Neo: Real-Time On-Device 3D Gaussian Splatting with Reuse-and-Update Sorting Acceleration

Changhun Oh
KAIST
Daejeon, Republic of Korea
choh@casys.kaist.ac.kr

Seongryong Oh
KAIST
Daejeon, Republic of Korea
sroh@casys.kaist.ac.kr

Jinwoo Hwang
KAIST
Daejeon, Republic of Korea
jwhwang@casys.kaist.ac.kr

Yoonsung Kim
KAIST
Daejeon, Republic of Korea
yskim@casys.kaist.ac.kr

Hardik Sharma
Meta
Sunnyvale, CA, USA
hardiksharma@meta.com

Jongse Park
KAIST
Daejeon, Republic of Korea
jspark@casys.kaist.ac.kr

Abstract

While 3D Gaussian Splatting (3DGS) has emerged as a promising technique for immersive AR/VR experiences, its practical adoption critically depends on whether real-time rendering can be achieved on resource-constrained devices, such as Meta Ray-Ban Display and Google Android XR Glasses. However, existing solutions struggle to achieve high frame rates, especially for high-resolution rendering. Our analysis identifies the sorting stage in the 3DGS rendering pipeline as the major bottleneck due to its high memory bandwidth demand. This paper presents Neo, which introduces a reuse-and-update sorting algorithm that exploits temporal redundancy in Gaussian ordering across consecutive frames and devises a hardware accelerator optimized for this algorithm. By efficiently tracking and updating Gaussian depth ordering instead of re-sorting from scratch, Neo significantly reduces redundant computations and memory bandwidth pressure. Experimental results show that Neo achieves up to 12.4× and 5.5× higher throughput than state-of-the-art edge GPU and ASIC solution, respectively, while reducing DRAM traffic by 94.6% and 81.4%. These improvements make high-quality and low-latency on-device 3D rendering more practical.

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

Keywords: Domain Specific Architecture (DSA), Accelerator, Neural Rendering, 3D Gaussian Splatting (3DGS)

ACM Reference Format:
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 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790192

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '26, Pittsburgh, PA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790192

1 Introduction

The success of Generative AI in text generation [1, 8, 12, 29, 46, 52, 60, 76, 77, 95, 106] has demonstrated its potential, paving the way for researchers to explore its next step: virtual world generation [4, 7, 15, 32, 33, 40, 44, 62, 84, 86]. Modern augmented and virtual reality (AR/VR) devices such as Meta Ray-Ban Display [67], Google Android XR Glasses [27], and Apple Vision Pro [2] exemplify the growing demand for immersive environments, where human interactions unfold in dynamic, computer-generated worlds, necessitating high-fidelity content and scalable virtual world creation.

View synthesis, which generates images of a scene from novel viewpoints, is crucial for enabling virtual experiences. Recently, 3D Gaussian Splatting (3DGS) has emerged as an effective technique that balances rendering quality and performance, while providing a low-cost solution for immersive applications. To ensure a smooth user experience [24, 101], such applications demand high-resolution rendering with ultra-low latency (e.g., 7–15 ms [3]). While cloud processing may appear attractive, it introduces critical delays [5, 6], thereby necessitating on-device 3DGS rendering.

However, this on-device requirement creates tension with the high computational demands of real-time view synthesis. For instance, industry-leading automotive GPUs such as the Jetson Orin [38, 74] and even the state-of-the-art 3DGS-optimized accelerator [50] deliver only around 60 FPS at HD resolution, offering lower frame rates when targeting the per-eye high-resolution commonly used in AR/VR, such as 2K or 4K per eye [2, 34, 68, 82, 93, 97]. This performance gap underscores the need for more efficient on-device solutions.

To better understand the bottlenecks and opportunities, we first conduct a thorough performance characterization of an existing 3DGS accelerator, GSCore [50]. Our analyses suggest that GSCore effectively mitigates bottleneck in

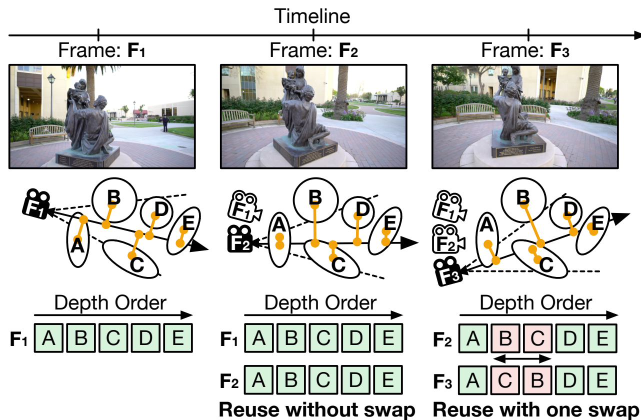

Figure 1. Reuse opportunities in the sorting stage of 3D Gaussian Splatting (3DGS) inference. The figure illustrates how the Gaussian order across three consecutive frames (F1, F2, and F3) exhibits significant temporal similarities. This redundancy enables Neo's reuse-and-update sorting mechanism to minimize redundant sorting computations, improving efficiency in real-time rendering.

rasterization, leaving the sorting stage as a new and dominant bottleneck in the rendering pipeline. Unfortunately, sorting is a well-established operation in modern computing, making it inherently difficult to accelerate through conventional means. In the context of 3DGS, however, frames are rendered sequentially over time, repeatedly performing sorting on largely similar Gaussian orders across consecutive frames. This temporal redundancy presents an opportunity to minimize redundant computations and improve efficiency.

Inspired by this insight, this paper proposes Neo by jointly designing ① reuse-and-update sorting mechanism that exploits temporal redundancy in Gaussian ordering, and ② a hardware-accelerated sorting pipeline for on-device 3DGS rendering. Figure 1 depicts the reuse opportunities that form the basis of the proposed reuse-and-update sorting mechanism. In designing Neo, we identify the following challenges:

- **Challenge 1: High sorting overhead in rendering.** Sorting in 3DGS differs from general-purpose sorting due to the need for reordering of millions of Gaussians, each with depth-dependent visibility. Existing sorting implementations rely on bandwidth-intensive memory accesses, making real-time on-device rendering impractical.
- **Challenge 2: Missed temporal redundancy in sorting.** Re-sorting Gaussians from scratch every frame overlooks the strong temporal locality inherent in 3DGS rendering. Since most Gaussians retain similar depth order across consecutive frames, treating each frame independently leads to highly overlapping computations and excessive DRAM traffic, limiting scalability at high resolutions.

To address the aforementioned challenges, this paper makes the following contributions.

- **Reuse-and-update sorting for temporal redundancy exploitation.** Neo introduces a reuse-and-update sorting mechanism that leverages the temporal redundancy in 3D Gaussian Splatting (3DGS) rendering. Instead of sorting Gaussians from scratch for every frame, Neo efficiently tracks and updates the sorted Gaussian table across consecutive frames. By identifying minimal changes in Gaussian ordering over time, this mechanism significantly reduces redundant sorting computations and memory bandwidth overhead. Moreover, by selectively updating only designated segments of the table, Neo minimizes unnecessary operations, thereby improving overall processing efficiency. As a result, this optimization enables real-time sorting at high resolutions while preserving rendering accuracy, even under frequent viewpoint changes.
- **Accelerating sorting stage for on-device rendering.** Neo implements a hardware-accelerated sorting pipeline optimized for efficient on-device 3DGS rendering. While existing accelerators focus primarily on optimizing rasterization, they overlook the inefficiencies of sorting, resulting in excessive DRAM traffic. Neo addresses this limitation by integrating dedicated sorting hardware that minimizes memory access overhead and accelerates depth-based Gaussian ordering within the rendering pipeline. This hardware adopts a hybrid sorting strategy that integrates both partial and global ordering, effectively performing Gaussian sorting with low computational overhead. By coupling this hardware support with a reuse-and-update sorting mechanism, Neo delivers significant improvements in both latency and throughput, making real-time, high-resolution AR/VR rendering practical.

We evaluate Neo by implementing it as both an in-house simulator and an RTL design to cross-validate functional correctness. Our evaluation uses a representative set of neural rendering workloads, including complex scenes, to assess the impact of temporal redundancy. We synthesize the RTL design using Synopsys Design Compiler with ASAP 7nm library [98] and measure its performance and memory bandwidth usage against the NVIDIA Orin AGX GPU [38] and the state-of-the-art accelerator GSCore [50]. Our results show that Neo achieves up to 12.4× and 5.5× higher throughput compared to the Orin AGX GPU and GSCore, respectively, while reducing end-to-end memory traffic by 94.6% and 81.4%. Furthermore, Neo enables real-time 3DGS rendering at QHD resolutions, achieving an average throughput of 97.7 FPS required for smooth AR/VR experiences. These results demonstrate that Neo effectively overcomes the sorting bottleneck in real-time 3DGS rendering, advancing the realization of generative virtual worlds and enabling truly immersive on-device experiences. A full-system simulator for Neo and the accuracy evaluation code are open-sourced at https://github.com/casys-kaist/Neo.git.

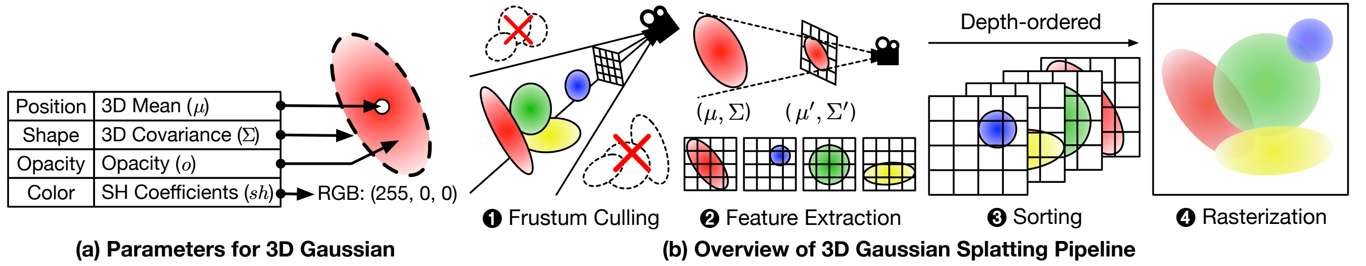

Figure 2. Brief overview of 3D Gaussian Splatting.

2 Background

2.1 Real-Time 3D Rendering in On-Device Systems

Recent advances in augmented and virtual reality (AR/VR) have driven the development of mobile devices and headsets [2, 35, 66, 81, 93] capable of rendering high-fidelity scenes. These on-device platforms support high frame rates and resolution, delivering immersive experiences. For instance, Apple Vision Pro [2] features a combined 23 million pixels, achieving 4K-level resolution. Meanwhile, Meta Quest 3 [66] provides a per-eye resolution of 2064×2208, with both devices supporting refresh rates of up to 90Hz. These stringent specifications are critical for delivering immersive experiences and ensuring user comfort [100, 101]. However, meeting these demands is challenging when rendering high-fidelity scenes given the limited computing resources of on-device systems. A straightforward approach is offloading rendering requests to cloud servers. However, this method suffers from frame drops due to network congestion, which is a critical issue in latency-sensitive rendering applications [5, 6]. These constraints highlight the necessity for on-device AR/VR platforms to integrate on-device rendering capabilities.

2.2 3D Gaussian Splatting (3DGS)

3DGS [39] has emerged as a promising rendering method for synthesizing complex real-world scenes while achieving real-time rendering performance. To render 3D scenes, 3DGS exploits millions of 3D Gaussians, modeled as anisotropic ellipsoids. As illustrated in Figure 2(a), each Gaussian is identified by a radial opacity α, as shown in Equation 1:

$$\alpha(x) = o \cdot e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} \tag{1}$$

where o is an opacity value, $\boldsymbol{\mu}$ is a mean vector, Σ is a 3D covariance matrix. In addition, sh refers to spherical harmonics coefficients, which enable rendering of view-dependent color [28]. These explicit 3D representations are learnable parameters, trained with differentiable rasterization and gradient-based optimization. Given a set of Gaussians, 3DGS employs a α-blending process [83], akin to traditional rendering methods [9] that use explicit 3D representations to render scenes. This innovative approach incorporates the strengths of both traditional scene reconstruction methods [51, 99, 102] and prior neural rendering techniques [23, 69, 71, 85], delivering exceptional rendering quality and performance.

2.3 3D Gaussian Splatting Pipeline

Figure 2(b) denotes the 3DGS pipeline with four main stages: frustum culling, feature extraction, sorting, and rasterization.

❶ **Frustum Culling.** First, the system discards Gaussians that are not visible from the current camera viewpoint and preserves only those required for subsequent stages. This initial filtering step reduces redundant computation for Gaussians outside the camera's field of view.

❷ **Feature Extraction.** With the filtered 3D Gaussians, the system projects each Gaussian onto the camera's image plane and extracts its view-dependent features. This process generates the 2D projected mean and covariance (μ', Σ') from the 3D parameters (μ, Σ) and computes color (c) from spherical-harmonics coefficients (sh) [28].

❸ **Sorting.** After extracting features from 3D Gaussians, the projected 2D Gaussians overlap on the image plane. At this stage, the system sorts them by depth. This ordering is a crucial step for the subsequent rasterization stage, which leverages it to blend overlapping 2D Gaussians by accumulating their pixel colors in a depth-sorted manner.

❹ **Rasterization.** At this stage, the system computes pixel colors by α-blending [83] the depth-sorted Gaussians. Starting from the foremost Gaussian, each Gaussian contributes to the pixel color and accumulates opacity. When the cumulative opacity exceeds a predefined threshold, further processing for that pixel stops, as its color is considered finalized, thereby reducing unnecessary computation.

2.4 3D Gaussian Splatting Acceleration

Tile-based parallelism. For efficient rendering, 3DGS subdivides the image plane into a grid of smaller 2D regions (*tiles*). In the ❸ sorting stage, it duplicates and distributes Gaussians to the intersected tiles, and sorts the assigned Gaussians within each tile. In the ❹ rasterization stage, it performs α-blending on a per-tile basis. This approach enables the system to process only the Gaussians within each tile's boundary, thereby reducing redundant computation. Furthermore, it leverages hardware threads to process multiple tiles in parallel, improving overall rendering performance.

GPU implementations. To leverage this parallelism, 3DGS methods [39, 73, 78] employ GPUs for tile-based sorting and rasterization. They utilize NVIDIA CUB library [75] for sorting and implement custom CUDA kernels [39] to

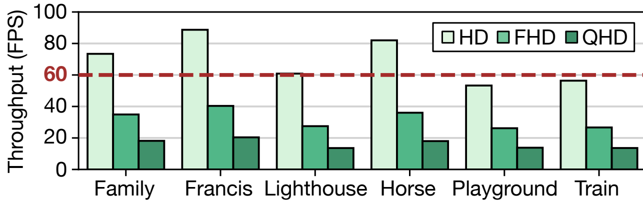

Figure 3. Throughput comparison with different resolutions.

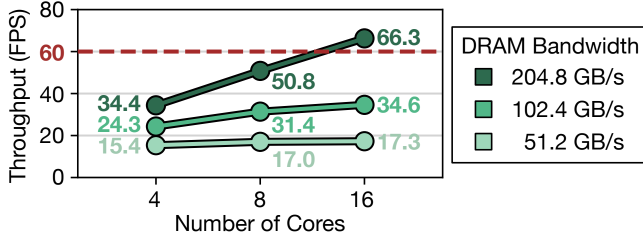

Figure 4. Throughput comparison across core counts and DRAM bandwidth when rendering at QHD resolution. Each colored label denotes the corresponding FPS performance.

support a cumulative α-blending during rasterization. Although these GPU-driven approaches enhance rendering performance through extensive parallel processing, the cumulative α-blending step remains a significant bottleneck, especially in on-device AR/VR environments.

ASIC acceleration: GSCore. To address this challenge, recent work GSCore [50] introduces a pioneering ASIC-based acceleration solution for on-device 3DGS systems. By adopting hierarchical tile-based sorting and subtile-based rasterization, GSCore efficiently processes millions of Gaussians under tight resource constraints. While this specialized co-design significantly outperforms GPU-based solutions, the stringent on-device constraints continue to push resource demands beyond what GSCore approaches can readily handle. In the following section, we analyze how these constraints affect on-device 3DGS performance and define the key research challenge of this work.

3 Motivation

3.1 Challenge in On-Device 3DGS Rendering

Despite GSCore's efforts to accelerate 3DGS rendering, achieving real-time performance under the high frame rate and resolution requirements of modern AR/VR platforms (see Section 2.1) remains a significant challenge. To analyze these characteristics, we selected six scenes from the Tanks and Temples dataset [43]: Family, Francis, Horse, Lighthouse, Playground, and Train, which were captured in real-world outdoor environments and serve as representative benchmarks for evaluating the performance of 3DGS rendering.

Rendering performance under constraints. Figure 3 shows the throughput (FPS) performance of GSCore at three different resolutions: HD (1280×720), FHD (1920×1080), and QHD (2560×1440). Following the rationale described in the

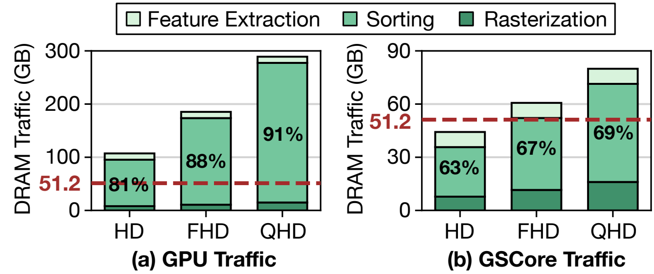

Figure 5. DRAM traffic (GB) required for rendering 60 frames and breakdown of memory bandwidth consumption across 3DGS pipeline stages.

original paper [50], we configure the evaluation system with 4 computing cores and a DRAM bandwidth of 51.2 GB/s. This configuration enables a thorough assessment of GSCore's FPS performance within the tight resource budgets typical of on-device AR/VR environments. The results show that GSCore achieves 66.7 FPS at HD (1280×720), exceeding the conservative service-level objective (SLO) requirement of 60 FPS. However, at higher resolutions GSCore experiences a significant FPS drop, achieving only 31.1 FPS and 15.8 FPS at FHD and QHD, respectively. These performance limitations underscore the need for a robust acceleration solution capable of delivering scalable and responsive 3DGS rendering within resource-constrained on-device systems.

3.2 Performance Characterization of 3DGS

To better understand the throughput performance of 3DGS at high-resolution (QHD), we perform a bottleneck analysis of GSCore by varying two primary system knobs: the number of compute unit cores and available DRAM bandwidth. Figure 4 shows FPS performance against varying core counts (4, 8, and 16 cores) under three distinct DRAM bandwidth conditions (51.2 GB/s, 102.4 GB/s, and 204.8 GB/s).

Implication from cores. First, we focus on the scenario representative of typical edge devices, characterized by limited DRAM bandwidth around 51.2 GB/s, as provided by conventional GSCore setups. Under this bandwidth constraint, increasing the core count from 4 to 16 provides minimal performance improvements; specifically, even a fourfold increase in cores yields only about a 1.12× improvement in FPS, falling significantly short of the targeted 60 FPS SLO. This result indicates that computational scaling alone is insufficient when bandwidth constraints are tight.

Implication from bandwidth. Next, we examine the impact of DRAM bandwidth on the FPS performance of the 3DGS system. At the highest bandwidth (204.8 GB/s), which is a 4× increase over a typical on-device system (51.2 GB/s), the 3DGS system surpasses the 60 FPS SLO, achieving a 3.83× improvement in FPS. These results indicate that high-resolution 3DGS performance is constrained by DRAM bandwidth rather than computational power, identifying memory bandwidth as the primary bottleneck.

Bandwidth breakdown. To further investigate the sources of bandwidth usage, we perform an analysis of bandwidth consumption in high-resolution 3DGS. Figure 5 presents the DRAM traffic (GB) required to render 60 frames and its breakdown for the GPU-based 3DGS and GSCore across different resolutions. In both systems, the results show that the sorting stage dominates bandwidth consumption, accounting for up to 90.8% on the GPU and 69.3% on GSCore. GSCore achieves a noticeable reduction in memory requirements for 3DGS rendering. However, it still requires an average bandwidth of 60.7 GB/s and 80.0 GB/s in FHD and QHD resolutions, respectively. This high bandwidth demand limits the availability of high-resolution 3DGS rendering in on-device AR/VR systems, whose practical DRAM bandwidths range from 17.8 GB/s to 59.7 GB/s [22, 31, 48, 50, 53–55, 70]. These findings highlight the need to optimize the sorting stage, which is the major bandwidth bottleneck in the 3DGS pipeline, to realize high-resolution and low-latency on-device 3DGS systems.

3.3 Opportunity to Reduce Computation in Sorting

To alleviate the bandwidth bottleneck caused by Gaussian sorting, we investigate the potential of temporally reusing Gaussians previously sorted in prior frames.

Per-frame sorting in 3DGS. In the sorting stage, each tile identifies the Gaussians that intersect it, determines their depth order for the subsequent rasterization stage, and stores this order in a Gaussian table. As the camera viewpoint changes, this table becomes outdated, so the system reprocesses the sorting stage from scratch for every frame. However, per-frame sorting overlooks the temporal similarity between consecutive frames, which arises from the gradual motion of Gaussians during camera movement and results in unnecessary memory bandwidth consumption.

Temporal similarity analysis. To quantify this temporal similarity, we first analyze the Gaussian retention between the Gaussian tables of consecutive frames. Figure 6 shows the cumulative distribution function (CDF) of the proportion of shared Gaussians across six real-world scenes. In all scenes, over 90% of tiles retain more than 78% of their Gaussians from the previous frame, highlighting the potential for reusing Gaussians from the previous frame's table.

In a second experiment, we measure how the ordering of Gaussians within each tile changes between consecutive frames. Figure 7 illustrates the sorting order differences at the 90th, 95th, and 99th percentiles. Results show that 99% of the sorting order remains largely consistent across consecutive frames. Notably, at the 99th percentile, the Gaussian with the greatest shift moves only 31 positions from its original location, a negligible deviation given that each tile contains thousands of Gaussians. Extending from the Gaussian retention observed in Figure 6, this result further substantiates the opportunity to leverage temporal similarity between frames to mitigate redundant updates in the Gaussian table.

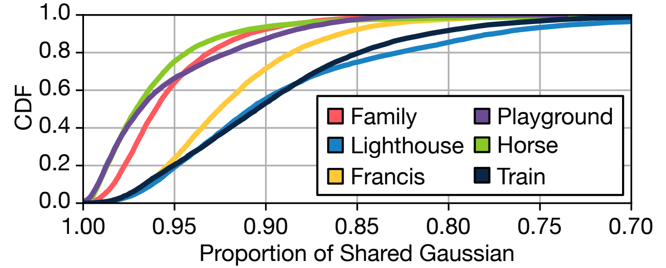

Figure 6. Temporal similarity of assigned Gaussian per tile.

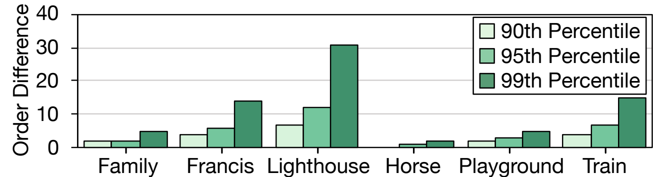

Figure 7. Temporal similarity of sort order per tile.

These findings indicate the need for a bandwidth-optimized sorting technique for on-device 3DGS systems. To address this, we propose Neo, an algorithm-hardware co-designed solution that enables efficient 3DGS rendering through memory-efficient sorting acceleration, leveraging temporal similarity in moving camera scenarios within AR/VR applications.

4 Neo's Reuse-and-Update Sorting

This section introduces the software component of our Neo solution, designed to efficiently leverage temporal similarity between consecutive frames to facilitate high-resolution 3DGS rendering on edge devices. By intelligently reusing sorting information across frames, we significantly reduce bandwidth requirements and computational overhead.

4.1 Design Space Exploration and Considerations

Design space exploration of sorting reuse methods. When leveraging temporal similarity under moving camera, two conventional strategies are periodic sorting and background sorting. Periodic sorting intermittently recomputes the full sorting order while skipping sorting in intermediate frames, which reduces average latency but introduces occasional spikes. Moreover, errors accumulate between refresh intervals, leading to gradual degradation in rendering quality. In contrast, background sorting [18, 45] continuously updates sorting results in parallel with rendering, and each frame uses the most recently prepared results. This approach mitigates latency spikes but introduces sustained memory traffic, which incurs memory contention and increases average latency. Moreover, discrepancies between the viewpoints of sorting and rendering frames degrade visual quality. To address these limitations, we propose an incremental update strategy that reuses the previous frame's sorting results while applying fine-grained corrections. Even under abrupt

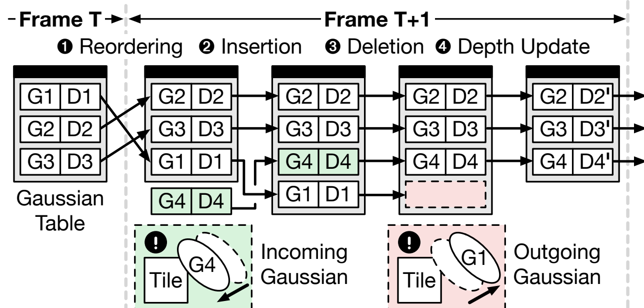

Figure 8. Overview of reuse-and-update sorting.

camera motion, this method recovers the correct ordering within a few frames, eliminating the need for full sorting.

Considerations for incremental update. In designing the incremental update strategy, we identify three primary sources of variability that the system must handle: ① Change in camera viewpoint can alter Gaussian depths, invalidating the previous ordering. ② New Gaussians can become visible in a specific tile. ③ Gaussians no longer relevant may disappear from the tile. Handling these factors is critical for robust incremental sorting, ensuring consistent accuracy.

4.2 Flow of Reuse-and-Update Sorting

Figure 8 shows the flow of the reuse-and-update sorting algorithm, consisting of four main operations.

❶ **Reordering.** We reuse the Gaussian table from the previous frame. However, when the viewpoint changes, the depth order of Gaussians changes, requiring reordering. To handle this efficiently, we employ *Dynamic Partial Sorting*, which performs partial sorting within on-chip memory and minimizes off-chip accesses. Section 4.3 provides further details.

❷ **Insertion.** We collect newly visible Gaussians entering a tile (i.e., incoming Gaussians) separately and insert them into the Gaussian table. Since the number of incoming Gaussians is typically small compared to the Gaussian table, this insertion incurs minimal computational overhead.

❸ **Deletion.** We identify Gaussians that move out of a tile (i.e., outgoing Gaussians) due to camera motion and remove them from the Gaussian table at each iteration.

After completing the three steps above, the system uses the updated Gaussian table for rasterization. During rasterization, it fetches necessary Gaussian features (e.g., position, depth, and color) to compute pixel values of the scene.

❹ **Depth Update.** During rasterization, we leverage the fact that each Gaussian's depth value is already available and refresh the corresponding depth entries in the Gaussian table to support reordering in future frames. This design eliminates additional irregular off-chip memory accesses that a separate depth-update pass would otherwise require. Section 4.4 describes this depth-update mechanism in detail.

After reordering, insertion, deletion, and depth update, the system forwards the Gaussian table with updated depth values to the next frame's 3DGS rendering pipeline.

Algorithm 1: Dynamic Partial Sorting

Input: I : *Current Frame Iteration Number*
G_{I-1} : *Previous Sorted Gaussian Table of Tile*
L : *Size of Gaussian Table of Tile*
C : *Size of Chunk for Chunk Sorting*
Output: G_I : *Current Sorted Gaussian Table of Tile*
// *Interleaving Sorting Boundaries*
1 **if** $I \ mod \ 2 \equiv 1$ **then**
2 $\quad range \leftarrow (start : 0, \ end : C)$;
3 **else**
4 $\quad range \leftarrow (start : 0, \ end : \lfloor \frac{C}{2} \rfloor)$;
5 **while** *true* **do**
$\quad$ // *Chunk-based Partial Sorting*
6 $\quad S \leftarrow \text{Slice}(G_{I-1}, \ range)$;
7 $\quad S' \leftarrow \text{Sort}(S)$;
8 $\quad \text{Slice}(G_I, \ range) \leftarrow S'$;
$\quad$ // *Update Sorting Parameters*
9 $\quad$ **if** $range.start + C \geq L$ **then**
10 $\quad\quad$ break ;
11 $\quad range.start \leftarrow range.start + C$;
12 $\quad range.end \leftarrow min(range.end + C, \ L)$;

4.3 Reordering the Reused Gaussian Table

In order to exploit the Gaussian tables carried over from previous frames, we devise a sorting algorithm that leverages temporal locality to make on-the-fly corrections. We refer to this algorithm as *Dynamic Partial Sorting*. Algorithm 1 outlines our *Dynamic Partial Sorting* approach, comprising two main strategies: chunk-based partial sorting and interleaving the sorting boundaries. By leveraging the sorted table from the previous frame, our method avoids a full global sort each time, reducing off-chip bandwidth usage while preserving accurate ordering over consecutive frames.

Chunk-based partial sorting (lines 5–12). We observe that the Gaussian sorting order within each tile remains largely consistent across adjacent frames. Based on this observation, our design partitions the Gaussian table into small chunks that fit within on-chip memory, and sorts each chunk independently. In our implementation, each chunk stores up to 256 Gaussians in on-chip memory. During each iteration, we read one chunk (line 6) from DRAM into on-chip memory, sort it in place (line 7), and write back the results (line 8). This local sorting approach significantly reduces bandwidth usage by retrieving and writing back each chunk only once, unlike conventional global sorting methods that make multiple passes over the entire table, repeatedly scanning and rewriting it. The algorithm continues chunk by chunk until the full table has been processed (lines 9–10).

Interleaving sorting boundaries (lines 1–4). Relying on a static partition can lead to inaccuracy if Gaussians need to cross chunk boundaries. Figure 9(a) illustrates the limitation of performing local sorting. Suppose at time t_1, the

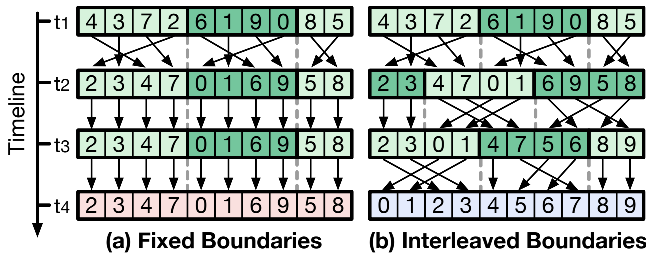

Figure 9. Comparison of two different partial sorting.

depths of Gaussians change due to camera viewpoint shifts, and requires reordering to depths 0 to 9. Because we only apply partial sorting within fixed chunks, Gaussians cannot cross these boundaries, even after multiple sorting iterations at subsequent times t_2, t_3, and t_4. In contrast, Figure 9(b) demonstrates our strategy to interleave sorting boundaries. Initially, at time t_1, we perform the same local sorting as in the fixed-boundary approach. However, starting from the next iteration (t_2), we interleave sorting boundaries, shifting chunk boundaries by half the chunk size. This staggered adjustment allows Gaussians to cross previous chunk boundaries, progressively reaching their correct positions. By repeating this process over subsequent iterations, Gaussians can freely move toward their correct sorting positions. As illustrated, by time t_4, all Gaussians have successfully reached their intended positions, highlighting the effectiveness of our interleaving sorting boundary method.

Single off-chip sorting pass. In our design, we retrieve each chunk from DRAM and write it back only once. While multiple sorting passes are possible, it introduces a trade-off between accuracy and memory traffic, depending on the number of off-chip passes. Increasing the number of passes guarantees more accurate Gaussian ordering, which improves rendering quality, but incurs memory traffic proportional to the number of passes. In practice, we observe that a single sorting pass introduces only negligible accuracy degradation (e.g., less than 0.1 dB). Because additional passes provide marginal benefit, we adopt a single off-chip sorting pass to minimize memory traffic.

Accuracy restoration. *Dynamic Partial Sorting* may require a few iterations to reestablish accurate ordering, potentially degrading accuracy. However, applying this technique reduces off-chip accesses and enhances sorting performance, leading to faster rendering. This creates a positive feedback loop: faster rendering enables more frequent sorting and updates, which, in turn, maintains accurate ordering during continuous camera movement. As a result, this technique leads to negligible accuracy degradation.

4.4 Bandwidth-Efficient Depth Update

Challenges in per-frame depth refresh. To keep the sorted Gaussian table up to date, we update the depth values stored in the table. A naïve design fetches each Gaussian's updated depth from the feature table in off-chip memory. However, this design incurs substantial random DRAM accesses because it refreshes the depth of each Gaussian in the table for every tile. Because our design prioritizes minimizing off-chip memory traffic, these random accesses significantly degrade performance.

Deferred depth update. We observe that rasterization already fetches each Gaussian's full feature from the off-chip memory. Instead of issuing a separate memory pass solely for depth updates, we piggyback on these accesses and directly overwrite the depth values in the sorted Gaussian table during rasterization, eliminating redundant memory accesses. However, this design defers the depth update such that, for any given frame, sorting relies on depth values that are one frame stale. In practice, this one-frame staleness introduces only negligible ordering error and does not degrade rendering quality. Without this optimization, depth updates require an additional memory access per Gaussian, which increases memory traffic by 33.2% compared to the full Neo design. Likewise, this optimization significantly reduces random off-chip memory accesses and improves overall performance.

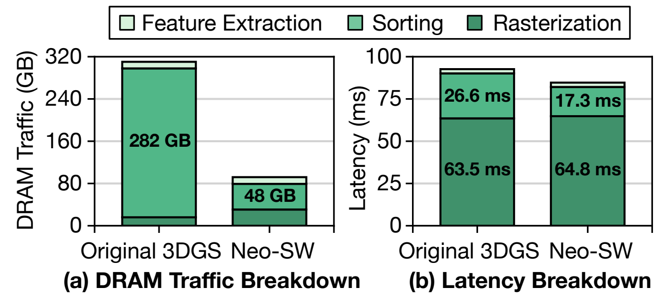

Figure 10. Performance comparison between the original 3DGS and Neo-SW on Orin AGX, with DRAM traffic measured over 60 rendered frames.

4.5 Performance Implication from Neo Algorithm

To evaluate the performance gains of a software-only version of Neo, we implement a custom CUDA kernel for sorting and modify the rasterization kernel. We enable *Dynamic Partial Sorting* in the ❶ reordering step by modifying existing sorting libraries from Meta [65] and NVIDIA [75]. We integrate the ❷ insertion and ❸ deletion steps into a merge-sorting process that combines incoming Gaussians with the reused Gaussian table. Finally, we implement ❹ deferred depth updates during rasterization. We evaluate this implementation on the NVIDIA Orin AGX platform.

Limitation of software-only solution. Figure 10 presents the latency and DRAM traffic of the software-only implementation of Neo, with DRAM traffic measured over 60 rendered frames. While the algorithm significantly reduces memory traffic, with 70.4% overall and 82.8% during the sorting stage as shown in Figure 10(a), its latency improvement remains modest at only 1.1×, as shown in Figure 10(b). This modest

speedup results from two primary factors. First, the insertion and deletion operations in the sorting stage induce irregular memory access patterns, degrading spatial locality and limiting SIMD utilization. Consequently, despite the significant reduction in memory traffic, the sorting stage achieves only a 1.54× speedup. Second, as prior work [49, 50, 104] highlights, rasterization remains the dominant bottleneck in GPU-based execution, accounting for 68.8% of total runtime. Since Neo specifically targets the sorting stage under the assumption that rasterization has already been accelerated, it delivers inherently limited end-to-end impact on GPU performance. These inefficiencies underscore the fundamental limitations of GPU-based execution. Addressing them requires a hardware-software co-designed architecture, which we introduce in the next section.

5 Neo Accelerator Architecture

This section presents the hardware architecture of Neo, which accelerates the 3DGS pipeline with reuse-and-update sorting. We first provide an overview of the data flow across the engines, followed by detailed descriptions of each engine.

5.1 Architecture Overview

Figure 11 illustrates the architecture of Neo. Our design consists of three main engines, Preprocessing Engine, Sorting Engine, and Rasterization Engine.

Preprocessing Engine. The Preprocessing Engine handles the first two stages of the 3DGS pipeline, namely frustum culling and feature extraction. For each Gaussian, it determines the tiles where the Gaussian has become newly visible and collects the per-Gaussian information required for rendering. These operations produce two types of tables: ❶ incoming Gaussian tables, which record the newly visible Gaussians for each tile, and ❷ a feature table, which stores Gaussian attributes required for rasterization.

Sorting Engine. The Sorting Engine performs three types of sorting. First, it applies *Dynamic Partial Sorting* to Gaussian tables from the previous frame that contain newly updated depth values. Second, it sorts the incoming Gaussian tables for the current frame provided by the Preprocessing Engine. Finally, it merges the results of these two steps. To accelerate reordering, sorting, and merging, the engine employs specialized parallel sorting units.

Rasterization Engine. Finally, the Rasterization Engine performs the last stage of the pipeline, rasterization, using both the sorted Gaussian tables and the feature table. To eliminate redundant computations, our design adopts subtiling [50], which applies α-blending only to each subtile for the Gaussians intersecting it. The Rasterization Engine integrates dedicated hardware for on-the-fly subtiling-based rasterization and for deferred depth updates, which enable reuse-and-update sorting in subsequent frames.

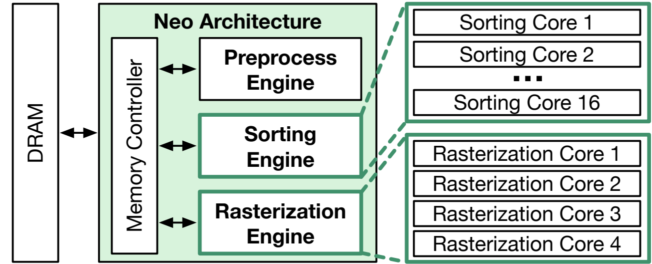

Figure 11. Overall of architecture of Neo.

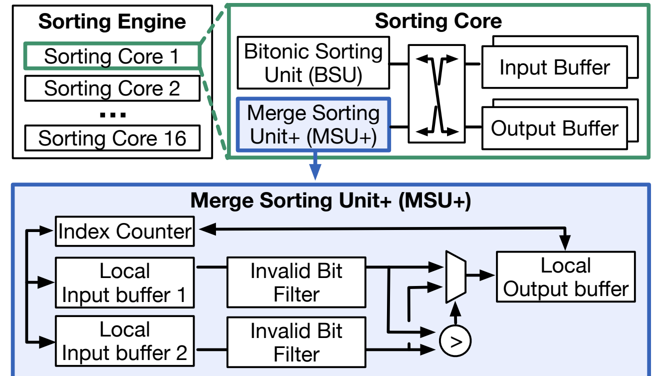

Figure 12. Microarchitecture of Sorting Engine.

5.2 Preprocessing Engine

The Preprocessing Engine consists of projection units and color calculation units for frustum culling and feature extraction, and incorporates duplication units to support the reuse-and-update sorting scheme.

Conventional preprocessing. The projection unit projects Gaussians onto the image plane and performs frustum culling to discard those outside the camera frustum. Next, the color calculation unit derives view-dependent color for each Gaussian using spherical harmonics. Together, these units generate a feature table that stores essential rasterization attributes, including color, mean, covariance matrix, opacity, and radius.

Processing incoming Gaussians. The duplication unit uses 2D Gaussian information to identify the tiles intersected by each Gaussian and generates the corresponding Gaussian tables. Our design adds a verification step that checks whether each Gaussian exists in the previous frame's Gaussian table. This step produces the incoming Gaussian tables by allowing the system to process only newly visible Gaussians, thereby enabling the reuse-and-update sorting scheme. As a result, the unit outputs per-tile Gaussian tables containing the IDs and depth values of newly incoming Gaussians.

5.3 Sorting Engine

Figure 12 shows the microarchitecture of our Sorting Engine, which comprises 16 parallel Sorting Cores. Each Sorting Core has a Bitonic Sorting Unit (BSU), a Merge Sorting

Unit+ (MSU+), and I/O buffers. Both input and output buffers employ double buffering to mask memory-access latency.

Conventional sorting. To perform conventional sorting of the Gaussian table from scratch, the Sorting Core follows standard merge-sort steps. Each core loads a 256-entry chunk into its input buffer and partitions it into smaller 16-entry sub-chunks that the BSU can process. The BSU sorts each sub-chunk, and the MSU+ merges the partially sorted results. The system then writes each fully sorted chunk back to DRAM, processes the remaining chunks, and finally performs a global merge across all sorted chunks.

Dynamic Partial Sorting. Beyond conventional sorting, the Sorting Engine supports *Dynamic Partial Sorting* to reuse the Gaussian table from the previous frame. Specifically, the engine loads a 256-entry chunk from the previous frame's table and reorders it in the same manner as conventional sorting, using the BSU to sort sub-chunks followed by an MSU+ merge. Because no additional merges are required across chunks, the design avoids extra off-chip memory traffic.

Inserting and deleting entries. Alongside conventional sorting and *Dynamic Partial Sorting*, the MSU+ merges the sorted Gaussian table with the incoming Gaussian table and removes outgoing Gaussians from the table. Specifically, the Preprocessing Engine first generates the incoming Gaussian tables, and the Sorting Engine sorts them using a conventional algorithm before merging them with the sorted Gaussian table from the previous frame. At the same time, the MSU+ removes outgoing Gaussians based on their valid bits, which were marked as valid or invalid during the previous frame's rasterization. This design stems from the observation that, although outgoing Gaussians can be excluded during rasterization using their valid-bit flag, immediately removing them would require costly shifting of subsequent entries. By deferring memory realignment to the merge step, the MSU+ efficiently deletes invalid entries without incurring the cost of shifting the entries. Furthermore, it inserts new entries simultaneously, thereby improving performance.

5.4 Rasterization Engine

Figure 13 illustrates the microarchitecture of our Rasterization Engine, comprising four Rasterization Cores. Each core contains four Subtile Compute Units (SCU), four Intersection Test Units (ITU), and dedicated buffers for bitmaps, 2D Gaussian features, and pixel data.

Intersection Test Unit (ITU). Our design adopts subtile-based rasterization, inspired by GSCore [50], which subdivides each tile into multiple smaller subtiles. Since a Gaussian typically intersects only a subset of subtiles within a tile, this approach reduces redundant computations. To track Gaussian–subtile intersections, GSCore maintains lightweight subtile metadata in the form of a bitmap that indicates whether a Gaussian overlaps each subtile. However, GSCore generates these bitmaps early in the pipeline and propagates them through all stages, even though they are only used during

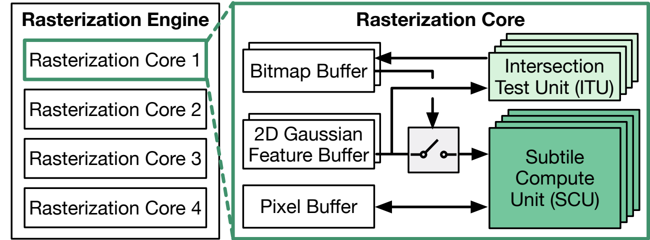

Figure 13. Microarchitecture of Rasterization Engine.

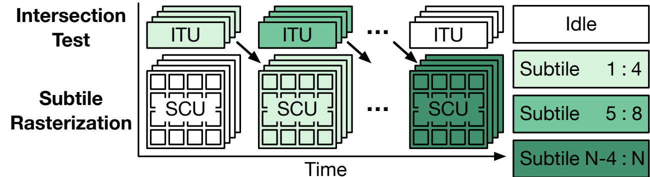

Figure 14. Execution timeline of Rasterization Engine. Intersection Test Unit (ITU) and Subtile Computation Unit (SCU) process subtiles in a pipelined manner.

rasterization, causing unnecessary memory traffic. To address this inefficiency, we integrate Intersection Test Units (ITUs) within the Rasterization Core to generate the required bitmaps on-the-fly. Specifically, each ITU uses the 2D parameters of a Gaussian to test intersection boundaries and store the resulting bitmap in a local buffer. Moreover, the ITU plays a key role in reuse-and-update sorting by detecting outgoing Gaussians through a cumulative OR operation. This operation accumulates intersection bitmaps across all subtiles to flag Gaussians with at least one intersection. By flipping this bit, the system identifies Gaussians with no intersections and will eliminate them during the next sorting stage.

Subtile Computation Unit (SCU). Following the intersection tests, each SCU performs α-blending to compute pixel values for its assigned subtile. The system filters Gaussians using the bitmaps generated by the ITUs and forwards them only to the SCUs whose subtiles intersect the Gaussian. After rasterizing a group of subtiles with the 2D Gaussian feature buffer, the system writes the intermediate pixel values to the pixel buffer and advances to the next group. To maximize efficiency, we pipeline intersection testing with rasterization. Figure 14 illustrates this pipelining. While the first group of four subtiles (1–4) must wait for its intersection tests to complete, subsequent groups benefit from overlapped execution, as the ITUs process the next group in parallel with the current group's rasterization. This overlap effectively hides the latency of on-the-fly bitmap generation.

Updating table for next frame. After rasterizing all subtiles, the system extracts the depth values from the feature buffer and retrieves the valid bits from the bitmap buffer. It then updates the corresponding entries in the Gaussian table and forwards the updated table to the next frame.

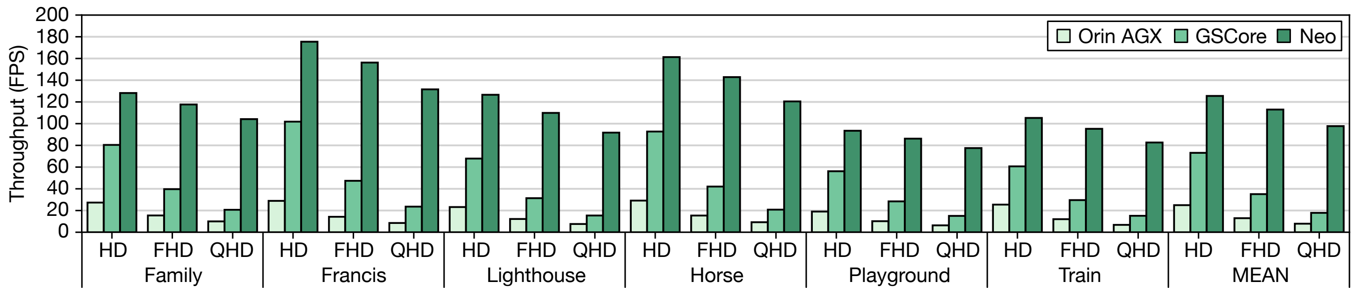

Figure 15. End-to-end system throughput of GSCore and Neo on six 3D scenes.

Table 1. Configuration of Neo system.

Hardware Component		Configuration
Neo	Tile Size	64×64 px
Preprocessing Engine	Projection Unit	4 units
	Color Unit	4 units
	Duplication Unit	4 units
Sorting Engine	Bitonic Sort Unit	16 units
	Merge Sort Unit+	16 units
	I/O Buffer Size	64 KB
Rasterization Engine	Subtile Compute Unit	16 units
	Intersection Test Unit	16 units
	Buffer Size	200 KB
	Subtile Size	8×8 px

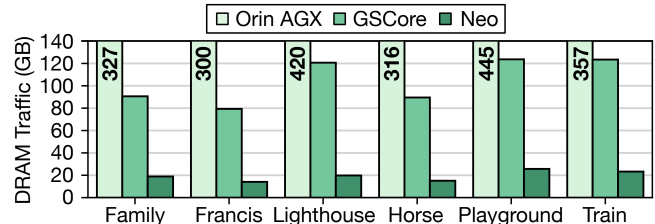

Figure 16. Comparison of DRAM traffic (GB) between Neo and GSCore for rendering 60 frames.

6 Evaluation

6.1 Methodology

Benchmarks. We select six scenes from the Tanks and Temples dataset [43], namely Family, Francis, Horse, Lighthouse, Playground, and Train, as representative benchmarks for evaluating 3DGS rendering quality. We capture each frame at 30 FPS in UHD resolution (3840×2160). Following the standard training procedure, we use 400 images per scene and train for 200K iterations to ensure model convergence. For inference, we configure each scene to one of three target resolutions: HD (1280×720), FHD (1920×1080), or QHD (2560×1440), and evaluate how many unique frames the system renders per second, with each frame using a distinct camera pose from the original sequence of the dataset.

Hardware development and synthesis. Table 1 shows the configurations of our hardware. We prototype Neo architecture at RTL level using Verilog, and measure power, area, and timing parameters using Synopsys Design Compiler with ASAP7 [98] 7 nm library. We measure the power and area of on-chip buffers using CACTI [72] under 22 nm technology and scale them to 7 nm using DeepScaleTool [87].

Baseline. We evaluate Neo against two baseline systems: NVIDIA Orin AGX 64GB (Orin AGX), GSCore. Orin AGX [38] is a high-performance on-device platform designed for autonomous systems. It supports up to 60W of power and provides 204.8 GB/s of memory bandwidth. This baseline enables evaluation of rendering performance on real on-device hardware. GSCore [50] originally features four sorting and rasterization cores. To support high-resolution workloads and enable a fair comparison with Neo, which includes 16 hardware units, we scale GSCore to 16 cores.

Hardware simulator. We measure the latency of GSCore, and Neo using an in-house hardware simulator based on the timing parameters obtained from RTL synthesis, with off-chip memory modeled as LPDDR4 based on Ramulator [42].

6.2 Performance Results

End-to-end throughput. Figure 15 presents the end-to-end rendering throughput for six scenes rendered at three target resolutions (HD, FHD, and QHD) on Orin AGX, GSCore, and Neo. Across all scenes and resolutions, Neo consistently outperforms both Orin AGX and GSCore, achieving average 5.0×, 8.7×, and 12.4× speedup over Orin AGX and 1.7×, 3.2×, and 5.5× over GSCore for HD, FHD, and QHD, respectively. The performance gains are particularly significant at higher resolutions, where sorting bottlenecks intensify, demonstrating Neo's effectiveness in addressing bandwidth bottlenecks through temporal similarity. Unlike Orin AGX and GSCore, which sort from scratch in every frame, Neo reduces bandwidth usage with a reuse-and-update sorting approach. It applies *Dynamic Partial Sorting* to tables from the previous frame, while performing conventional sorting only on small incoming Gaussian tables. Notably, Neo achieves an average throughput of 97.7 FPS at QHD resolution, meeting the real-time rendering requirement.

End-to-end memory traffic. Figure 16 shows the required DRAM traffic to render 60 frames in QHD resolution on Orin AGX, GSCore, and Neo. For Orin AGX, rendering 60 frames requires an average of 360.8 GB across six scenes, compared to 104.6 GB for GSCore and 19.5 GB for Neo, representing a 94.6% and 81.4% reduction over OrinAGX and

Table 2. Quality comparison of original 3DGS and Neo.

Scene	Original 3DGS		Neo	
	PSNR↑	LPIPS↓	PSNR↑	LPIPS↓
Family	30.2	0.037	30.0 (▼0.2)	0.041 (▲0.004)
Francis	29.2	0.161	29.2 (●)	0.161 (●)
Horse	28.0	0.085	27.9 (▼0.1)	0.086 (▲0.001)
Lighthouse	26.1	0.071	26.1 (●)	0.071 (●)
Playground	25.2	0.179	25.2 (●)	0.179 (●)
Train	25.4	0.087	25.3 (▼0.1)	0.088 (▲0.001)

Table 3. Evaluated GSCore and Neo accelerators.

Device	Technology	Frequency	Area (mm^2)	Power (mW)
GSCore	7 nm	1 GHz	0.417	719.9
Neo			0.387	797.8

Table 4. Area and power breakdown of hardware components in Neo accelerator.

Component	Area (mm^2)	Power (mW)
Preprocessing Engine	**0.026**	**194.9**
Merge Sort Unit+	0.005	12.4
Bitonic Sort Unit	0.008	75.0
Buffers + others	0.040	71.6
Sorting Engine	**0.053**	**159.0**
Subtile Compute Unit	0.228	375.0
Intersection Test Unit	0.030	58.7
Buffers + others	0.050	10.2
Rasterization Engine	**0.308**	**443.9**
Total	**0.387**	**797.8**

GSCore, respectively. This reduction is largely due to significant traffic savings achieved through reuse-and-update sorting. As a result, even in scenarios with limited bandwidth (e.g., 51.2 GB/s), the system can perform computations without being bottlenecked by the bandwidth constraints.

Rendering quality. Table 2 compares the rendering quality of GSCore and Neo. We evaluate rendering quality using standard graphics metrics: PSNR, where higher values indicate better image fidelity, and LPIPS, where lower values indicate better perceptual quality. Across all scenes, the maximum observed difference is less than 1.0 dB in PSNR, an imperceptible level of quality degradation [19, 20]. These results highlight the effectiveness of Neo's sorting mechanism in maintaining high rendering quality while significantly reducing sorting overhead by exploiting temporal similarity.

Area and power. Table 3 compares the evaluated area and power of Neo and GSCore. For a fair comparison, we scale GSCore's area and power estimates down to 7 nm using DeepScaleTool [87], as it was originally synthesized in a 28 nm process. The results show that Neo achieves a slightly smaller total area than GSCore with a marginal increase in power consumption. To further understand the area and power overhead, Table 4 provides a detailed breakdown of

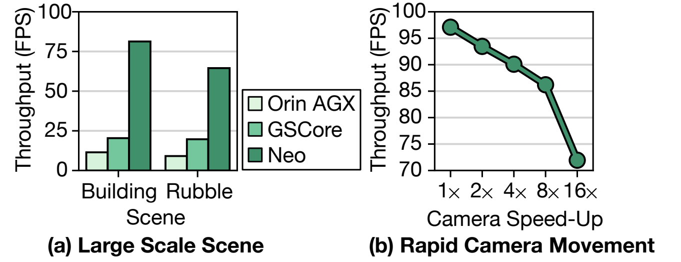

Figure 17. Throughput of 3DGS rendering systems in extreme AR/VR scenarios: (a) Large-Scale Scene, and (b) Rapid Camera Movement.

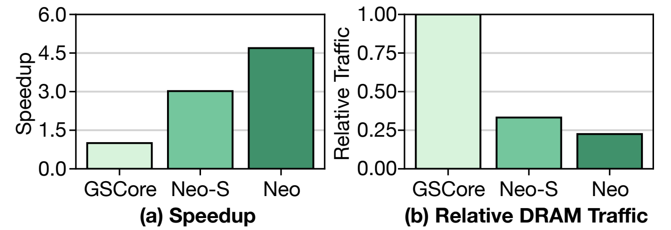

Figure 18. Speedup and DRAM traffic normalized to GSCore. Neo-S replaces GSCore Sorting Engine with Neo's Sorting Engine.

our Neo accelerator, showing that its additional hardware components (Merge Sort Unit+ and Intersection Test Unit) together account for 9.04% and 8.91% of the total area and power consumption, respectively. With minimal overhead, our additional hardware block delivers high throughput under high-resolution settings.

Performance results of extreme AR/VR scenarios. Figure 17 shows the throughput of Neo under the more stringent constraints of AR/VR scenarios, including large scale scene rendering or rapid camera movement. Figure 17(a) shows the results of Building and Rubble scenes from Mill 19 [96] which features high-resolution aerial imagery commonly used to represent a complex, large-scale scene. Neo delivers an average throughput of 72.9 FPS, whereas both Orin AGX and GSCore struggle to meet the high frame rate requirements, dropping below 10.3 FPS and 20.0 FPS, respectively. Figure 17(b) presents Neo's performance under different levels of rapid camera movement (2×, 4×, 8× and 16×). In these scenarios, although Gaussian reusability decreases under rapid camera motion, Neo maintains a frame rate above 60 FPS, satisfying the conservative SLO requirement for rendering. These results demonstrate the effectiveness of Neo's reuse-and-update sorting mechanism, exhibiting superior rendering performance across extreme AR/VR scenarios.

6.3 Ablation Studies

Performance breakdown of Neo hardware. The hardware of Neo consists of two main components: the Sorting Engine and the Rasterization Engine. The Sorting Engine

implements the first three steps of Neo's reuse-and-update sorting algorithm: reordering, insertion, and deletion, while the Rasterization Engine performs subtile-based rasterization and depth update. Figure 18 shows the incremental benefits of integrating these components into GSCore. Although GSCore alone does not support Neo's algorithm, adding the Sorting Engine (Neo-S) enables reuse-and-update sorting, reducing memory traffic by 66.8% and improving performance by 3.0×. However, without hardware support for depth update, the system requires separate post-processing to update Gaussian table metadata (e.g., depth, valid bit), which incurs additional delay. Integrating the Rasterization Engine removes this overhead, achieving a further 32.2% traffic reduction and an additional 1.6× speedup. While the Sorting Engine provides substantial benefits, full algorithm support requires a co-designed sorting and rasterization engine. Accordingly, Neo adopts a holistic hardware design that integrates both components to maximize performance.

Comparison with existing sorting methods. In Section 4.1, we explore the design space of sorting reuse methods, including periodic sorting, background sorting, and incremental update sorting. Based on this analysis, we adopt incremental update sorting in our design. To evaluate its effectiveness in terms of latency and rendering quality, we compare it against two alternative strategies on the Neo hardware: ❶ periodic sorting and ❷ background sorting. We also consider ❸ hierarchical sorting, originally proposed in GSCore, which accelerates sorting by combining coarse-grained and fine-grained sorting. We evaluate hierarchical sorting in conjunction with incremental update sorting to quantify the benefits of the *Dynamic Partial Sorting*. Figure 19 compares the three sorting methods. ❶ Periodic sorting achieves lower average latency than Neo by avoiding continuous updates. However, it introduces periodic latency spikes that violate the 16.6 ms SLO for 60 FPS and suffers from severe quality degradation due to error accumulation between updates. ❷ Background sorting maintains relatively stable latency by continuously sorting in the background. Despite this, it incurs higher average latency than Neo and degrades rendering quality due to temporal viewpoint discrepancies between sorting and rendering. ❸ Hierarchical sorting accurately sorts reused Gaussians and delivers rendering quality comparable to Neo. However, it requires multiple passes over memory, which increases latency. These results demonstrate the effectiveness of Neo's sorting scheme and support our design choice.

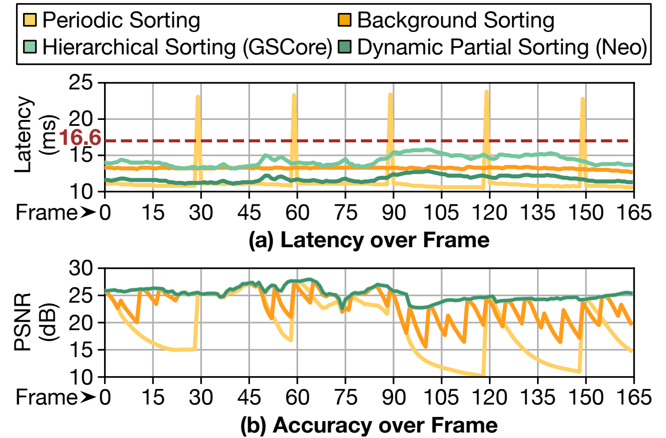

Figure 19. Latency and rendering quality across frames for four sorting reuse methods.

7 Related Work

View synthesis acceleration. Neural rendering has driven significant advances, prompting the architecture community to explore hardware acceleration [20, 22, 47, 48, 53, 55, 56, 61, 70, 80, 89, 90]. With the emergence of 3D Gaussian Splatting (3DGS), research focus has shifted accordingly. GSCore [50] introduces hierarchical sorting and an optimized rasterization pipeline. GBU [104] improves efficiency by reusing rasterization results. VR-Pipe [49] repurposes the GPU rasterization engine with microarchitectural optimizations. MetaSapiens [59] applies load balancing techniques to tile-based rasterization. For training, ARC [13] and GSArch [30] mitigate atomic overheads using warp-level reductions and gradient filtering. GauSPU [103] targets SLAM-integrated 3DGS with a sparsity-adaptive rasterizer and a relaxed-memory backpropagation engine. Our work builds on prior 3DGS accelerators that address rasterization bottlenecks and shifts the focus to a more practical constraint in high resolution rendering, revealing sorting as the next critical performance limiter. Neo targets this bottleneck with a lightweight, reuse-aware sorting engine that complements prior efforts.

Memory-efficient 3DGS. Another branch of research reduces the memory footprint of 3DGS through pruning and quantization. Pruning eliminates low-impact Gaussians based on importance metrics such as opacity [16, 73] or α-blending weights [17, 59]. Quantization techniques, including vector quantization [16] and learned compression [17, 25, 73, 78], reduce both memory and storage overhead. However, these approaches require retraining or fine-tuning. In contrast, Neo introduces an orthogonal sorting scheme that requires no retraining. Moreover, it complements existing methods, enabling further gains in bandwidth efficiency.

Leveraging temporal similarity. Numerous studies on real-time streaming systems [10, 14, 21, 36, 37, 41, 58, 64, 79, 91, 92, 94, 105, 107, 108] have been proposed to reduce computation while maintaining accuracy. Across diverse domains, one of the most effective strategies is to exploit temporal redundancy. In graphics, prior work [11, 20, 26, 57, 63, 88] applies this principle through pixel-wise reuse, known as warping. Neural rendering also benefits from temporal redundancy. For example, Potamoi [19] leverages it to bypass MLP overheads in NeRF through pixel reuse. However, 3DGS

performs Gaussian-wise feature extraction and tile-wise sorting and rasterization, which limit the applicability of pixel-wise reuse and fail to address the memory traffic induced by sorting. In contrast, Lumina [18] leverages temporal similarity at the sorting stage but performs background sorting, which continuously consumes memory bandwidth, incurs contention, and increases average latency, as discussed in Section 4.1 and Section 6.3. Neo instead exploits temporal similarity at the sorting stage through an incremental update strategy, effectively reducing the associated memory traffic.

8 Conclusion

Real-time on-device 3D Gaussian Splatting (3DGS) rendering demands both low latency and high frame generation rate, yet existing solutions struggle to meet these conflicting requirements under stringent resource constraints. This work identifies Gaussian sorting as a key bottleneck in the 3DGS inference pipeline, and presents Neo, an on-device acceleration solution that introduces a reuse-and-update sorting algorithm and a hardware-accelerated sorting pipeline to reduce redundant computations and alleviate memory bandwidth pressure. By tackling this challenge, Neo takes a step toward enabling real-time, on-device generative virtual worlds, bringing us closer to immersive AR/VR experiences.

Acknowledgments

We thank the anonymous reviewers for their comments and feedback. This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation-Information Technology Research Center (IITP-ITRC) grant funded by the Korea government (MSIT) (IITP-2026-RS-2020-II201795, 33%), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2024-00342148, 33%), and Institute of Information & Communications Technology Planning & Evaluation (IITP) under the Graduate School of Artificial Intelligence Semiconductor grant funded by the Korea government (MSIT) (IITP-2025-RS-2023-00256472, 33%). The EDA tool was supported by the IC Design Education Center (IDEC), Korea.

References

[1] Anthropic. 2024. Introducing the next generation of Claude. https://www.anthropic.com/news/claude-3-family
[2] Apple. 2023. Apple Vision Pro - Technical Specifications. https://www.apple.com/apple-vision-pro/specs/.
[3] Ars Technica. 2013. How fast does "virtual reality" have to be to look like "actual reality"? https://arstechnica.com/gaming/2013/01/how-fast-does-virtual-reality-have-to-be-to-look-like-actual-reality/.
[4] Miguel Angel Bautista, Pengsheng Guo, Samira Abnar, Walter Talbott, Alexander Toshev, Zhuoyuan Chen, Laurent Dinh, Shuangfei Zhai, Hanlin Goh, Daniel Ulbricht, et al. 2022. Gaudi: A neural architect for immersive 3d scene generation. In *NeurIPS*.
[5] Sandeepa Bhuyan, Ziyu Ying, Mahmut T Kandemir, Mahanth Gowda, and Chita R Das. 2024. GameStreamSR: Enabling Neural-Augmented Game Streaming on Commodity Mobile Platforms. In *ISCA*.
[6] Sandeepa Bhuyan, Shulin Zhao, Ziyu Ying, Mahmut T Kandemir, and Chita R Das. 2022. End-to-end characterization of game streaming applications on mobile platforms. In *POMACS*.
[7] Andreas Blattmann, Tim Dockhorn, Sumith Kulal, Daniel Mendelevitch, Maciej Kilian, Dominik Lorenz, Yam Levi, Zion English, Vikram Voleti, Adam Letts, et al. 2023. Stable video diffusion: Scaling latent video diffusion models to large datasets. *arXiv preprint arXiv:2311.15127* (2023).
[8] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, Jean-Baptiste Lespiau, Bogdan Damoc, Aidan Clark, et al. 2022. Improving language models by retrieving from trillions of tokens. In *ICML*.
[9] Robert A Brebin, Loren Carpenter, and Pat Hanrahan. 1998. Volume rendering. In *Seminal graphics: pioneering efforts that shaped the field.*
[10] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting temporal redundancy in live computer vision. In *ISCA*.
[11] Shenchang Eric Chen. 1995. Quicktime VR: An image-based approach to virtual environment navigation. In *SIGGRAPH*.
[12] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *NeurIPS*.
[13] Sankeerth Durvasula, Adrian Zhao, Fan Chen, Ruofan Liang, Pawan Kumar Sanjaya, Yushi Guan, Christina Giannoula, and Nandita Vijaykumar. 2025. ARC: Warp-level Adaptive Atomic Reduction in GPUs to Accelerate Differentiable Rendering. In *ASPLOS*.
[14] Matthew Dutson, Yin Li, and Mohit Gupta. 2023. Eventful transformers: Leveraging temporal redundancy in vision transformers. In *ICCV*.
[15] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, et al. 2024. Scaling rectified flow transformers for high-resolution image synthesis. In *ICML*.
[16] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, Zhangyang Wang, et al. 2025. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. In *NeurIPS*.
[17] Guangchi Fang and Bing Wang. 2024. Mini-splatting: Representing scenes with a constrained number of gaussians. In *ECCV*.
[18] Yu Feng, Weikai Lin, Yuge Cheng, Zihan Liu, Jingwen Leng, Minyi Guo, Chen Chen, Shixuan Sun, and Yuhao Zhu. 2025. Lumina: Real-Time Neural Rendering by Exploiting Computational Redundancy. In *ISCA*.
[19] Yu Feng, Weikai Lin, Zihan Liu, Jingwen Leng, Minyi Guo, Han Zhao, Xiaofeng Hou, Jieru Zhao, and Yuhao Zhu. 2024. Potamoi: Accelerating Neural Rendering via a Unified Streaming Architecture. In *TACO*.
[20] Yu Feng, Zihan Liu, Jingwen Leng, Minyi Guo, and Yuhao Zhu. 2024. Cicero: Addressing algorithmic and architectural bottlenecks in neural rendering by radiance warping and memory optimizations. In *ISCA*.
[21] Yu Feng, Paul Whatmough, and Yuhao Zhu. 2019. Asv: Accelerated stereo vision system. In *MICRO*.
[22] Yonggan Fu, Zhifan Ye, Jiayi Yuan, Shunyao Zhang, Sixu Li, Haoran You, and Yingyan Lin. 2023. Gen-nerf: Efficient and generalizable neural radiance fields via algorithm-hardware co-design. In *ISCA*.
[23] Stephan J Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. 2021. Fastnerf: High-fidelity neural rendering at 200fps. In *ICCV*.
[24] Ali Geris, Baris Cukurbasi, Murat Kilinc, and Orkun Teke. 2024. Balancing performance and comfort in virtual reality: A study of FPS, latency, and batch values. In *Software: Practice and Experience.*
[25] Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. 2024. Eagles: Efficient accelerated 3d gaussians with lightweight encodings. In *ECCV*.

[26] Chris A Glasbey and Kantilal Vardichand Mardia. 1998. A review of image-warping methods. In *Journal of applied statistics*.

[27] Google. 2025. Android XR. https://www.android.com/intl/en_eu/xr/.

[28] Robin Green. 2003. Spherical harmonic lighting: The gritty details. In *Archives of the game developers conference*.

[29] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[30] Houshu He, Gang Li, Fangxin Liu, Li Jiang, Xiaoyao Liang, and Zhuoran Song. 2025. GSArch: Breaking Memory Barriers in 3D Gaussian Splatting Training via Architectural Support. In *HPCA*.

[31] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A roofline model for mobile socs. In *HPCA*.

[32] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. In *NeurIPS*.

[33] Jonathan Ho, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J Fleet. 2022. Video diffusion models. In *NeurIPS*.

[34] HP. 2021. HP Reverb G2 Virtual Reality Headset Datasheet.

[35] HTC. 2023. VIVE XR Elite. https://www.vive.com/us/product/vive-xr-elite/specs/.

[36] Jinwoo Hwang, Daeun Kim, Sangyeop Lee, Yoonsung Kim, Guseul Heo, Hojoon Kim, Yunseok Jeong, Tadiwos Meaza, Eunhyeok Park, Jeongseob Ahn, and Jongse Park. 2025. Déjà Vu: Efficient Video-Language Query Engine with Learning-Based Inter-Frame Computation Reuse. In *PVLDB*.

[37] Jinwoo Hwang, Minsu Kim, Daeun Kim, Seungho Nam, Yoonsung Kim, Dohee Kim, Hardik Sharma, and Jongse Park. 2022. CoVA: Exploiting Compressed-Domain analysis to accelerate video analytics. In *ATC*.

[38] Leela S Karumbunathan. 2022. Nvidia jetson agx orin series. *A Giant Leap Forward for Robotics and Edge AI Applications. Technical Brief* (2022).

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

[40] Daeun Kim, Jinwoo Hwang, Changhun Oh, and Jongse Park. 2025. MixDiT: Accelerating Image Diffusion Transformer Inference With Mixed-Precision MX Quantization. In *CAL*.

[41] Yoonsung Kim, Changhun Oh, Jinwoo Hwang, Wonung Kim, Seongryong Oh, Yubin Lee, Hardik Sharma, Amir Yazdanbakhsh, and Jongse Park. 2024. Dacapo: Accelerating continuous learning in autonomous systems for video analytics. In *ISCA*.

[42] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. In *CAL*.

[43] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. In *ACM Transactions on Graphics*.

[44] Weihao Kong, Yifan Hao, Qi Guo, Yongwei Zhao, Xinkai Song, Xiaqing Li, Mo Zou, Zidong Du, Rui Zhang, Chang Liu, et al. 2024. Cambricon-d: Full-network differential acceleration for diffusion models. In *ISCA*.

[45] Kwok, Kevin. 2025. WebGL 3D Gaussian Splat Viewer. https://github.com/antimatter15/splat.

[46] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *SOSP*.

[47] Hongseok Lee, Gwangtae Park, Wonhoon Park, Wooyoung Jo, Jongjun Park, and Hoi-Jun Yoo. 2024. A 66.6 FPS High Quality Gaussian Splats Rendering FPGA Processor with Reconfigurable Computation Architecture. In *A-SSCC*.

[48] Junseo Lee, Kwanseok Choi, Jungi Lee, Seokwon Lee, Joonho Whangbo, and Jaewoong Sim. 2023. Neurex: A case for neural rendering acceleration. In *ISCA*.

[49] Junseo Lee, Jaisung Kim, Junyong Park, and Jaewoong Sim. 2025. VR-Pipe: Streamlining Hardware Graphics Pipeline for Volume Rendering. In *HPCA*.

[50] 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 *ASPLOS*.

[51] Marc Levoy and Pat Hanrahan. 2023. Light field rendering. In *Seminal Graphics Papers: Pushing the Boundaries*.

[52] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*.

[53] Chaojian Li, Sixu Li, Linrui Jiang, Jingqun Zhang, and Yingyan Celine Lin. 2025. Uni-Render: A Unified Accelerator for Real-Time Rendering Across Diverse Neural Renderers. In *HPCA*.

[54] Chaojian Li, Sixu Li, Yang Zhao, Wenbo Zhu, and Yingyan Lin. 2022. Rt-nerf: Real-time on-device neural radiance fields towards immersive ar/vr rendering. In *ICCAD*.

[55] Sixu Li, Chaojian Li, Wenbo Zhu, Boyang Yu, Yang Zhao, Cheng Wan, Haoran You, Huihong Shi, and Yingyan Lin. 2023. Instant-3d: Instant neural radiance field training towards on-device ar/vr 3d reconstruction. In *ISCA*.

[56] Sixu Li, Yang Zhao, Chaojian Li, Bowei Guo, Jingqun Zhang, Wenbo Zhu, Zhifan Ye, Cheng Wan, and Yingyan Celine Lin. 2024. Fusion-3D: Integrated Acceleration for Instant 3D Reconstruction and Real-Time Rendering. In *MICRO*.

[57] Yong Li and Wei Gao. 2019. DeltaVR: Achieving high-performance mobile VR dynamics through pixel reuse. In *IPSN*.

[58] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *SIGCOMM*.

[59] Weikai Lin, Yu Feng, and Yuhao Zhu. 2025. MetaSapiens: Real-Time Neural Rendering with Efficiency-Aware Pruning and Accelerated Foveated Rendering. In *ASPLOS*.

[60] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434* (2024).

[61] Tianbo Liu, Xinkai Song, Zhifei Yue, Rui Wen, Xing Hu, Zhuoran Song, Yuanbo Wen, Yifan Hao, Wei Li, Zidong Du, et al. 2025. Cambricon-SR: An Accelerator for Neural Scene Representation with Sparse Encoding Table. In *ISCA*.

[62] Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, et al. 2024. Sora: A review on background, technology, limitations, and opportunities of large vision models. *arXiv preprint arXiv:2402.17177* (2024).

[63] Seán K Martin, Seán Bruton, David Ganter, and Michael Manzke. 2019. Synthesising light field volume visualisations using image warping in real-time. In *VISIGRAPP*.

[64] Jiayi Meng, Sibendu Paul, and Y Charlie Hu. 2020. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *ASPLOS*.

[65] Meta. 2018. Facebook's CUDA extensions. https://github.com/facebookarchive/fbcuda.

[66] Meta. 2023. Meta Quest 3. https://www.meta.com/quest/quest-3/.

[67] Meta. 2025. Ray-Ban Meta. https://www.meta.com/ai-glasses/ray-ban-meta.

[68] Microsoft. 2023. About HoloLens 2. https://learn.microsoft.com/en-us/hololens/hololens2-hardware

[69] 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. In *ECCV*.

[70] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. 2023. Hardware acceleration of neural graphics. In *ISCA*.

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

[72] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *MICRO*.

[73] KL Navaneet, Kossar Pourahmadi Meibodi, Soroush Abbasi Koohpayegani, and Hamed Pirsiavash. 2024. CompGS: Smaller and Faster Gaussian Splatting with Vector Quantization. In *ECCV*.

[74] NVIDIA. 2023. NVIDIA Jetson AGX Orin Series - A Giant Leap Forward for Robotics and Edge AI Applications. https://www.nvidia.com/content/dam/en-zz/Solutions/gtcf21/jetson-orin/nvidia-jetson-agx-orin-technical-brief.pdf.

[75] NVIDIA. 2025. NVIDIA CUDA Core Compute Libraries (CCCL). https://github.com/NVIDIA/cccl.

[76] OpenAI. 2022. Introducing ChatGPT. https://openai.com/index/chatgpt/

[77] OpenAI. 2024. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2024).

[78] Panagiotis Papantonakis, Georgios Kopanas, Bernhard Kerbl, Alexandre Lanvin, and George Drettakis. 2024. Reducing the memory footprint of 3d gaussian splatting. In *PACMCGIT*.

[79] Mathias Parger, Chengcheng Tang, Christopher D Twigg, Cem Keskin, Robert Wang, and Markus Steinberger. 2022. Deltacnn: End-to-end cnn inference of sparse frame differences in videos. In *CVPR*.

[80] 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 *MICRO*.

[81] PICO. 2024. PICO 4 Ultra Specs. https://www.picoxr.com/global/products/pico4-ultra/specs.

[82] Pimax. 2023. Pimax Crystal. https://pimax.com/pages/crystal

[83] Thomas Porter and Tom Duff. 1984. Compositing digital images. In *ACM SIGGRAPH*.

[84] Guocheng Qian, Jinjie Mai, Abdullah Hamdi, Jian Ren, Aliaksandr Siarohin, Bing Li, Hsin-Ying Lee, Ivan Skorokhodov, Peter Wonka, Sergey Tulyakov, et al. 2024. Magic123: One Image to High-Quality 3D Object Generation Using Both 2D and 3D Diffusion Priors. In *ICLR*.

[85] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. 2021. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *ICCV*.

[86] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *CVPR*.

[87] Satyabrata Sarangi and Bevan M. Baas. 2021. DeepScaleTool: A Tool for the Accurate Estimation of Technology Scaling in the Deep-Submicron Era. In *ISCAS*.

[88] Andre Schollmeyer, Simon Schneegans, Stephan Beck, Anthony Steed, and Bernd Froehlich. 2017. Efficient hybrid image warping for high frame-rate stereoscopic rendering. In *TVCG*.

[89] Xinkai Song, Yuanbo Wen, Xing Hu, Tianbo Liu, Haoxuan Zhou, Husheng Han, Tian Zhi, Zidong Du, Wei Li, Rui Zhang, et al. 2023. Cambricon-r: A fully fused accelerator for real-time learning of neural scene representation. In *MICRO*.

[90] Zhuoran Song, Houshu He, Fangxin Liu, Yifan Hao, Xinkai Song, Li Jiang, and Xiaoyao Liang. 2024. SRender: Boosting Neural Radiance Field Efficiency via Sensitivity-Aware Dynamic Precision Rendering. In *MICRO*.

[91] Zhuoran Song, Chunyu Qi, Fangxin Liu, Naifeng Jing, and Xiaoyao Liang. 2024. Cmc: Video transformer acceleration via codec assisted matrix condensing. In *ASPLOS*.

[92] Zhuoran Song, Feiyang Wu, Xueyuan Liu, Jing Ke, Naifeng Jing, and Xiaoyao Liang. 2020. VR-DANN: Real-time video recognition via decoder-assisted neural network acceleration. In *MICRO*.

[93] Sony Interactive Entertainment (SIE). 2023. PlayStation VR2 Tech Specs. https://www.playstation.com/en-us/ps-vr2/ps-vr2-tech-specs/.

[94] Raúl Taranco, José-María Arnau, and Antonio González. 2023. δLTA: Decoupling Camera Sampling from Processing to Avoid Redundant Computations in the Vision Pipeline. In *MICRO*.

[95] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[96] Turki, Haithem and Ramanan, Deva and Satyanarayanan, Mahadev. 2022. Mega-NERF: Scalable Construction of Large-Scale NeRFs for Virtual Fly-Throughs. In *CVPR*.

[97] Varjo. 2024. Varjo Aero. https://varjo.com/products/aero/

[98] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. 2017. ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper. In *ICCAD*.

[99] Michael Waechter, Nils Moehrle, and Michael Goesele. 2014. Let there be color! Large-scale texturing of 3D reconstructions. In *ECCV*.

[100] Jialin Wang, Rongkai Shi, Zehui Xiao, Xueying Qin, and Hai-Ning Liang. 2022. Effect of render resolution on gameplay experience, performance, and simulator sickness in virtual reality games. In *PACMCGIT*.

[101] Jialin Wang, Rongkai Shi, Wenxuan Zheng, Weijie Xie, Dominic Kao, and Hai-Ning Liang. 2023. Effect of frame rate on user experience, performance, and simulator sickness in virtual reality. In *IEEE Transactions on Visualization and Computer Graphics*.

[102] Daniel N Wood, Daniel I Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H Salesin, and Werner Stuetzle. 2023. Surface light fields for 3D photography. In *Seminal Graphics Papers: Pushing the Boundaries*.

[103] Lizhou Wu, Haozhe Zhu, Siqi He, Jiapei Zheng, Chixiao Chen, and Xiaoyang Zeng. 2024. GauSPU: 3D Gaussian Splatting Processor for Real-Time SLAM Systems. In *MICRO*.

[104] 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 *HPCA*.

[105] Ziyu Ying, Shulin Zhao, Haibo Zhang, Cyan Subhra Mishra, Sandeepa Bhuyan, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2022. Exploiting frame similarity for efficient inference on edge devices. In *ICDCS*.

[106] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[107] Shulin Zhao, Haibo Zhang, Sandeepa Bhuyan, Cyan Subhra Mishra, Ziyu Ying, Mahmut T Kandemir, Anand Sivasubramaniam, and Chita R Das. 2020. Déja view: Spatio-temporal compute reuse for energy-efficient 360 vr video streaming. In *ISCA*.

[108] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-SoC Co-Design for Low-Power Mobile Continuous Vision. In *ISCA*.

A Artifact Appendix

A.1 Abstract

Our artifact provides scripts to reproduce the evaluation results of Neo and baseline 3DGS, including rendering latency, DRAM traffic, and accuracy. The artifact contains the source code and a public README that describes the exact commands to rebuild the environments, run experiments, and regenerate the figures and a table reported in the paper. Reproducing all results requires two execution environments: ❶ an NVIDIA Jetson Orin AGX system for on-device measurements, and ❷ an NVIDIA RTX-class server GPU system for Neo simulation. We provide dedicated script sets for each platform to facilitate their respective evaluations.

A.2 Artifact check-list

- **Algorithm:** 3D Gaussian Splatting (3DGS) with a reuse-and-update sorting algorithm.
- **Compilation:** Docker-based environments; one-time initialization scripts for installing all required dependencies such as Python, PyTorch, build tools, and supporting libraries.
- **Data set:** Tanks and Temples dataset: Family, Francis, Horse, Lighthouse, Playground, and Train, automatically downloaded via the provided scripts.
- **Model:** Pre-trained 3DGS models for all evaluated scenes, automatically downloaded via the provided scripts.
- **Hardware:** Two heterogeneous evaluation platforms, an edge device and a GPU server; ① NVIDIA Jetson Orin AGX 64GB, and ② an NVIDIA RTX-class GPU system. For the RTX-class system, evaluators need an Ampere-or-newer GPU with at least 24 GB memory, such as RTX 3090.
- **Run-time environment:** Docker with NVIDIA Container Toolkit; base images:
  - Orin AGX: dustynv/pytorch:2.7-r36.4.0-cu128-24.04
  - Server: pytorch/pytorch:2.7.0-cuda12.8-cudnn9-devel
- **Execution:** End-to-end bash scripts; initialization, experiments, result packaging, and figure generation.
- **Metrics:** Latency, accuracy (PSNR, LPIPS), and DRAM traffic.
- **Output:** Raw results, summarized figures and a table.
- **Experiments:** Figures 5, 10, 15–19 and Table 2 in the paper.
- **Required disk space:** 300 GB free storage.
- **Time to prepare workflow:** 1 hour.
- **Time to complete experiments:** 4 days.
- **Publicly available URL:** https://github.com/casys-kaist/Neo
- **Archived URL:** https://doi.org/10.5281/zenodo.17926773

A.3 Description

How to access. Clone our public GitHub repository:

```
$ git clone https://github.com/casys-kaist/Neo
```

Software dependencies. We distribute the artifact as Docker images. Therefore, evaluators need Docker and the NVIDIA Container Toolkit on both systems. We assume CUDA version ≥ 12.8 in the server environment as used by the provided base image. On Orin AGX, additional Python packages and build tools are installed via a provided initialization script. On the RTX server, Python dependencies are installed via `uv` and a provided initialization script.

Data set and models. We provide helper scripts that automatically download all required dataset and models.

A.4 Installation

To install the artifact, build the Docker images. We provide separate Dockerfiles for each platform. Execute the following steps on each host system.

❶ **Set up the environment.** Edit appropriately, replacing `{PLACEHOLDER}` with absolute paths on your system:

```
# For Orin AGX:  {REPOSITORY}/env/orin_env.sh
# For RTX 3090:  {REPOSITORY}/env/server_env.sh
NSYS_PATH={PLACEHOLDER}          # Nsight Systems
NCU_PATH={PLACEHOLDER}           # Nsight Compute
STORAGE_PATH={PLACEHOLDER}       # Storage
```

❷ **Build the Docker image.** From the repository root, build and launch the Docker image for the corresponding platform:

```
$ cd {REPOSITORY}/docker/orin   # For Orin AGX
$ cd {REPOSITORY}/docker/server # For RTX Server
$ ./build.sh
$ ./run.sh
```

❸ **Initialize the container.** Attach to the running container and perform one-time initialization inside it:

```
$ docker exec -it neo-ae-container /bin/bash
$ cd /workspace/docker/orin     # For Orin AGX
$ cd /workspace/docker/server   # For RTX server
$ . ./init.sh                   # ". ./init.sh"
```

❹ **Download datasets and models.** The `init.sh` script triggers downloads automatically. If any download fails, re-run the helper script:

```
$ cd /workspace/script
$ ./orin_download.sh            # For Orin AGX
$ ./server_download.sh          # For RTX server
```

A.5 Experiment workflow

All experiments are executed inside the containers. Each run generates intermediate outputs under `/mnt/output`.

① **Set up the environment.** Edit appropriately by replacing `{GPU}` with the GPU index you want to use:

```
# /workspace/script/env.sh
DATASET_PATH=/mnt/dataset       # Don't touch
MODEL_PATH=/mnt/model           # Don't touch
OUTPUT_PATH=/mnt/output         # Don't touch
CUDA_VISIBLE_DEVICES={GPU}      # GPU Index
```

② **Run experiments on Jetson Orin AGX.** To reproduce all Orin-side results end-to-end, run the script:

```
$ cd /workspace/script
$ ./orin_run.sh
```

Option. To reproduce a specific figure or table, run the script in its directory. For example, Figure 15:

```
$ cd /workspace/script
$ cd ./chapter6_evaluation/figure_15
```

```
$ ./orin_run.sh
```

③ **Run experiments on the RTX server.** To reproduce all server-side results end-to-end:

```
$ cd /workspace/script
$ ./server_run.sh
```

Option. To reproduce a specific figure or table, run the script in its directory (e.g., Figure 15):

```
$ cd /workspace/script
$ cd ./chapter6_evaluation/figure_15
$ ./server_run.sh
```

④ **Package and transfer results.** After finishing Orin runs, compress the outputs:

```
$ cd /workspace/script
$ ./orin_zip.sh
```

This produces `/mnt/output/orin.tar`. Copy `orin.tar` to the RTX server environment at the same path.

⑤ **Generate paper figures and tables.** On the RTX server, compress server outputs and generate plots:

```
$ cd /workspace/script
$ ./server_zip.sh
$ ./draw.sh
```

A.6 Evaluation and expected results

After running scripts, the artifact generates a summary under `/mnt/output/summary` with the expected results:

- Figure 5
  - figure_5_3dgs.pdf
  - figure_5_gscore.pdf
- Figure 10
  - figure_10_memory.pdf
  - figure_10_runtime.pdf
- Figure 15
  - figure_15.pdf
- Figure 16
  - figure_16.pdf
- Figure 17
  - figure_17_large_scale_scene.pdf
  - figure_17_rapid_camera_movement.pdf
- Figure 18
  - figure_18_runtime.pdf
  - figure_18_traffic.pdf
- Figure 19
  - figure_19_accuracy.pdf
  - figure_19_latency.pdf
- Table 2
  - table_2.csv

The PDF figures and CSV table correspond to the plots or a table reported in the paper. In addition to these final outputs, each experiment directory stores raw logs and intermediate CSV files under `/mnt/output`.

A.7 Notes

The README.md file in the public GitHub repository offers additional information on setting up the artifacts, detailed steps for running experiments, and summarizing results.