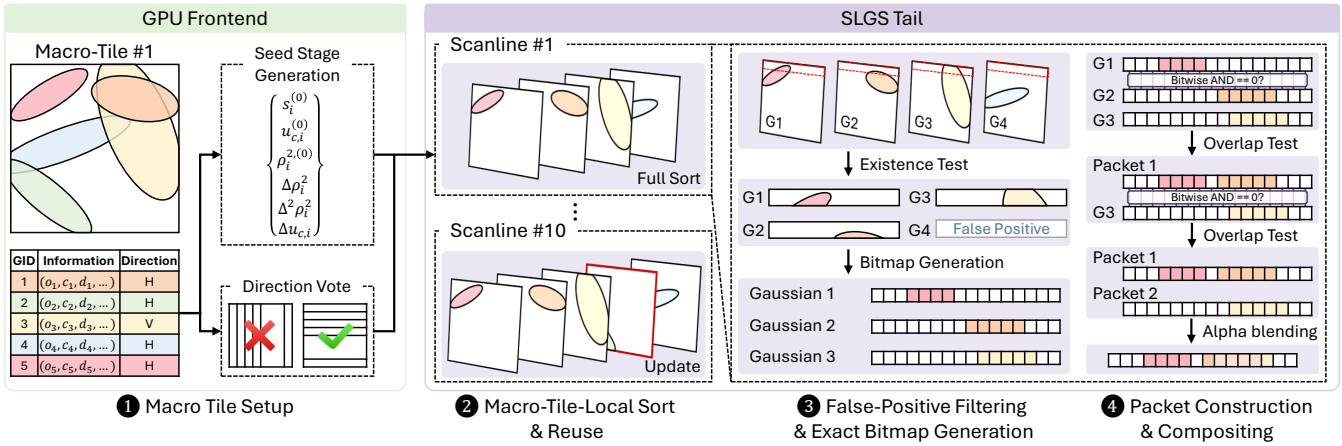


Figure 7: Overview of scanline rendering pipeline.

support extraction, and packetized scanline compositing, which are described below.

1 GPU Front-End & Macro-Tile Setup. The pipeline begins with GPU-side frustum culling and preprocess, as in original 3DGS. For the remaining visible Gaussians, the front-end computes the projected mean μ_i', projected covariance Σ_i', inverse conic Q_i, opacity o_i, depth key d_i, color payload c_i, and related attributes. The image plane is then partitioned into 32×32 macro-tiles, and a Gaussian candidate set is constructed for each macro-tile.

For each macro-tile, the front-end selects either a horizontal or vertical scanline direction using a simple direction vote over the local Gaussians. Each Gaussian casts a one-bit vote based on the principal eigenvector of its projected covariance: it votes horizontal if the absolute horizontal component is larger, and vertical otherwise. The macro-tile direction is selected by majority vote. The selected direction defines a scanline-local coordinate system (u, v), where u is the along-scanline axis and v is the cross-scanline axis. For a horizontal macro-tile, $(u, v) = (x, y)$; for a vertical macro-tile, $(u, v) = (y, x)$.

The front-end then generates the recurrence seed used by the HW-tail for scanline-local support extraction. For each candidate Gaussian, the seed is initialized at the first active scanline $s_i^{(0)}$ and contains

$$\mathcal{R}_i = \left(s_i^{(0)}, u_{c,i}^{(0)}, \rho_i^{2,(0)}, \Delta u_{c,i}, \Delta\rho_i^{2,(0)}, \Delta^2\rho_i^2\right).$$

Here, $u_{c,i}^{(0)}$ and $\rho_i^{2,(0)}$ denote the center and squared half-width of the Gaussian's scanline-local active pixel span at the first active scanline. $\Delta u_{c,i}$ is the per-scanline change of the span center, $\Delta\rho_i^{2,(0)}$ is the initial first difference of the squared half-width, and $\Delta^2\rho_i^2$ is its constant second difference. These seed values are derived once from the scanline-aligned projected conic and allow the HW-tail to advance support state across scanlines using simple recurrence updates.

After this stage, the HW-tail receives, for each macro-tile, the selected scanline direction, the Gaussian candidate stream, the depth key of each candidate, the scanline-support recurrence seed, and Gaussian payloads such as color and opacity.

2 Macro-Tile-Local Sort and Reuse. Using this input, the HW-tail maintains a macro-tile-local ordered stream that is initialized from the first active scanline and then updated as execution advances across scanlines. If the current scanline is the first scanline of the macro-tile, the HW-tail extracts the subset of candidates whose scanline-level active range includes that line, fully sorts them by depth key, and partitions the resulting ordered stream into bounded-size buckets. Here, a bucket is a small contiguous segment of the depth-ordered stream, used as the update granularity for scanline-to-scanline reuse. This initializes the depth-ordered state for subsequent scanline reuse.

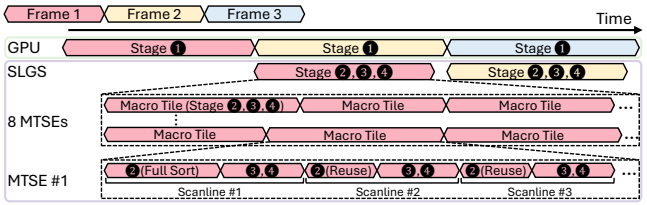


Figure 8: Execution schedule of the SLGS pipeline. GPU Stage 1 overlaps with HW-tail Stages 2–4 across frames, while eight MTSEs process macro-tile tasks in parallel with first-scanline full sort and subsequent scanline-level reuse.

For later scanlines, entry events insert newly active candidates into their target buckets, while exit events invalidate candidates that no longer have support. Each entry event carries the candidate descriptor, including its Gaussian ID, depth key, payload pointer, and recurrence-state pointer. Since each Gaussian has a fixed stable sort key within the current view, e.g., (d_i, gid_i), the target bucket and bucket-local insertion position can be determined without rebuilding the entire ordered stream. Thus, later scanlines reuse the existing depth-ordered layout while updating only the affected bucket entries and bucket metadata.

As a result, the sorted state initialized from the first scanline is reused across the remaining scanlines of the macro-tile, reducing repeated scanline-level sorting overhead.

3 Scanline-Local False-Positive Filtering and Exact Bitmap Generation.

Once the sorting state for the current scanline is available, the HW-tail tests each Gaussian candidate for nonempty support on

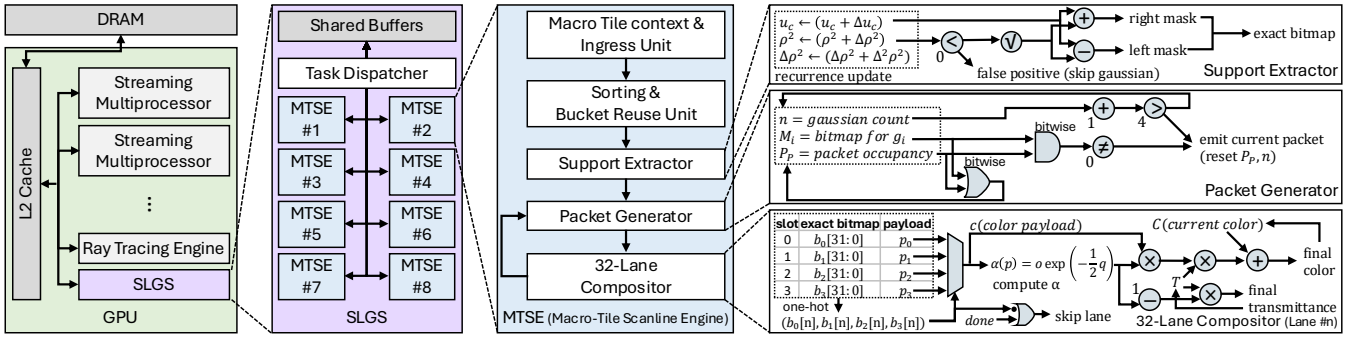


Figure 9: Overview of SLGS Hardware.

that scanline. This step removes scanline-level false positives before the HW-tail computes the per-pixel support bitmap used for scheduling and compositing.

The HW-tail uses the recurrence seed provided by the front-end to maintain the scanline-local support state of each Gaussian. At the first active scanline $s_i^{(0)}$, the state is initialized to $u_{c,i}^{(0)}$, $\rho_i^{2,(0)}$, and $\Delta\rho_i^{2,(0)}$. As the scanline index advances, the HW-tail updates the state using

$$u_{c,i} \leftarrow u_{c,i} + \Delta u_{c,i}, \qquad \rho_i^2 \leftarrow \rho_i^2 + \Delta\rho_i^2, \qquad \Delta\rho_i^2 \leftarrow \Delta\rho_i^2 + \Delta^2\rho_i^2.$$

Here, $u_{c,i}$ denotes the center of the Gaussian's active pixel span on the current scanline, and ρ_i^2 denotes its squared half-width. A Gaussian has nonempty support on the current scanline only if $\rho_i^2 \geq 0$. Candidates with $\rho_i^2 < 0$ are discarded immediately, avoiding both the extent computation needed to obtain the span endpoints and the subsequent bitmap generation.

For each surviving Gaussian, the HW-tail computes the integer support endpoints

$$u_i^{\mathrm{L}} = \left\lceil u_{c,i} - \sqrt{\rho_i^2} \right\rceil, \qquad u_i^{\mathrm{R}} = \left\lfloor u_{c,i} + \sqrt{\rho_i^2} \right\rfloor.$$

The interval is clipped to the 32-pixel scanline range and converted into a 32-bit exact bitmap M_i. Each bit of M_i indicates whether the corresponding pixel is active in the Gaussian's scanline-local support. At this point, the Gaussian's effective projected area has been materialized as a bitmap-native support representation for downstream scheduling and compositing.

❹ **Packetized Scanline Compositing.** The generated exact bitmaps become the execution primitive used by the downstream scheduler. They also turn the batching opportunity discussed earlier into a concrete scheduling mechanism, which we call packetization. In SLGS, a packet is a fixed-width execution unit that groups non-overlapping Gaussian work on the same scanline and issues it to the compositor together. For the remaining Gaussians, the scheduler forms these packets from the scanline-local ordered stream while preserving front-to-back order. In our design, each packet can contain up to $K = 4$ Gaussians. To keep scheduling cost bounded while preserving order, the scheduler performs in-order greedy packetization: it admits the next candidate into the current packet only when its bitmap does not overlap with the packet occupancy bitmap; otherwise, the current packet is emitted and the same candidate starts the next packet.

Each packet is then sent to a 32-lane compositor. Each packet slot carries a Gaussian payload and its current-scanline bitmap. Because packet construction admits only non-overlapping bitmaps within a packet, at most one packet slot is active for each lane. The four slot bitmap bits therefore act as a one-hot selector for the Gaussian payload consumed by that lane. The compositor activates only lanes whose bitmap bits are set and performs front-to-back compositing on those lanes. The compositing rule itself is identical to that of original 3DGS. The difference is that, instead of 256 pixels in a square tile sharing the same ordered stream, only the 32 pixels on the current scanline are selectively activated according to exact bitmaps. Packet execution repeats scanline by scanline, with lane-level state fed back to later packet scheduling on the current scanline.

Figure 8 summarizes how the four stages are scheduled in time. Stage ❶ runs on the GPU front-end, while Stages ❷–❹ run on the SLGS HW-tail. The two units overlap across frames: the HW-tail processes frame f while the GPU front-end prepares metadata for frame $f + 1$. Within the HW-tail, eight MTSEs process macro-tile tasks in parallel; each MTSE performs a full sort for the first scanline and reuses the sorted state for later scanlines before Stage ❸ support extraction and Stage ❹ packetized compositing.

5.2 SLGS Hardware Architecture

Figure 9 illustrates the overall organization of the SLGS hardware. The SLGS hardware is organized as a GPU-integrated HW-tail that consumes the macro-tile candidate stream produced by the GPU front-end. In our design point, the HW-tail contains eight Macro-Tile Scanline Engines (MTSEs), and a dispatcher dynamically assigns ready macro-tiles to the available MTSEs. Each macro-tile is mapped to one MTSE and processed as 32 sequential scanlines of size 32×1 or 1×32, depending on the selected direction. SLGS operates as a persistent, L2-connected HW-tail with double-buffered frame storage, allowing GPU preprocessing and HW-tail execution to overlap across consecutive frames. Frame-level ready/completion state and backpressure coordinate buffer ownership, while macro-tile-level dispatch avoids per-Gaussian dispatch overhead.

1. Input Contract and Macro-Tile Context. At the input boundary, each MTSE receives a macro-tile candidate stream whose Gaussian entries include a depth key, projected screen-space state, scanline-scheduling metadata, and the recurrence metadata required for scanline-local support evaluation. The ingress block initializes the context of the current macro-tile and separates each

Gaussian into scheduling metadata and shading payload. The descriptor buffer stores depth keys, recurrence-related fields, scanline-support state, scanline-update metadata, and validity bits, while payload SRAM stores color, opacity, projected mean, and conic parameters. Macro-tile-context SRAM maintains the selected scanline direction, current scanline index, ordered-stream pointer, scheduler-window pointer, payload SRAM base pointer, and pixel-state SRAM pointer.

2. Sorting State & Bucket Reuse Unit. The sorting block performs a full sort over the first-scanline active candidate stream, writes the result into an ordered-entry SRAM organized as bounded-size buckets, and initializes the corresponding bucket-state metadata. From the second scanline onward, full sorting is not repeated. Instead, the update engine consumes the scanline-update metadata for the current scanline and decodes each update into its target bucket ID, update direction, and candidate descriptor. For an entry update, the corresponding descriptor is inserted into the bucket-local ordered entry storage according to the stable sort key (d_i, gid_i). For an exit update, the matching bucket entry is invalidated or removed. Since each Gaussian has a fixed depth key within the current view, these updates preserve the bucket-local ordering rule without rebuilding the entire ordered stream. In parallel, entry and exit updates targeting the same bucket are reduced into a net signed delta, accumulated in a small bucket-delta scratchpad, and committed only for touched buckets. The delta path updates bucket occupancy metadata, while the bucket-local ordered entry storage maintains the actual descriptors consumed by the scheduler.

3. Exact Support Extractor. Candidates from the ordered stream are then passed to the support extraction engine. This block first checks whether each Gaussian has support on the current scanline using the current scanline index together with the recurrence metadata, and immediately drops Gaussians that do not. For surviving Gaussians, the recurrence datapath updates the current-scanline support state and, when needed, a dedicated square-root unit computes the support extent to obtain the left and right boundaries. The bitmap generator then uses these boundaries together with the scanline range to construct a 32-bit exact bitmap. At this point, each active descriptor entry contains a depth key, a current-scanline bitmap, a payload pointer, a recurrence-state pointer, and a validity bit.

4. Bitmap Scheduling, Packetization, and Compositing. The scheduler directly uses the exact bitmap M_i as its execution primitive and greedily packs non-overlapping Gaussians into packets of up to $K = 4$. Specifically, the packetizer maintains a packet occupancy bitmap P_p and slot count, and admits a candidate into the current packet only if

$$M_i \,\&\, P_p = 0, \qquad \text{slot_count} < 4.$$

When both conditions hold, the packet state is updated as

$$P_p \leftarrow P_p \mid M_i,$$

and the slot count is incremented. Otherwise, the current packet is closed and emitted, and the same candidate is reconsidered as the first entry of the next packet.

The resulting packet is forwarded to the 32-lane compositor. Each packet slot carries a payload pointer identifying the Gaussian and a bitmap indicating which lanes of the current scanline are active for that Gaussian. Pixel-state SRAM stores the accumulated color C_x, the current transmittance T_x, and a done bit for each lane. The compositor evaluates the Gaussian response and opacity contribution only on lanes whose bitmap bits are set and updates C_x and T_x according to the same front-to-back compositing rule used in original 3DGS. After packet execution, the updated done mask of the current scanline is generated and written back as lane-level state for later scheduling decisions.

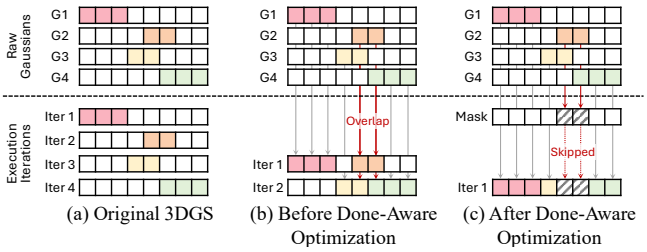


Figure 10: Illustration of done-aware scheduling. (a) Original 3DGS processes Gaussians sequentially. (b) SLGS packetizes non-overlapping Gaussians. (c) Done-aware scheduling masks out early-terminated lanes, removes unnecessary Gaussian work, and enables more compact packetization.

5.3 Done-Aware Optimizations

As packet execution proceeds, some lanes reach the transmittance threshold and no longer require further Gaussian contributions. This lane-level termination is the same early-termination principle used in original 3DGS, but SLGS exposes it to the scheduler rather than consuming it only inside the compositor. As shown in Figure 10 (a), original 3DGS processes Gaussians strictly in sequence. Scanline packetization reduces the number of iterations by grouping non-overlapping Gaussians, as shown in Figure 10 (b). With done-aware scheduling, SLGS further masks out lanes that have already terminated and repacks only the remaining active work, enabling a more compact schedule as shown in Figure 10 (c).

Concretely, let M_i denote the exact bitmap of Gaussian g_i on the current scanline, and let D denote the current done mask. The scheduler forms an effective bitmap

$$\widetilde{M}_i = M_i \wedge \neg D.$$

If $\widetilde{M}_i = 0$, then g_i contributes only to lanes that have already terminated and can be discarded before execution. Otherwise, the scheduler uses $\widetilde{M}_i$, rather than the raw bitmap M_i, for packet construction. When g_i is admitted into a packet, $\widetilde{M}_i$ is also stored as the packet-slot execution bitmap. Thus, the bitmap used for conflict checking is the same bitmap consumed by the 32-lane compositor. As a result, Gaussians that appeared conflicting under raw bitmaps may become non-overlapping after terminated lanes are masked out, allowing them to be placed into the same packet without violating the one-hot lane-selection property.

This optimization is enabled by SLGS's bitmap-native support representation. Because each Gaussian already carries an exact scanline bitmap, the scheduler can directly combine it with the done mask to eliminate unnecessary Gaussian execution and increase packet density for the remaining active lanes.

Scene	 bicycle	 bonsai	 counter	 flowers	 garden	 kitchen	 room	 stump	 train	 treehill	 truck
Resolution(px)	1280×850	1280×853	1280×853	1280×843	1280×829	1280×853	1280×852	1280×848	1280×711	1280×840	1280×713
Total Gaussians	4,870,740	1,057,532	1,034,936	2,902,836	4,184,582	1,565,339	1,262,371	4,152,615	1,073,660	3,000,957	2,056,858

Table 2: Evaluated scenes with example images, resolutions, and total Gaussian counts. All scenes use 30K-iteration checkpoints.

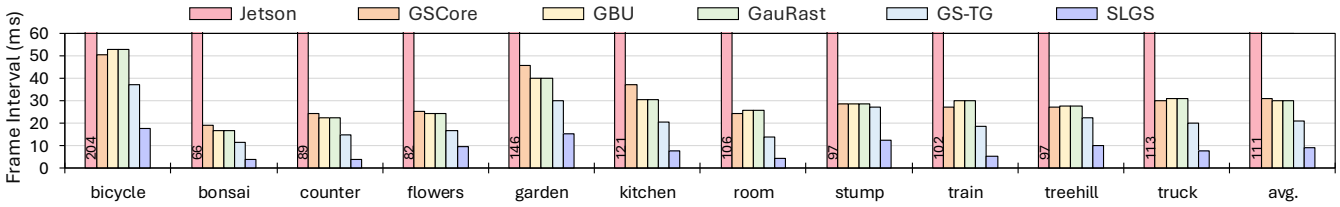


Figure 11: Steady-state full-forward frame time comparison across hardware platforms. Jetson bars exceeding 60 ms are clipped for readability. Numeric labels indicate the actual frame time; lower is better.

5.4 Image Quality Preservation

The proposed scanline-oriented rendering preserves the per-pixel compositing semantics of original 3DGS: each live pixel observes the same depth-ordered sequence of contributing Gaussians under the same opacity and transmittance thresholds. Sort reuse does not change this order because depth keys are fixed within a view and scanline updates only activate or invalidate entries. Exact support extraction removes only candidates with no current-scanline contribution above the same opacity threshold ($\alpha_{\min} = 1/255$).

Packetization is also order-preserving because the packetizer scans candidates in depth order and emits the current packet before reconsidering any conflicting candidate. Gaussians in the same packet have disjoint execution bitmaps and therefore affect disjoint lanes. The execution bitmap is M_i in the normal case and $\widetilde{M}_i = M_i \wedge \neg D$ under done-aware scheduling, where D contains only lanes already past the original early-termination threshold. Since this same bitmap is used for both packet construction and compositor lane selection, done-aware scheduling removes only terminated-lane work without violating the one-hot lane-selection property. These scheduling transformations preserve per-pixel compositing order for the retained contributions.

6 Evaluation

6.1 Methodology

In this section, we evaluate the performance and hardware efficiency of SLGS using direct measurements for the GPU front-end and a cycle-level simulator for the HW-tail. The GPU front-end latency is measured directly on a Jetson Orin NX and includes both the original 3DGS preprocessing operations and the SLGS-specific macro-tile candidate construction, direction voting, and recurrence-seed generation.

The SLGS simulator then takes the post-preprocess state as input and models the HW-tail behavior including macro-tile-local sorting-state reuse, scanline-local support evaluation, bitmap-based packetization, and 32-lane compositing. In this way, we evaluate the proposed scanline-oriented 3DGS forward pipeline under a pipelined execution model in which the GPU front-end and HW-tail operate concurrently across frames. For hardware cost evaluation, we use synthesis results obtained with *Synopsys Design Compiler Version O-2018.06-SP4 for linux64* [3], targeting the *SAED 14nm library* [22]. The clock frequency for SLGS is set to *1 GHz*.

The evaluation workload consists of 11 scenes [2, 13]. The name, example image, resolution, and Gaussian count of each scene are summarized in Table 2. All scenes are evaluated at a fixed image width of 1280 pixels, and all models are tested using checkpoints trained for 30K iterations and sufficiently converged.

We evaluate the following six design points:

- **Jetson [1, 25]:** Original 3DGS full-forward execution on Jetson Orin NX.
- **GSCore [15]:** Prior full-forward accelerator with tile-based candidate admission, sorting, and rendering optimizations.
- **GBU [37]:** Prior render-stage accelerator for Gaussian blending, with preprocessing and sorting left on the GPU path.
- **GauRast [17]:** GPU-rasterizer enhancement that accelerates Gaussian rasterization while leaving preprocessing and sorting on CUDA cores.
- **GS-TG [11]:** Tile-grouping accelerator that shares sorting across a 64×64 group while retaining 16×16 rasterization through per-Gaussian membership bitmasks.
- **SLGS:** Proposed scanline-oriented in-GPU accelerator with direction-aligned scanlines and joint sorting/rendering reorganization.

All accelerator cycle counts are converted to latency at a common frequency of 1 GHz. For a fair comparison, all designs use the same scene set, image resolution, post-preprocess Gaussian streams, opacity threshold, and front-to-back compositing rule as the original 3DGS execution on Jetson. For GSCore, GBU, GauRast, and GS-TG, we implement dedicated simulators configured to reproduce their reported architectural dataflows; for SLGS, Table 3 further verifies that the proposed scheduling changes preserve rendering quality.

6.2 Full-Forward and Stage-Wise Performance

Figure 11 compares the steady-state full-forward frame time of Jetson, GSCore, GBU, GauRast, GS-TG, and SLGS across all scenes. Before analyzing performance, we confirm that SLGS preserves

Scene	Original	SLGS	ΔPSNR	Scene	Original	SLGS	ΔPSNR
bicycle	21.53	21.53	0.00	kitchen	32.26	32.26	0.00
bonsai	31.27	31.29	-0.02	room	35.49	35.50	-0.01
counter	29.89	29.89	0.00	stump	29.88	29.88	0.00
flowers	22.28	22.27	+0.01	train	24.03	24.02	+0.01
garden	26.71	26.71	0.00	treehill	21.83	21.83	0.00
truck	25.88	25.87	+0.01				

Table 3: Rendering quality comparison between original 3DGS and SLGS. PSNR is reported in dB, and ΔPSNR = PSNR(original 3DGS, GT) − PSNR(SLGS, GT). Positive values indicate lower SLGS PSNR.

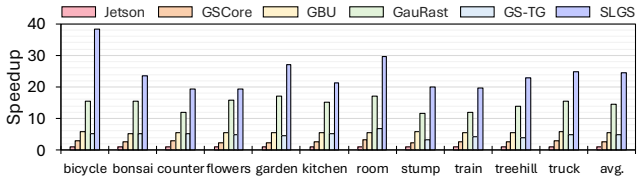


Figure 12: Render stage speedup normalized to the Jetson baseline; higher is better.

rendering semantics: it uses the same front-to-back compositing equation and opacity threshold as original 3DGS. As shown in Table 3, SLGS matches the original renderer across all scenes, with PSNR differences within 0.02 dB, consistent with floating-point rounding and implementation-level arithmetic differences. SLGS achieves an average speedup of 12.43× over original 3DGS, 3.44× over GSCore, 3.35× over both GBU and GauRast, and 2.36× over GS-TG. This shows that scanline-oriented execution reduces the cost of the full-forward pipeline, rather than improving only a single stage in isolation.

The relatively small gap between the full-forward frame intervals of GSCore and GBU is also informative. GBU reduces blending cost effectively, but preprocessing and sorting remain on the critical path, whereas GSCore accelerates a broader portion of the full-forward pipeline. GBU and GauRast likewise show nearly identical full-forward frame intervals because both primarily accelerate rendering while leaving the same GPU-side preprocessing and sorting path. GS-TG further reduces sorting through tile grouping and therefore outperforms these render-stage-focused designs, but still incurs group-union candidate processing, per-tile membership management, and 16 × 16 tile rasterization. In contrast, SLGS jointly reorganizes the original dominant stages—sorting and rendering—around exact, direction-aligned scanline support. Under our pipelined execution model, SLGS substantially reduces both stages and shifts the remaining bottleneck toward GPU-side preprocessing.

Figure 12 isolates the render stage, which is generally the heaviest part of the 3DGS forward pipeline. SLGS achieves an average 24.35× speedup over Jetson, corresponding to 9.23×, 4.38×, 1.67×, and 5.18× higher render-stage throughput than GSCore, GBU, GauRast, and GS-TG, respectively.

GSCore reduces false positives through OBB-based admission and subtile skipping, but these remain proxy-granular and do not restrict execution to exact pixel-level support. GBU improves blending through IRSS, but its fixed-row datapath does not expose support before the render loop, precluding exact-bitmap packetization, done-aware repacking, and direction alignment.

Hierarchy	Config.	Tech.	Freq. (GHz)	SRAM (KB)	Area (mm^2)	Power (mW)
Jetson Orin NX [1, 25]	1024 cores	8 nm	0.918	–	200	25000
GSCore [15]	64 VR cores	28 nm	1.0	272	3.950	870
GBU-Standalone [37]	8 Row PEs	28 nm	1.0	63	1.780	780
GauRast [17]	16 PEs	28 nm	1.0	–	2.434	1700
GS-TG [11]	4 cores	28 nm	1.0	336	3.984	1063
SLGS-Standalone	**8 MTSEs**	**14 nm**	**1.0**	**90**	**2.118**	**688**
Tile & ingress	–	–	–	–	0.002	0.90
Sorting & bucket reuse	–	–	–	–	0.348	130.34
Support extractor	–	–	–	–	0.012	2.41
Packet generator	–	–	–	–	0.008	4.89
32-lane compositor	–	–	–	–	1.736	545.90
Others	–	–	–	–	0.013	3.18

Table 4: Reported hardware design points and SLGS block-level area/power breakdown.

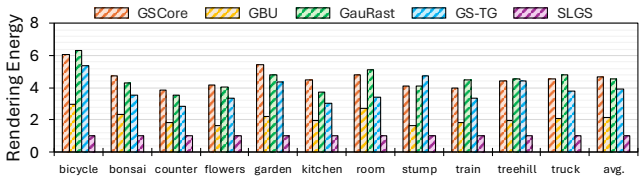


Figure 13: Accelerator energy consumption per frame under steady-state operation, after technology normalization to 28 nm and 1 GHz. Each scene is normalized to SLGS; lower is better.

GauRast achieves the strongest prior render-stage result (14.59×) by mapping Gaussian evaluation and accumulation onto an enhanced GPU rasterizer, but does not reorganize execution around exact support and cross-Gaussian non-overlap. Its much larger render-stage speedup than GBU's (14.59× vs. 5.56×), despite their similar full-forward intervals, confirms that their end-to-end performance is limited by the shared GPU-side preprocessing and sorting path. GS-TG primarily reduces sorting, while its render stage retains 16 × 16 tile execution and tile-granular membership. SLGS instead combines exact-support filtering, non-overlap packetization, direction alignment, and termination-aware repacking, achieving 24.35× render-stage speedup.

Taken together, Figures 11 and 12 show that SLGS improves performance by jointly reducing sorting and rendering work rather than optimizing one stage in isolation. The full-forward gain reflects scanline-local sorting reuse, while the 24.35× render-stage speedup results from exact-support filtering, packetized execution, direction alignment, and done-aware scheduling within a unified execution organization.

6.3 Area and Energy Efficiency

Table 4 summarizes the reported hardware design points of prior accelerators and SLGS. GauRast reports the footprint and power of an enhanced GPU rasterizer, whereas GSCore, GBU, GS-TG, and SLGS report standalone accelerator organizations. Because these designs differ in technology node, precision, and integration scope, their raw area and power values serve as reference design points rather than direct iso-node efficiency comparisons. At its design point, SLGS uses 90 KB of SRAM, occupies 2.118 mm^2, and consumes 688 mW at 1 GHz.

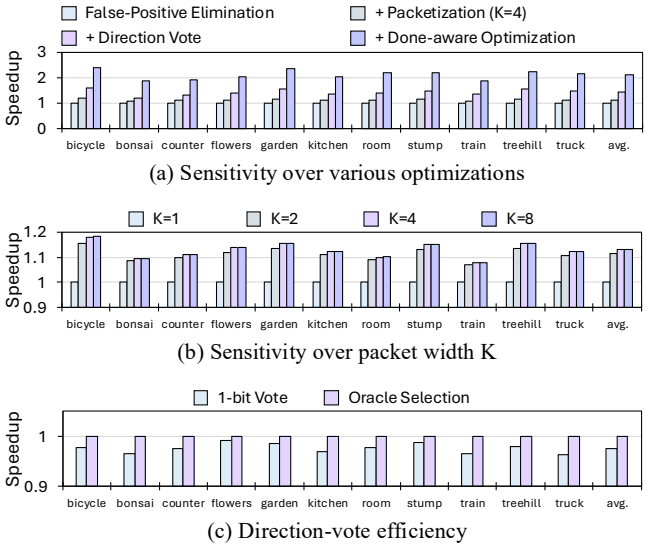


Figure 14: Ablation and sensitivity studies: (a) incremental render-stage speedup, (b) packet-width sensitivity, and (c) 1-bit direction vote versus oracle selection.

Although the 32-lane compositor dominates the SLGS hardware cost, this is primarily a throughput-width choice rather than an intrinsic cost of scanline execution. Post-synthesis analysis shows that the SLGS-specific bitmap selection and $K = 4$ packet-control logic account for only 2.72% of the compositor area and 1.18% of its power; over 97% comes from conventional parallel blending and pixel-state logic.

To reduce technology-node and frequency effects, we normalize all accelerators to 28 nm and 1 GHz using DeepScaleTool [30], and compute per-frame accelerator energy consumption from the normalized power and scene-specific execution time. Figure 13 reports this energy normalized to SLGS for each scene. On average, GSCore, GBU, GauRast, and GS-TG consume 4.65×, 2.11×, 4.55×, and 3.90× the rendering energy of SLGS, respectively. Thus, prior accelerators require 2.11–4.65× SLGS's energy per frame, confirming that SLGS retains a substantial efficiency advantage after technology normalization.

This advantage results from reducing work before compositing through exact-support filtering and converting the exposed non-overlap and termination information into useful lane activity through packetized and done-aware scheduling.

6.4 Ablation and Case Studies

Figure 14 (a) shows the cumulative render-stage gains of the SLGS components. Relative to the preceding configuration, packetization improves performance by 1.13×, direction voting by 1.27×, and done-aware scheduling by 1.48×, yielding a cumulative 2.12× speedup over the effective-area-only baseline. These mechanisms address distinct inefficiencies: serialized non-overlapping work, orientation mismatch, and conflicts involving terminated lanes. In particular, because conventional pixel-level early termination is already applied to every configuration, the final 1.48× gain comes from propagating terminated-lane state to the scheduler, removing such conflicts, and repacking the remaining live work rather than merely enabling early termination.

Orientation split	5:5	6:4	7:3	8:2	9:1	10:0
Macro-tile share	19.09%	29.97%	23.93%	17.42%	9.13%	0.47%
Vote / Fixed	1.018×	1.103×	1.324×	1.634×	1.943×	2.165×

Table 5: Macro-tile distribution and vote-based speedup across orientation-balance bins.

Figure 14 (b) evaluates packet width K. Increasing K from 1 to 2 provides a clear benefit, whereas increasing it from 4 to 8 yields less than 0.2% additional improvement. Since wider packets increase scheduler state and conflict-checking cost without exposing meaningful additional parallelism, we select $K = 4$.

Figure 14 (c) compares the 1-bit direction vote with an oracle that evaluates both horizontal and vertical execution for each macro-tile. The vote remains within 4% of the oracle for every scene and within 2.4% on average, capturing most of the available direction-selection benefit without evaluating both alternatives.

Direction-vote robustness. Table 5 summarizes 55 stress-test views across all 11 scenes. The distribution exhibits clear local directional structure: only 19.09% of macro-tiles fall into the near-balanced 5:5 bin, whereas 50.95% exhibit a 7:3 or stronger directional majority. For each bin, the fixed-direction baseline applies the same direction to every selected macro-tile. Vote-based selection improves performance in every bin, from 1.018× at 5:5 to 2.165× at 10:0, with no view-level regression across the 55 stress tests. Thus, even when the orientation is maximally ambiguous, voting does not reduce performance, while its benefit grows as the local directional tendency becomes clearer.

GPU-integration robustness. Using Jetson Orin NX as the GPU integration target, we evaluate memory-system interference under concurrent GPU and SLGS execution by running GPU preprocessing with 26.59 GB/s of synthetic L2 traffic, matching SLGS's highest observed demand of 26.41 GB/s. Across 15 views from three scenes, preprocessing latency increases by only 2.20%. SLGS consumes 9.44% of the measured L2 bandwidth on average and 18.62% at peak; even under a conservative all-miss assumption, its traffic requires at most 35.8% of the measured LPDDR bandwidth. These results show that persistent GPU–SLGS overlap introduces only modest memory-system interference while leaving substantial bandwidth headroom.

7 Conclusion

This paper presented SLGS, a scanline-oriented rendering scheme and in-GPU accelerator for efficient 3D Gaussian Splatting. By reorganizing the forward pipeline around direction-aligned scanlines, SLGS better matches Gaussian structure and enables ordering-state reuse, low-overhead effective-area handling, and packetized execution of non-overlapping Gaussian work. The resulting support-aligned execution model substantially improves performance over original 3DGS and prior accelerators, showing that scanline-oriented rendering is a more suitable organization for Gaussian rendering than conventional square-tile pipelines.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments. This work was supported by the Samsung Electronics Co., Ltd. (IO250717-13275-01), and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2025-02263080, Development of PIM Memory Technology for a High-Speed Interface-Based Programmable Computing Architecture). Won Woo Ro is the corresponding author.

References

[1] [n. d.]. NVIDIA Jetson Orin NX 16 GB. https://www.techpowerup.com/gpu-specs/jetson-orin-nx-16-gb.c4086.

[2] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. 2022. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5470–5479.

[3] Synopsys Design Compiler. 2016. Synopsys design compiler. *Pages/default. aspx* (2016).

[4] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Linning Xu, Zhilin Pei, Hengjie Li, et al. 2025. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. In *Proceedings of the Computer Vision and Pattern Recognition Conference*. 26652–26662.

[5] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance fields without neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5501–5510.

[6] Antoine Guédon and Vincent Lepetit. 2024. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5354–5363.

[7] Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. 2025. Speedy-splat: Fast 3d gaussian splatting with sparse pixels and sparse primitives. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 21537–21546.

[8] Peter Hedman, Pratul P Srinivasan, Ben Mildenhall, Jonathan T Barron, and Paul Debevec. 2021. Baking neural radiance fields for real-time view synthesis. In *Proceedings of the IEEE/CVF international conference on computer vision*. 5875–5884.

[9] Qiqi Hou, Randall Rauwendaal, Zifeng Li, Hoang Le, Farzad Farhadzadeh, Fatih Porikli, Alexei Bourd, and Amir Said. 2024. Sort-free gaussian splatting via weighted sum rendering. *arXiv preprint arXiv:2410.18931* (2024).

[10] Shengyu Huang, Zan Gojcic, Zian Wang, Francis Williams, Yoni Kasten, Sanja Fidler, Konrad Schindler, and Or Litany. 2023. Neural lidar fields for novel view synthesis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 18236–18246.

[11] Joongho Jo and Jongsun Park. 2025. GS-TG: 3D Gaussian Splatting Accelerator with Tile Grouping for Reducing Redundant Sorting while Preserving Rasterization Efficiency. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[12] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, George Drettakis, et al. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.* 42, 4 (2023), 139–1.

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

[14] Junseo Lee, Jaisung Kim, Junyong Park, and Jaewoong Sim. 2025. Vr-pipe: Streamlining hardware graphics pipeline for volume rendering. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 217–230.

[15] Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. 2024. 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*. 497–511.

[16] Jiwon Lee, Yunjae Lee, Youngeun Kwon, and Minsoo Rhu. 2024. Characterization and analysis of the 3D Gaussian Splatting rendering pipeline. *IEEE Computer Architecture Letters* 24, 1 (2024), 13–16.

[17] Sixu Li, Ben Keller, Yingyan Celine Lin, and Brucek Khailany. 2025. GauRast: Enhancing GPU triangle rasterizers to accelerate 3D Gaussian splatting. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[18] Tianye Li, Mira Slavcheva, Michael Zollhoefer, Simon Green, Christoph Lassner, Changil Kim, Tanner Schmidt, Steven Lovegrove, Michael Goesele, Richard Newcombe, et al. 2022. Neural 3d video synthesis from multi-view video. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5521–5531.

[19] Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *Advances in Neural Information Processing Systems* 33 (2020), 15651–15663.

[20] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. 2019. Neural volumes: Learning dynamic renderable volumes from images. *arXiv preprint arXiv:1906.07751* (2019).

[21] Tao Lu, Mulin Yu, Linning Xu, Yuanbo Xiangli, Limin Wang, Dahua Lin, and Bo Dai. 2024. Scaffold-gs: Structured 3d gaussians for view-adaptive rendering. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 20654–20664.

[22] Vazgen Melikyan, Meruzhan Martirosyan, Anush Melikyan, and Gor Piliposyan. 2018. 14nm educational design kit: Capabilities deployment and future. In *Small Systems Simulation Symposium*. 42–55.

[23] Ben Mildenhall, Pratul P Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. 2019. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics (ToG)* 38, 4 (2019), 1–14.

[24] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2021. Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1 (2021), 99–106.

[25] NVIDIA. 2022. Jetson Orin NX Series Datasheet. https://developer.download.nvidia.com/assets/embedded/secure/jetson/orin_nx/docs/Jetson_Orin_NX_DS-10712-001_v0.5.pdf.

[26] Changhun Oh, Seongryong Oh, Jinwoo Hwang, Yoonsung Kim, Hardik Sharma, and Jongse Park. 2026. Neo: Real-Time On-Device 3D Gaussian Splatting with Reuse-and-Update Sorting Acceleration. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1268–1284.

[27] Minnan Pei, Gang Li, Junwen Si, Zeyu Zhu, Zitao Mo, Peisong Wang, Zhuoran Song, Xiaoyao Liang, and Jian Cheng. 2025. Gcc: A 3dgs inference architecture with gaussian-wise and cross-stage conditional processing. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. 1824–1837.

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

[29] Christian Reiser, Rick Szeliski, Dor Verbin, Pratul Srinivasan, Ben Mildenhall, Andreas Geiger, Jon Barron, and Peter Hedman. 2023. Merf: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes. *ACM Transactions on Graphics (ToG)* 42, 4 (2023), 1–12.

[30] Satyabrata Sarangi and Bevan Baas. 2021. DeepScaleTool: A tool for the accurate estimation of technology scaling in the deep-submicron era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. Ieee, 1–5.

[31] Cheng Sun, Min Sun, and Hwann-Tzong Chen. 2022. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5459–5469.

[32] Matthew Tancik, Vincent Casser, Xinchen Yan, Sabeek Pradhan, Ben Mildenhall, Pratul P Srinivasan, Jonathan T Barron, and Henrik Kretzschmar. 2022. Block-nerf: Scalable large scene neural view synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8248–8258.

[33] Qitai Wang, Lue Fan, Yuqi Wang, Yuntao Chen, and Zhaoxiang Zhang. 2024. Freevs: Generative view synthesis on free driving trajectory. *arXiv preprint arXiv:2410.18079* (2024).

[34] Xinzhe Wang, Ran Yi, and Lizhuang Ma. 2024. Adr-gaussian: Accelerating gaussian splatting with adaptive radius. In *SIGGRAPH Asia 2024 Conference Papers*. 1–10.

[35] Zhican Wang, Guanghui He, Lingjun Gao, Dantong Liu, Shell Xu Hu, Chen Zhang, Zhuoran Song, Nicholas Lane, and Hongxiang Fan. 2026. Efficient 3D Gaussian Splatting with Axis-Shared Rasterization and Order-independent Transmittance. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2014–2028.

[36] Mengtian Yang, Yipeng Wang, Chieh-Pu Lo, Xiuhao Zhang, Sirish Oruganti, and Jaydeep P Kulkarni. 2025. GSAcc: Accelerate 3D Gaussian Splatting via Depth Speculation and Gaussian-centric Rasterization. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–7.

[37] Zhifan Ye, Yonggan Fu, Jingqun Zhang, Leshu Li, Yongan Zhang, Sixu Li, Cheng Wan, Chenxi Wan, Chaojian Li, Sreemanth Prathipati, et al. 2025. Gaussian blending unit: An edge gpu plug-in for real-time gaussian-based rendering in ar/vr. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 353–365.

[38] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. 2001. Surface splatting. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 371–378.

[39] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. 2002. Ewa splatting. *IEEE Transactions on Visualization & Computer Graphics* 8, 03 (2002), 223–238.