PipeGS: Unlocking 3DGS Pipeline Parallelism via Hierarchical Reuse and Dynamic Partitioning

Xueling Wang[1], Yuzhou Chen[1], Zhican Wang[1], Pan Zhao[1], Renda Jian[1], Yitian Chen[1], Chen Zhang[1*], Yang Hu[2], Guanghui He[1*]

[1]School of Integrated Circuits, Shanghai Jiao Tong University, [2]Tsinghua University

{wangxueling02,huygens,wang_zhican,panzhao08,jian_rd,kaorimsly,chenzhang.sjtu,guanghui.he}@sjtu.edu.cn

hu_yang@tsinghua.edu.cn

Abstract

3D Gaussian Splatting (3DGS) delivers high quality and speed for 3D scene reconstruction. However, achieving real-time performance on edge platforms remains challenging due to pipeline inefficiencies. Our profiling reveals two major bottlenecks: (1) insufficient pipeline overlap due to the latency gap between sorting and rasterization, and (2) pipeline parallelism constrained by globally serialized preprocessing. To address these challenges, we present PipeGS, a 3DGS accelerator with algorithm–architecture co-design. At the algorithm level, we propose hierarchical Gaussian reuse, which shortens rasterization latency by eliminating redundant computation and reduces 76.2% of pipeline bubbles. To further unlock full pipeline overlap, we introduce dynamic orthogonal partitioning, which breaks global serial dependencies and hides 63% of preprocessing latency. At the hardware level, PipeGS employs a customized layer-wise pipelined architecture that supports concurrent execution across stages. Implemented on a 28nm technology, PipeGS achieves **1.96∼3.34×** on area efficiency, and **1.20∼2.77×** on energy efficiency compared with SOTA 3DGS accelerators.

1 Introduction

Recent advances in neural rendering have greatly improved the efficiency of 3D scene reconstruction [1–5]. 3D Gaussian Splatting (3DGS) [3] has emerged as a promising alternative to Neural Radiance Fields (NeRF) [1], achieving comparable rendering quality while delivering higher speed through its explicit representation. Owing to its efficiency, 3DGS is adopted in real-time applications such as augmented reality (AR), virtual reality (VR), and embodied AI, which demand low-latency rendering on edge platforms.

Despite its strong performance on high-end GPUs, 3DGS remains difficult to deploy on edge devices such as Nvidia Orin NX [6]. The core challenge lies in the pipeline inefficiency across preprocessing, sorting, and rasterization, which exhibit highly heterogeneous workloads. Such heterogeneity leads to severe pipeline underutilization and degraded throughput on resource-constrained platforms.

To improve efficiency, existing works [7–10] have explored two optimization directions. GSCore [7] overlaps sorting and rasterization via hierarchical sorting and subtile skipping, but its global preprocessing stage remains serialized (Fig. 1b). GSAcc [8] leverages temporal reuse of sorting results to parallelize stages. However, it is highly sensitive to camera motion. When the viewpoint changes beyond small deviations, the reused results become invalid, leading to degraded pipeline and performance (Fig. 1c). Consequently, existing designs still fail to achieve a fully overlapped pipeline. Our profiling of the 3DGS pipeline further reveals that the **insufficient pipeline overlap due to latency gap between sorting and rasterization** and **pipeline parallelism constrained by globally serialized preprocessing**, remain the dominant challenges.

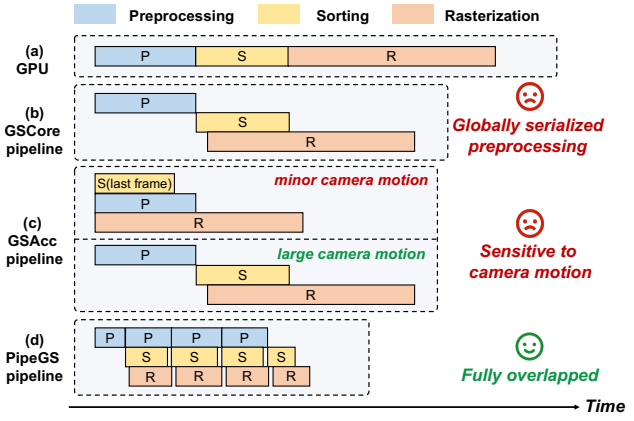

Figure 1: Comparison of rendering pipelines of GPU-based 3DGS, GSCore, GSAcc, and the PipeGS designs.

To tackle these challenges, we propose PipeGS, a customized accelerator for 3DGS with algorithm-architecture co-design. The key insight is restructuring the pipeline through workflow optimization and decoupling global data dependencies to achieve a fully overlapped pipeline as shown in Fig. 1d. In summary, our key contributions are as follows:

- We performed a detailed profiling and analysis of computation workload of 3DGS stages and identified two major bottlenecks: (1) latency gap between sorting and rasterization, and (2) globally serialized preprocessing, which together introduce bubbles and hinder the full overlap of pipeline.
- To realize a fully overlapped pipeline, we first propose *Hierarchical Gaussian Reuse*, which eliminates redundant computations at both pixel and tile levels and significantly shortens rasterization latency. Then, we present *Dynamic Orthogonal Partitioning*, which breaks the global dependency of preprocessing to overlap with sorting and rasterization.

*Corresponding author: Guanghui He, Chen Zhang.

DAC '26, Long Beach, CA, USA

ACM ISBN 978-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804263

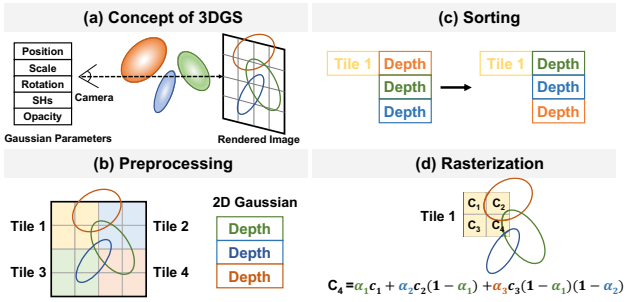

Figure 2: Overview of 3D Gaussian Splatting.

- We design PipeGS, a hardware accelerator that supports the proposed optimized pipeline. Compared with state-of-the-art (SOTA) 3DGS architectures, PipeGS achieves up to 3.34× on area efficiency, and 2.77× on energy efficiency.

2 Background and Motivation

2.1 3D Gaussian Splatting Pipeline

3DGS [3] represents a scene with a collection of anisotropic 3D Gaussians $G_0, G_1, \ldots, G_N$, each parameterized by its position, scale, rotation, spherical harmonic (SH) coefficients, and opacity. The rendering process, visualized in Fig. 2a, unfolds through three sequential stages: *preprocessing*, *sorting*, and *rasterization*.

Preprocessing: The 3D Gaussians are projected onto the 2D image plane. For each screen tile, a list of potentially contributing Gaussians is constructed based on their footprints (Fig. 2b).

Sorting: Within each tile, the associated Gaussians are meticulously sorted by their depth values (computed during preprocessing) to resolve occlusion relationships for correct compositing (Fig. 2c).

Rasterization: This stage computes the final pixel colors. For each pixel, the contribution of every overlapping Gaussian is evaluated. The key operation is calculating the splatting opacity α for a Gaussian-pixel pair, based on Elliptic Weighted Average (EWA) [11]:

$$\alpha_i(p) = \sigma_i e^{-\frac{1}{2}(p-\mu_i)^T(\Sigma_i^{2d})^{-1}(p-\mu_i)} \tag{1}$$

Here, μ_i, σ_i and Σ_i^{2d} denote $Gaussian_i$'s 2D center, opacity and 2D covariance matrix, respectively, and p is the pixel center. The quadratic term $-\frac{1}{2}(p-\mu_i)^T(\Sigma_i^{2d})^{-1}(p-\mu_i)$ represents the spatial contribution of $Gaussian_i$ to pixel p, commonly referred to as the $Power(p)$. The final pixel color C is then obtained by alpha-blending all N contributing Gaussians in sorted order (Fig. 2d):

$$C = \sum_{i \in N} T_i c_i \alpha_i, \tag{2}$$

where c_i is the color from $Gaussian_i$'s SH coefficients, and $T_i = \prod_{j=1}^{i-1}(1-\alpha_j)$ is the transmittance representing the cumulative visibility from all preceding Gaussians.

2.2 Challenges

Despite its compelling performance, a detailed profiling of the 3DGS pipeline reveals fundamental inefficiencies that hinder its deployment on resource-constrained edge devices. Our analysis, summarized in Fig. 3, uncovers two critical bottlenecks.

2.2.1 Pipeline Profiling and Bottleneck Analysis. Our profiling on RTX 3090 GPU shows that *rasterization* consumes approximately 70% of total rendering time (Fig. 3a, above). Per-tile analysis further reveals a staggering **7.48**× average ratio of rasterization-to-sorting time (Fig. 3a, middle), exposing the large 69% pipeline bubbles in the execution timeline (Fig.3a, bottom). Furthermore, the pipeline's dataflow enforces *global serialization*: preprocessing must finish before any sorting or rasterization can begin, creating approximately 15% latency at the pipeline front (Fig. 3a, bottom). These observations point to two root causes: (1) insufficient pipeline overlap due to latency gap between sorting and rasterization, and (2) the pipeline parallelism constrained by globally serialized preprocessing.

Challenge I: Insufficient Pipeline Overlap Due to Latency Gap Between Sorting and Rasterization. The latency gap arises because rasterization carries substantial redundant computation at both pixel and tile levels, making it much heavier than sorting.

- **Per-Pixel Redundancy:** The computation of *Power* in Eq. 1 is performed independently for each pixel, ignoring the strong spatial coherence among adjacent pixels. Within a tile, consecutive pixels calculate the same Gaussian's contribution using almost identical intermediate values, leading to substantial redundant computation. As shown in Fig. 3b (above), computing the contribution of the green Gaussian to the blue tile requires two steps: (1) computing pixel–Gaussian center offsets (orange dashed box); (2) calculating spatial contribution *Power* to each pixel using its 2D covariance matrix (yellow dashed box). However, neighboring pixels satisfy simple relations such as $p_{x1} = p_{x0} + dx$. This coherence naturally propagates to the Gaussian's power across pixels, yet the original algorithm still treats each pixel independently and recomputes the quadratic form for every pixel.
- **Per-Tile Redundancy:** Rasterization is organized tile-by-tile. A Gaussian that overlaps multiple tiles (e.g., the green Gaussian spanning four tiles) has its parameters loaded and computed repeatedly for each tile. Statistically, over 70% of Gaussians overlap two or more tiles, exposing substantial reuse opportunities (Fig. 3b, middle). However, the tile-wise processing breaks spatial continuity, causing the inherent coherence of Gaussian contributions to be entirely wasted.

This redundancy not only wastes computational resources but directly creates the orders-of-magnitude latency gap between the lightweight sorting and the heavyweight rasterization stages.

Key Insight I: Our initial countermeasure is Hierarchical Gaussian Reuse (HGR), which reduces redundancy by cross-pixel and cross-tile reuse. This is achieved by restructuring the workflow to offload a portion of computation to preprocessing. *This shift improves the pipeline overlap, however, unveils a second bottleneck.*

Challenge II: Pipeline Parallelism Constrained by Globally Serialized Preprocessing. With HGR, the preprocessing stage now handles both Gaussian projection and precomputation, increasing the workload and exacerbating the global serialization bottleneck. Specifically, constructing the Gaussian list for each tile requires *complete traversal* of all Gaussians in the view frustum. Consequently, even if a tile's list (e.g., the red tile overlapping with only two Gaussians 1 and 2) is ready early, its sorting and rasterization are stalled until preprocessing for the *entire scene* finishes (Fig. 3c,

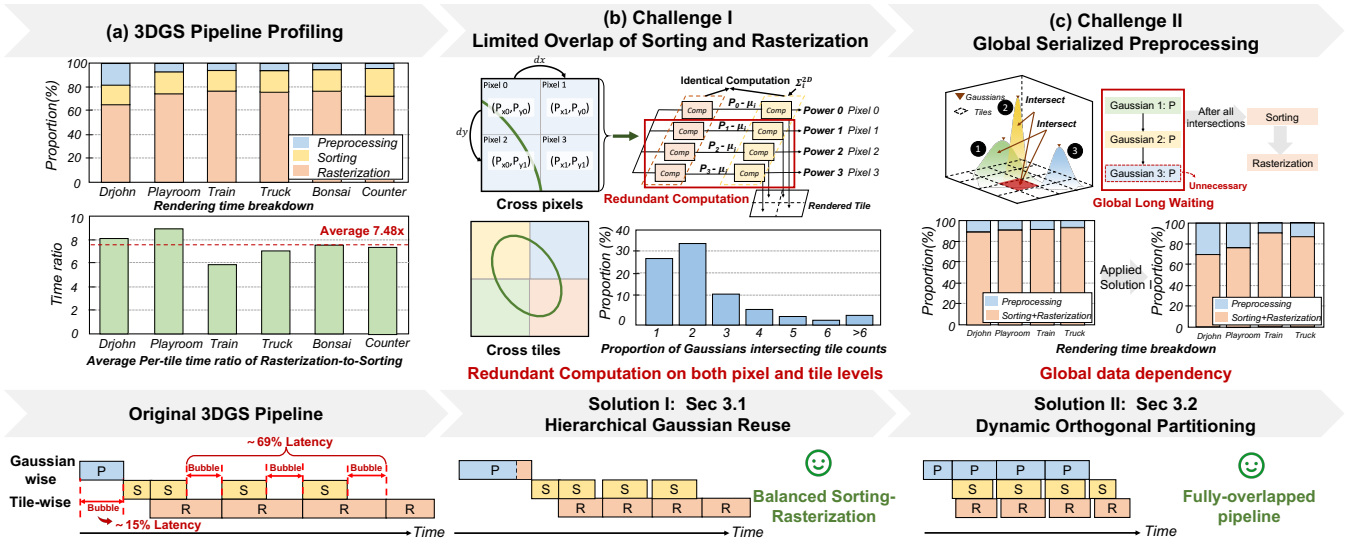

Figure 3: Overview of profiling and two key challenges on 3DGS rendering pipeline and our corresponding solutions.

above). This creates a classic *producer-consumer* problem with a single, slow producer, causing pipeline bubbles.

This increased workload further limits pipeline parallelism. After applying Solution I, the relative serial preprocessing time increases by up to 20% in the Drjohn scene (Fig. 3c, middle). Thus, a localized optimization is insufficient; a systemic solution is required to break global dependency and enable full pipeline overlap.

Key Insight II: Our ultimate solution to achieve a balanced, high-throughput pipeline is Dynamic Orthogonal Partitioning (DOP), which strategically partitions the scene to parallelize preprocessing and overlap it with downstream stages, thereby enabling a ***fully overlapped pipeline***.

3 Proposed Pipeline Optimization Approach

3.1 Hierarchical Gaussian Reuse (HGR)

To tackle the insufficient pipeline overlap originating from redundant computations in rasterization (Challenge I), we propose Hierarchical Gaussian Reuse (HGR), which eliminates redundancy at both pixel and tile levels through two synergistic methods.

Differential Power Computation (DPC). To address redundant per-pixel computation in tile-wise rasterization, we exploit the differential form of the Gaussian power. First, we decompose pixel coordinate into an *anchor pixel* (the top-left pixel of a tile) $p_0 = (p_{x0}, p_{y0})^T$ and an *offset* $(dx, dy)^T$, i.e., $p = (p_x, p_y)^T = p_0 + (dx, dy)^T$, so that the Gaussian-pixel coordinate difference in Eq. 1 becomes $(p - \mu_i)^T = (\Delta x_0 + d_x,\ \Delta y_0 + d_y)^T$, where $\Delta x_0 = p_{x0} - \mu_{ix}$ and $\Delta y_0 = p_{y0} - \mu_{iy}$ are constants for a given Gaussian and a tile. By expanding the quadratic component in Eq 1, the power term can be rewritten as

$$\text{Power}(d_x, d_y) = \underbrace{A + C\,d_y + E\,d_y^2}_{P_{init}(d_y)} + \underbrace{(B + F\,d_y)}_{\beta(d_y)}\,d_x + D\,d_x^2. \quad (3)$$

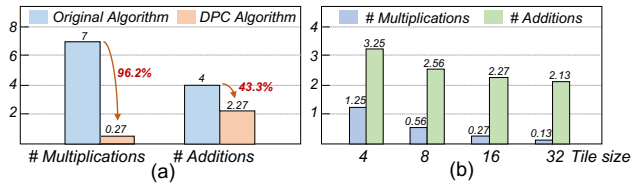

Figure 4: Computation reduction of DPC. (a) Per-pixel numbers of multiplications and additions reduction. (b) Per-pixel arithmetic cost of DPC under different tile sizes.

The power computation needs three steps (Fig. 5a): (1) We first compute Gaussian coefficients A–F at the anchor pixel p_0, performed once per Gaussian-tile pair. (2) We then calculate the *Row Initialization* terms, $P_{\text{init}}(d_y)$ and $\beta(d_y)$ for each row. (3) Finally, along each row, per-pixel powers are obtained by incremental updates from $P_{\text{init}}(d_y)$, avoiding recomputation of the quadratic form.

This differential approach drastically reduces arithmetic complexity. For a specific 16×16 tile, the original per-pixel power computation requires 7 multiplications and 4 additions. With DPC, only 4 multiplications and 6 additions are needed for coefficient computations, 4 multiplications and 4 additions for row initializations, and 2 additions for every pixel. As a result, the number of multiplications is reduced by 96.2% and total arithmetic operations are reduced by 76.9% per pixel (Fig. 4a). Moreover, as the tile size grows, the cost of coefficient computation can be further amortized (Fig. 4b).

Tile-Shared Power Precomputation (TSPP). While DPC optimizes within a tile, TSPP tackles redundancy across tiles. A key observation is that a Gaussian's contribution α on the image is sorting-independent. Therefore, its computation can be decoupled from the rasterization stage and moved to preprocessing.

TSPP restructures the workflow from a tile-centric to a Gaussian-centric process to compute Gaussian's contribution α. For each Gaussian, we first determine not only which tiles a Gaussian overlaps but also a fine-grained 4-bit bitmap that identifies the active

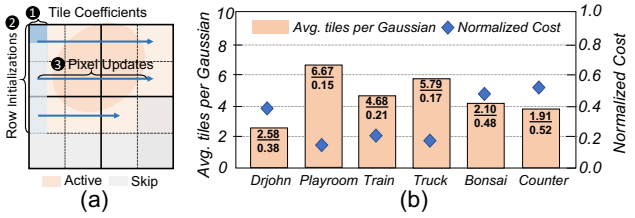

Figure 5: Illustration and analysis of HGR (DPC+TSPP). (a) Three-step Power Computation. (b) Average overlapped tiles and corresponding coefficient cost reduction with HGR.

regions within each intersected tile. We then *precompute and cache the alphas for all active pixels in intersected tiles* during preprocessing. This ensures that each Gaussian's parameters and computational intermediates are loaded and processed only once, regardless of how many tiles it covers.

Based on DPC, we take the anchor pixel of the top-left tile as the reference for the entire Gaussian footprint. The power coefficients A–F are computed once at this anchor pixel and reused across all intersecting tiles. Each tile's bitmap defines the active spans, restricting computation to only the active pixels and eliminating unnecessary work (Fig. 5a).

The synergy of DPC and TSPP is powerful. As Fig. 5b shows, each Gaussian overlaps 2.6 to 6.7 tiles on average. TSPP exploits this, reducing the computation cost to 0.15 to 0.52× of the baseline. More importantly, by shifting alpha computation into preprocessing, the workflow becomes more balanced and pipeline bubbles are reduced.

3.2 Dynamic Orthogonal Partitioning (DOP)

While DPC and TSPP collectively mitigate the rasterization bottleneck, the latter shifts substantial computational workload to the preprocessing stage, exacerbating the global serialization issue (Fig. 3c). To address this and break the data dependency of global preprocessing, we introduce Dynamic Orthogonal Partitioning (DOP) to unlock full pipeline parallelism.

The core idea is to decompose the global preprocessing task into smaller, independent units that can be processed in parallel with the sorting and rasterization of other units. We achieve this by partitioning the 3D scene into *orthogonal layers*. During training, we discretize the scene into an $N \times N \times N$ grid. At render time, given the camera view direction, we first determine the *principal axis* (the world axis most aligned with the view direction). Then, the scene is partitioned along this axis, grouping grid cells along the other two axes into 2D layers (Fig. 6a). This creates a set of layers that can be preprocessed independently (Fig. 6b).

A critical challenge arises when Gaussians straddle multiple layers, as this can violate depth ordering and corrupt the final composition. To ensure correctness, we introduce *Cross-Layer Dependency Regularization (CLDR)* during model training. For each Gaussian G_i, we measure its distance to the nearest layer boundary $d_{boundary}$ along the *principal axis*, and compute its effective standard deviation along the principal axis $\mathbf{n}$ as

$$\sigma_n^2 = \mathbf{n}^T \mathbf{R}\, \mathrm{diag}(\mathbf{s}^2)\, \mathbf{R}^T \mathbf{n}. \tag{4}$$

where $\mathbf{s}$ and $\mathbf{R}$ denote Gaussian's scale and rotation. Then, we define an allowable extent $\sigma_{target} = \gamma\, d_{boundary}$ with $\gamma \in (0, 1)$, and

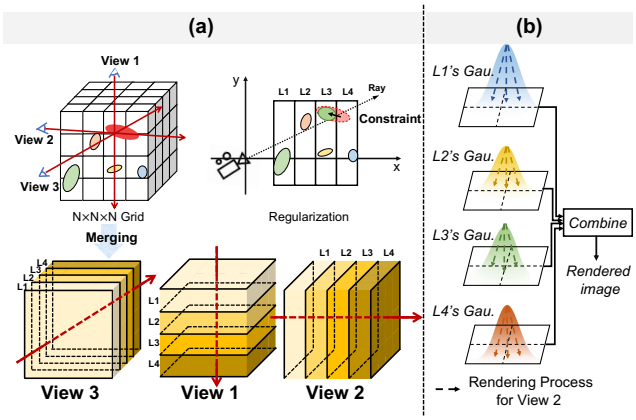

Figure 6: Dynamic Orthogonal Partitioning. (a) Orthogonal partitioning and regularization used to assign Gaussians. (b) Layer-wise rendering and combination into the final image.

penalize only cases where σ_n exceeds this margin:

$$\mathcal{L}_{boundary} = \frac{1}{N} \sum_{i=1}^{N} \left[\max\left(0,\ \sigma_n^{(i)} - \sigma_{target}^{(i)}\right) \right]^2. \tag{5}$$

This constraint suppresses cross-layer spillover, keeping Gaussians aligned with their assigned layers while preserving details. As shown in Fig. 6(a), the previous misaligned red Gaussian spanning layer 3 and layer 4 is regularized into the layer-consistent green one. The total training objective is

$$\mathcal{L}_{total} = \mathcal{L}_{origin} + \lambda_{cldr}\, L_{boundary} \tag{6}$$

where λ_{cldr} controls the strength of the regularization. By effectively constraining Gaussians to layers, DOP transforms the global serial dependency into fine-grained, localized dependencies, enabling preprocessing to overlap with downstream stages.

4 Proposed PipeGS Architecture

We present the PipeGS architecture, a customized accelerator that implements the optimized pipeline described in Section 3. As illustrated in Fig. 7, the design comprises four units: Dynamic Orthogonal Partitioning Unit (DOPU), Preprocessing Unit (PU), Sorting Unit (SU), and Rasterization Unit (RU). These units operate in a coordinated, pipelined manner to maximize throughput. To sustain fully overlapped execution, we employ double buffers for Gaussian parameters and dedicated on-chip buffers for intermediate results.

4.1 Inter-layer Pipeline Scheduling

The rendering process begins with the DOPU, which partitions Gaussians into multiple layers based on the camera viewpoint using a normalization unit and a three-way multiplexer. This enables a coarse-grained, inter-layer pipeline that overlaps preprocessing with the subsequent stages.

As shown in the execution timeline in Fig. 7, the pipeline processes adjacent layers concurrently: while the PU preprocesses layer $i+1$, the SU and RU concurrently sort and rasterize layer i, effectively hiding preprocessing latency. To ensure correct compositing across layers, intermediate transmittance T and color are

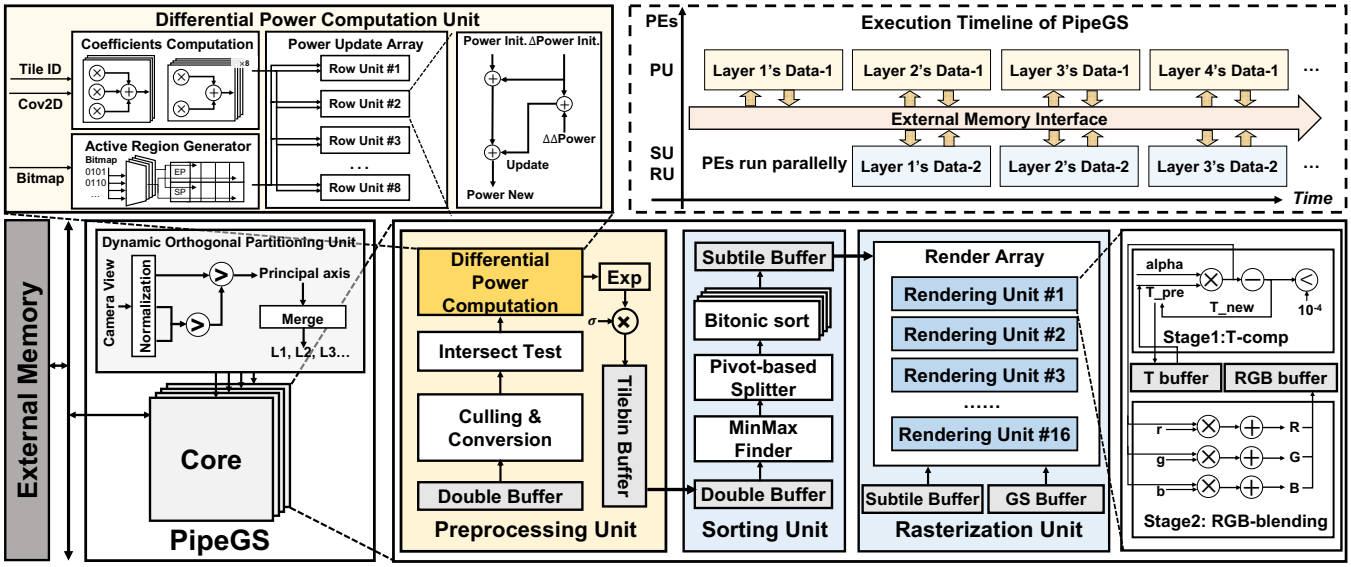

Figure 7: Overview of PipeGS architecture.

buffered, enabling progressive image generation and early termination. Within each layer, Gaussians pass through a fine-grained, three-stage intra-layer pipeline (preprocessing, sorting, rasterization). In steady state, the system maintains concurrent execution across layers and stages, maximizing hardware utilization.

4.2 Intra-layer Pipeline

4.2.1 Preprocessing Unit (PU). The PU first culls invisible Gaussians and projects the remaining ones to generate 2D features. The Intersect Test Unit then determines the set of tiles intersected by each Gaussian and generates a 4-bit bitmap describing the active regions for every overlapped tile. Following the HGR, the *Differential Power Computation Unit* immediately computes the powers for all intersecting Gaussian–tile pairs.

For each Gaussian, the coefficients A–F are computed once based on the anchor pixel of its top-left tile and reused across all intersecting tiles. Each tile's bitmap specifies the active spans (Start Point, SP; End Point, EP), which are dispatched to eight Row Units. These units compute power values for active pixels using DPC, leveraging the shared coefficients to avoid redundant work. The final alpha value for the pixel is obtained by applying exponential function to the power and scaling by the Gaussian's opacity σ.

4.2.2 Sorting Unit (SU). The SU performs depth-based Gaussian sorting using a two-stage hierarchical scheme. The first stage uses a MinMax Finder to determine the depth range of Gaussians, followed by a Pivot-based Splitter that partitions them into several depth groups. In the second stage, a 16-channel Bitonic Sorter is applied within each group to produce the final depth-ordered Gaussian list.

4.2.3 Rasterization Unit (RU). The RU comprises 16 parallel Rendering Units to process pixels concurrently. It leverages precomputed alpha values from the PU, avoiding on-the-fly redundant computation. To ensure correct cross-layer blending, two dedicated on-chip buffers (T buffer and RGB buffer) store the accumulated T and RGB for each pixel, respectively. For each new layer, the RU reads the previous T and RGB from the buffers and blends the current layer's Gaussian color contributions using Eq. 2.

5 Evaluation

5.1 Experimental Setup

We implement the RTL design of PipeGS and synthesize it using Synopsys Design Compiler with a commercial 28nm technology. On-chip memory area and power are estimated with CACTI-7.0 [12], and off-chip memory performance is evaluated using a cycle-level simulator with DRAM timing modeled by Ramulator [13].

For algorithm evaluations, the Gaussian grid resolution is set to 4×4×4 for outdoor scenes and 8×8×8 for indoor scenes; the hyperparameters λ_{cldr} and γ are set to 0.05 and 0.9, respectively. We conduct experiments on three datasets: Mip-NeRF360 [14], Tanks&Temples [15] and Deep Blending [16], covering both indoor and outdoor scenes. For hardware evaluations, we compare PipeGS against four dedicated 3DGS accelerators: GSCore [7], GS-TG [10], STREAMINGGS [17], and GSAcc [8]. For a fair comparison, we adopt the same evaluation settings as described in prior works.

5.2 Evaluation on Proposed Algorithm

Table 1 compares rendering quality between the original algorithm and PipeGS. Overall, our method preserves the fidelity of 3DGS well. Across all evaluated scenes, the PSNR degradation remains within **1%** on average and notably, in several scenes, PipeGS even achieves higher PSNR than the original implementation.

To further analyze the contribution of our refinements, we also evaluate rendering quality without CLDR. As shown in Table 1, removing CLDR causes noticeable PSNR drops in multiple scenes, indicating accumulated cross-layer compositing errors. With CLDR enabled, the accuracy is effectively restored and in some cases, matches or even surpasses the original results.

Table 1: Comparison of rendering quality (PSNR ↑). Bold indicates the best performance.

Algorithm	Tanks&Temples		Deep Blending		Mip-NeRF360						Average
	Train	Truck	Playroom	Drjohnson	Bonsai	Counter	Bicycle	Stump	Flowers	Treehill	
Original	**22.16**	25.41	**29.97**	**29.36**	**32.39**	**29.08**	**25.25**	**26.63**	21.41	**22.55**	26.42
PipeGS(w/o CLDR)	21.64	24.62	28.16	28.55	28.97	26.39	24.13	25.66	21.57	22.21	25.19
PipeGS	**22.16**	**25.49**	29.87	**29.36**	31.82	28.76	24.81	25.91	**21.62**	22.33	26.21

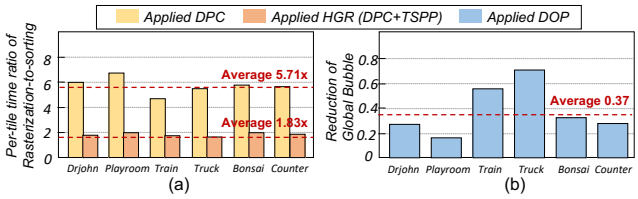

Figure 8: Improvements in pipeline overlap and parallelism from DPC, HGR(DPC+TSPP), and DOP.

Table 2: Hardware configuration.

Module	Configuration	Area [mm^2]	Power [W]
DOPU	1	0.001	0.004
PU	4	1.695	0.466
SU	4	0.065	0.068
RU	4	0.472	0.171
Buffers	428KB	1.038	0.236
Total		**3.283**	**0.948**

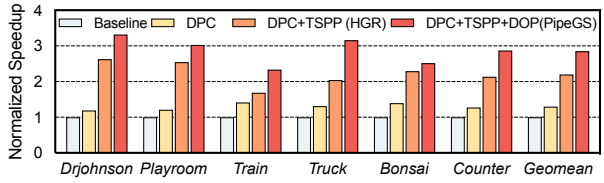

Figure 9: Speedup from HGR (DPC and TSPP) and the final design (DPC+TSPP+DOP). The baseline refers to our reproduction of GSCore based on its published design.

5.3 Evaluation on Proposed Accelerator

5.3.1 Pipeline Efficiency Improvement. HGR substantially improves the overlap between sorting and rasterization: the time ratio of rasterization-to-sorting drops from 7.48× (Baseline) to 5.71× with DPC and further to a well-balanced 1.83× with DPC+TSPP (Fig. 8a), reducing pipeline bubbles by 76.2% in total.

Beyond balancing these two stages, DOP breaks the global preprocessing serialization. As shown in Fig. 8b, the preprocessing delay is reduced to 37% of the original pipeline, allowing most of the preprocessing work to be overlapped with sorting and rasterization, thereby eliminating the global pipeline stall.

5.3.2 End-to-end Speedup. Fig. 9 shows the end-to-end speedup from our three techniques: DPC, TSPP, and DOP. We evaluate four variants: Baseline, DPC, DPC+TSPP, and DPC+TSPP+DOP. Compared with the baseline, HGR (DPC+TSPP) achieves up to 2.33× speedup by improving the overlap between sorting and rasterization stages. DOP further contributes an additional 1.22× speedup by hiding preprocessing latency. Overall, PipeGS yields up to 2.85× end-to-end throughput improvement over the baseline.

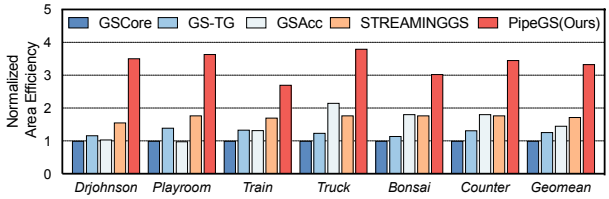

Figure 10: Normalized area efficiency comparison results of PipeGS with SOTA 3DGS accelerators.

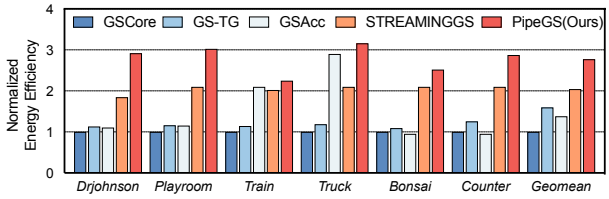

Figure 11: Normalized energy efficiency comparison results of PipeGS with SOTA 3DGS accelerators.

5.4 Comparison with SOTA 3DGS Accelerators

Table 2 presents the implementation details of PipeGS. At 1GHz clock, our design has a total area of 3.283 mm^2 with a power consumption of 0.948 W. We conduct area and energy efficiency comparison of PipeGS with leading 3DGS accelerators [7, 8, 10, 17]. Leveraging HGR and DOP, PipeGS achieves 1.96 ~ 3.34× on area efficiency in Fig. 10, and 1.20 ~ 2.77× on energy efficiency in Fig. 11. These improvements stem from two sources: (1) HGR substantially cuts redundant computations per Gaussian and leads to more balanced pipeline; and (2) DOP unlocks global pipeline parallelism.

6 Conclusion

We present PipeGS, an algorithm-hardware co-design for 3DGS acceleration. Our approach combines Hierarchical Gaussian Reuse to eliminate redundant rasterization with Dynamic Orthogonal Partitioning to decouple global preprocessing, supported by a customized layer-wise pipelined architecture. Evaluations demonstrate that PipeGS achieves up to 3.34× area efficiency, and 2.77× energy efficiency over SOTA 3DGS accelerators.

Acknowledgments

This work was supported by the New Generation Artificial Intelligence-National Science and Technology Major Project under Grant 2025ZD0122400 and National Natural Science Foundation of China under Grant U25B2057.

References

[1] 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.

[2] Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. 2021. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF international conference on computer vision*. 5855–5864.

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

[4] 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.

[5] Ben Fei, Jingyi Xu, Rui Zhang, Qingyuan Zhou, Weidong Yang, and Ying He. 2024. 3d gaussian splatting as new era: A survey. *IEEE Transactions on Visualization and Computer Graphics* (2024).

[6] NVIDIA. 2025. NVIDIA Jetson Orin NX. https://developer.nvidia.com/jetson-orin-nx. Accessed: 2025-11-13.

[7] 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.

[8] 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.

[9] Zhican Wang, Guanghui He, Dantong Liu, Lingjun Gao, Shell Xu Hu, Chen Zhang, Zhuoran Song, Nicholas Lane, Wayne Luk, and Hongxiang Fan. 2025. Accelerating 3D Gaussian Splatting with Neural Sorting and Axis-Oriented Rasterization. *arXiv preprint arXiv:2506.07069* (2025).

[10] 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.

[11] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. 2001. Ewa volume splatting. In *Proceedings Visualization, 2001. VIS'01.* IEEE, 29–538.

[12] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[13] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[14] 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.

[15] 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.

[16] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. 2018. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)* 37, 6 (2018), 1–15.

[17] Chenqi Zhang, Yu Feng, Jieru Zhao, Guangda Liu, Wenchao Ding, Chentao Wu, and Minyi Guo. 2025. STREAMINGGS: Voxel-Based Streaming 3D Gaussian Splatting with Memory Optimization and Architectural Support. *arXiv preprint arXiv:2506.09070* (2025).