Lumina: Real-Time Neural Rendering by Exploiting Computational Redundancy

Yu Feng
Shanghai Jiao Tong University
Shanghai, Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, Shanghai, China
y-feng@sjtu.edu.cn

Weikai Lin
University of Rochester
Rochester, New York, USA
wlin33@ur.rochester.edu

Yuge Cheng
Shanghai Jiao Tong University
Shanghai, Shanghai, China
chengyuge@sjtu.edu.cn

Zihan Liu
Shanghai Jiao Tong University
Shanghai, Shanghai, China
altair.liu@sjtu.edu.cn

Jingwen Leng*
Shanghai Jiao Tong University
Shanghai, Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, Shanghai, China
leng-jw@cs.sjtu.edu.cn

Minyi Guo*
Shanghai Jiao Tong University
Shanghai, Shanghai, China
Shanghai Qi Zhi Institute
Shanghai, Shanghai, China
guo-my@sjtu.edu.cn

Chen Chen
Shanghai Jiao Tong University
Shanghai, Shanghai, China
chen-chen@sjtu.edu.cn

Shixuan Sun
Shanghai Jiao Tong University
Shanghai, Shanghai, China
sunshixuan@sjtu.edu.cn

Yuhao Zhu
University of Rochester
Rochester, New York, USA
yzhu@rochester.edu

Abstract

3D Gaussian Splatting (3DGS) has vastly advanced the pace of neural rendering, but it remains computationally demanding on today's mobile SoCs. To address this challenge, we propose Lumina, a hardware-algorithm co-designed system, which integrates two principal optimizations: a novel algorithm, $\mathcal{S}^2$, and a radiance caching mechanism, $\mathcal{RC}$, to improve the efficiency of neural rendering. $\mathcal{S}^2$ algorithm exploits temporal coherence in rendering to reduce the computational overhead, while $\mathcal{RC}$ leverages the color integration process of 3DGS to decrease the frequency of intensive rasterization computations. Coupled with these techniques, we propose an accelerator architecture, LuminCore, to further accelerate cache lookup and address the fundamental inefficiencies in Rasterization. We show that Lumina achieves 4.5× speedup and 5.3× energy reduction against a mobile Volta GPU, with a marginal quality loss (< 0.2 dB peak signal-to-noise ratio reduction) across synthetic and real-world datasets.

CCS Concepts

• **Computer systems organization** → **Neural networks**; **Real-time system architecture**; • **Computing methodologies** → **Rasterization**.

*Corresponding Authors.

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731003

Keywords

Mobile Architecture, Neural Rendering

ACM Reference Format:
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 *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731003

1 Introduction

Neural Radiance Fields (NeRF) [56] has transformed the landscapes of Virtual/Augmented Reality (VR/AR) [18, 33, 35, 61], large-scale landscape modeling [9, 52, 67, 73], virtual avatar [15, 37, 70], novel view synthesis [50, 77, 79], and beyond [29]. Although NeRF yields impressive results, its intensive computation is always a key bottleneck to achieving real-time and high-resolution rendering [14, 71].

To address this challenge, 3D Gaussian Splatting (3DGS) has been proposed as a fast alternative to the NeRF pipeline [40]. Unlike NeRF, which requires dense sampling along each ray, 3DGS projects precomputed Gaussian points onto the rendering screen, simplifying the color integration process and light transport modeling in 3D spaces. Despite numerous efforts [14, 20, 21, 40, 47, 71], 3DGS still falls short of the real-time requirement, e.g., 90 frames per second (FPS) [1, 2, 69], in AR/VR applications. For instance, on a mobile Volta GPU [5], 3DGS merely achieves 5 - 21 FPS on real-world scenes [10, 32, 42], far from the real-time target.

We aim to enhance the efficiency of 3DGS rendering in AR/VR applications. Our characterizations show that Sorting and Rasterization dominate the 3DGS rendering time, accounting for over 90% of the total execution (Sec. 2.2). To address the two main bottlenecks

in 3DGS, we introduce Lumina, a hardware-algorithm co-designed system in this paper.

Algorithms. We first propose a novel *Sorting-Shared* ($\mathcal{S}^2$) algorithm to address one of the bottlenecks, Sorting. $\mathcal{S}^2$ algorithm extends 3DGS by leveraging temporal redundancy in rendering (Sec. 3.1), sharing sorting results across multiple frames. In addition, $\mathcal{S}^2$ preemptively executes Sorting, effectively hiding Sorting execution time and mitigating frame stuttering. By doing so, Lumina reduces the Sorting computations without losing quality.

While $\mathcal{S}^2$ algorithm effectively hides Sorting latency in 3DGS, it does leave the other bottleneck, Rasterization, which is compute-intensive due to its color integration process. To accelerate Rasterization, we propose *Radiance Caching* ($\mathcal{RC}$), a lightweight caching mechanism that leverages the previous rendering results to approximate the current pixel values with a negligible compute overhead.

$\mathcal{RC}$ leverages a key insight: *two rays intersecting the same sequence of Gaussian points would likely yield the same pixel values* (Sec. 3.2). Specifically, we propose to cache a short record of "significant" ray-Gaussian intersections from previous renderings. Instead of executing the whole color integration, this short record allows us to perform only the initial segment of the color integration and determine the pixel values by matching with the cached record.

Fine-Tuning. However, directly applying $\mathcal{RC}$ to 3DGS occasionally introduces rendering artifacts. To address this, we propose end-to-end fine-tuning for our new pipeline to enhance rendering quality while maintaining the caching efficiency. Overall, our result shows that $\mathcal{RC}$ avoids 55% computation, on average, in color integration with minimal impact on quality.

Architectural Support. Coupled with two optimizations, we further propose a customized architecture, LuminCore (Sec. 4), to address the inherent computation inefficiencies in Rasterization. Our characterization shows that the color integration in Rasterization is inherently sparse, leading to low GPU utilization due to warp divergence. Radiance-cached Rasterization further exacerbates this divergence, transforming a full pixel rendering into a sparser one. Thus, we design dedicated Neural Rendering Units (NRUs) in LuminCore to tackle the sparse color integration in Rasterization. Along with NRUs, we co-design a specialized hardware cache, LuminCache, to speed up the cache lookup in $\mathcal{RC}$.

Result. By integrating Lumina into an off-the-shelf mobile SoC with negligible hardware overhead (0.4%), we show that Lumina achieves 4.5× speedup and 5.3× energy efficiency against a mobile Volta GPU, with a marginal quality loss.

Our contributions are summarized as follows:

- We introduce a plug-and-play $\mathcal{S}^2$ algorithm that leverages the temporal coherence in 3DGS to completely hide the Sorting overhead during real-time rendering.
- We propose a novel $\mathcal{RC}$ mechanism that caches ray-Gaussian intersections and reduces the computation of color integration in Rasterization by 55%.
- We propose a LuminCore architecture, co-designed to address the GPU warp divergence in Rasterization.
- We demonstrate that Lumina can achieve 4.5× speedup and 5.3× energy reduction against a mobile Volta GPU, with a marginal loss on visual quality.

2 Motivation

We begin by introducing 3DGS fundamentals in Sec. 2.1. Then, we identify the main inefficiencies of the 3DGS pipeline in Sec. 2.2.

2.1 3DGS Pipeline

Why 3DGS? Recent 3DGS [40] revolutionizes NeRF rendering by drastically accelerating the conventional NeRF rendering process. Conventional NeRF rendering is known for its computational intensity due to the extensive ray samplings. Each ray sample is required to perform a MLP operation.

In contrast, 3DGS proposes a reverse operation called "splatting", where Gaussian points (or "Gaussians" for short) are directly projected onto the rendering screen. This method sidesteps the compute-heavy task of ray-object intersections by reversing the workflow: rather than rays seeking Gaussians, Gaussians are directly mapped onto the screen.

To date, all 3DGS variants adhere to a *uniform* rendering process [14, 20, 21, 40, 47, 71]. The primary variation across different 3DGS algorithms is the training methodology, *not* the rendering process. Thus, this paper focuses on the original 3DGS rendering process [40] without loss of generality.

As Fig. 1 shows, 3DGS executes rendering tile-by-tile through three steps: *Projection*, *Sorting*, and *Rasterization*.

Projection. Given a camera pose, Projection serves two main purposes: first, it filters out Gaussians that fall outside the view frustum (e.g. grey ellipsoids), retaining only those Gaussians between the near- and far-clip planes (e.g. colored ellipsoids) as shown in Fig. 1; second, it projects each Gaussian, with a defined cutoff radius, onto the screen to determine its intersecting tiles.

Sorting. Once each tile collects its intersected Gaussian IDs, Sorting then determines the rendering order of those Gaussians, ensuring that all points are rendered according to their depth, from the closest to the furthest (relative to the camera pose), as shown in Sorted Splatting Table.

Rasterization. Once all Gaussians are sorted, Rasterization renders these Gaussians tile-by-tile. Every pixel within a tile would iterate the same set of Gaussians, calculating the transparency and integrating the color of those Gaussians to its pixel in the sorted order. For example, every pixel in tile T_0 integrates Gaussians in ❹ → ❷ → ❶ order. Eqn. 1 governs the color integration of pixel $\mathbf{p}$:

$$C(\mathbf{p}) = \sum_{i=1}^{N} \Gamma_i \alpha_i \mathbf{c}_i, \ \textit{where}\ \Gamma_i = \prod_{j=1}^{i-1}(1-\alpha_j) \tag{1}$$

where Γ_i denotes the accumulative transmittance of pixel $\mathbf{p}$ from the first Gaussian 1st to the $i-1$th. α_i and $\mathbf{c}_i$ stand for the transparency and the color at the ith Gaussian, respectively.

Note that, if the Gaussian's α_i value is smaller than $\frac{1}{255}$, this Gaussian will be skipped in the color integration process to avoid numerical instabilities, as shown in Fig. 1. The color integration terminates once the accumulative transmittance Γ_i falls below a predefined threshold, θ.

2.2 Performance Characterization

Model Size. To achieve photorealism, 3DGSs require dense Gaussians to reconstruct the scene. Fig. 2a shows the correlation between scene complexity and model size. Moving from small-scale

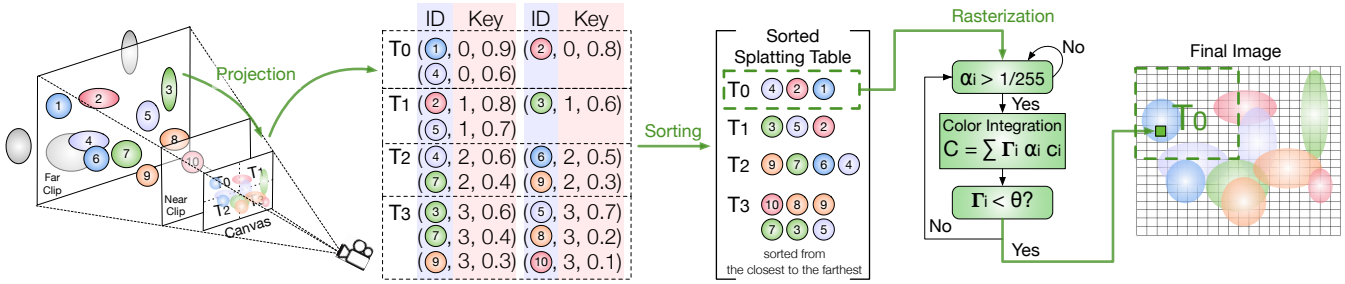
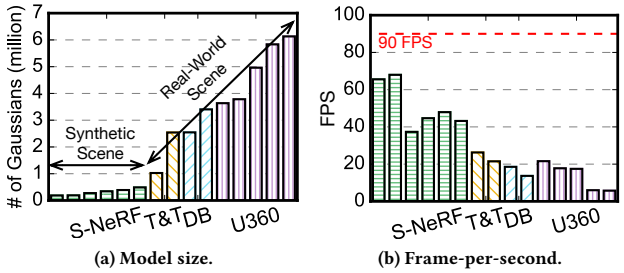


Fig. 1: The computation flow of today's 3DGS rendering pipeline is highlighted in green. Gaussian points are first projected onto the rendering screen to determine their intersection with image tiles (e.g., T_0, T_1). Next, each tile sorts the intersected points based on their depth values, from the nearest to the farthest. Finally, within each tile, Gaussian points are splatted onto the screen in the sorted order to render the final image governed by the color integration equation (Eqn. 1). Note that, only those Gaussians with significant transparency α_i ($\alpha_i > \frac{1}{255}$) will be integrated in the final pixel.



(a) Model size. (b) Frame-per-second.

Fig. 2: Model size and runtime performance trends with scene complexity. Rendering real-world scenes requires longer latencies, driven by their increased scene complexity. Performance measurements conducted across four key datasets: S-NeRF [56], T&T [42], DB [32], and U360 [10].

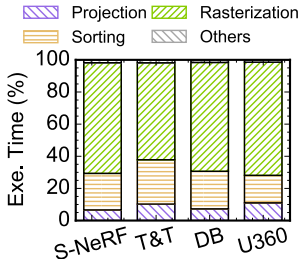


Fig. 3: Normalized execution breakdown across scenes. Sorting and Rasterization dominate the execution.

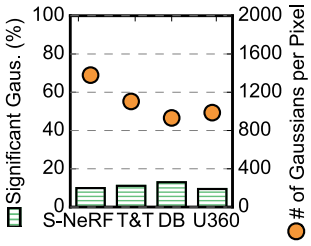


Fig. 4: The percentage of significant Gaussians per pixel and the average number of iterated Gaussians per pixel.

synthetic to real-world scenes, there is a significant increase in model size. Synthetic scenes in the S-NeRF dataset [56] contain relatively few Gaussians (less than 1 million). However, real-world datasets [10, 32, 42] increase the number of Gaussians over 10×, with the U360 dataset reaching over 6 million Gaussians.

Performance. For real-world scenes, the growth in model size poses computational challenges compared to their synthetic counterparts. As more detailed rendering is demanded in today's applications, increasingly large-scale models [41, 53] are necessitated for the intricate geometries and light interactions in the real world. Fig. 2b shows the rendering performance across datasets on a mobile Volta GPU on Nvidia Xavier SoC [1] [5]. As the rendering scenes transition from synthetic to real-world scenarios, the performance drops from 66 to 5 FPS, far below the real-time requirements of 90-100 FPS for AR/VR platforms [1, 2].

Execution Breakdown. To understand the main bottlenecks in 3DGS, Fig. 3 presents the normalized execution breakdown across four datasets. Overall, Sorting and Rasterization dominate the execution time, accounting for 23% and 67% on average, respectively.

[1]The performance of the Volta GPU (2.8 TFLOPS) is comparable to that of the GPU (3.5 TFLOPS) in the Snapdragon XR2 [8], released in September 2023 for VR platforms [7].

One thing worth mentioning, as the model size increases, no clear trend shows significant changes in the execution distribution. This means that the optimization focus will not shift as the scene scales.

Sparse Color Integration. With Rasterization dominating the overall execution time, we further pinpoint its performance bottleneck. By default, pixels within a tile are designed to iterate the same set of Gaussians, as shown in Fig. 1. However, Gaussians contribute to the final pixel in Eqn. 1 only if their transparencies, α, exceed $\frac{1}{255}$. We refer to these Gaussians as *significant Gaussians*. Fig. 4 characterizes the percentage of significant Gaussians versus the total Gaussians iterated per pixel across four datasets in Rasterization.

In Fig. 4, the left y-axis shows the percentage of significant Gaussians while the right y-axis shows the average number of total Gaussians per pixel. Despite each pixel iterating over a thousand of Gaussians, the percentage of significant Gaussians remains low, averaging only 10.3% with a standard deviation of 2.1%. This shows that each pixel is generated by only a small subset of Gaussians.

Warp Divergence. Due to the sparsity in the color integration (Fig. 4), different pixels might integrate different subsets of Gaussians. However, in typical GPU implementations, Rasterization is parallelized across pixels, assigning one thread to one pixel. The workload discrepancy across pixels would result in inefficient GPU utilization due to warp divergence in Fig. 5. Since threads are

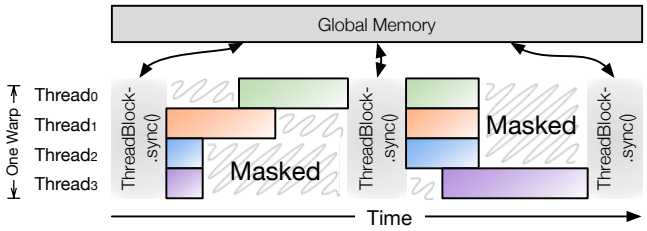


Fig. 5: Example of a warp execution during Rasterization, assuming 4 threads per warp. Colored blocks show that threads are not masked and perform meaningful computations. Between two computation periods, all threads load data from global memory and synchronize. Sparse color integration shown in Fig. 4 leads to severe warp divergence.

grouped into warps in modern GPU, all threads in a warp execute together in a *Single Instruction Multiple Thread* (SIMT) fashion.

The example in Fig. 5 shows a simplified GPU execution model with four threads within a warp. Here, one color represents one thread activity during rendering. The color integrations are interleaved with data synchronization, i.e., fetching Gaussians from global memory to shared memory. Since each thread requires different Gaussians, the GPU will mask threads that do not need to integrate specific Gaussians at that time. Our result shows that threads remain masked over 69% of the time across scenes with a standard deviation of 10%, showing low GPU utilization. This inefficiency underscores that GPUs are *ill-suited* for Rasterization.

Although prior studies propose various techniques to address GPU warp divergence [28, 55, 59], these methods are not suited for Rasterization in 3DGS. Techniques such as dynamic warp formation [28] and dynamic warp subdivision [55] rely on dynamic scheduling to mitigate warp divergence but inherently require synchronization, introducing additional runtime scheduling overhead. To overcome these limitations, we propose customized hardware for Rasterization (in Sec. 4) that eliminates synchronization overhead while maintaining high utilization of hardware resources.

3 LuminSys

This section presents our system, LuminSys. We first introduce two plug-and-play optimizations, *Sorting Sharing* (Sec. 3.1) and *Radiance Caching* (Sec. 3.2), to address two main bottlenecks, Sorting and Rasterization, respectively. We then explain how LuminSys integrates the optimizations to guarantee a smooth rendering (Sec. 3.3).

3.1 Sorting Sharing, $\mathcal{S}^2$

Intuition. The high-level idea of our $\mathcal{S}^2$ algorithm is to reuse the previous sorting result and bypass Sorting in 3DGS. The rationale here is that the sorting results tend to remain the same across consecutive poses. Fig. 6 gives an example. As the camera moves from pose M to pose N, the depth order of these Gaussians remains unchanged. Numbers in Fig. 6 show the depth order. The key observation here is that the sequence in which these Gaussians get rendered does not require frequent re-computation. This allows us to skip Sorting in adjacent frames.

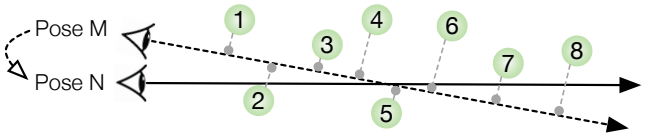


Fig. 6: The intuition of $\mathcal{S}^2$ algorithm. The rendering orders, a.k.a, the depth orders, of two spatially closed camera poses, M and N, are the same and can be reused.

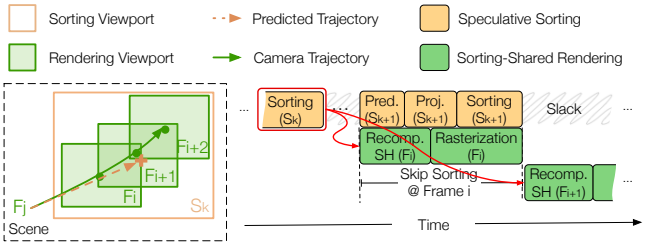


Fig. 7: Our $\mathcal{S}^2$ algorithm consists of two concurrent paths: *speculative sorting* and *sorting-shared rendering*. Speculative sorting predicts the future pose (e.g., S_k) and preemptively sorts Gaussians at that predicted pose. Note that, the sorting viewport at S_k requires to be large enough to accommodate all rendering ports in the sharing window, i.e., F_i, F_{i+1}, and F_{i+2}. Sorting-shared rendering later bypasses Sorting and performs Rasterization directly.

Even if these Gaussians' rendering order does change locally during rendering, the locally incorrect order is unlikely to impact the overall rendering results. This stability comes from the sparsity of color integration characterized in Sec. 2.2. Significant Gaussians (whose $\alpha > \frac{1}{255}$) of each pixel are likely separated apart after sorting, and their relative order will not change too much from one camera pose to another. Our experiment shows that only 0.2% of these Gaussian orders are changed.

Algorithm. One potential issue with a naive skip-sorting strategy is that we still need to perform Sorting in the middle of our rendering process. This inevitably introduces frame stuttering due to additional computation from Sorting. To hide the Sorting latency, we propose our $\mathcal{S}^2$ algorithm to keep a more consistent delay across frames. The key idea of $\mathcal{S}^2$ is to predict the pose trajectory and pre-compute the sorting results in advance.

Fig. 7 illustrates the workflow of our $\mathcal{S}^2$ algorithm when rendering a scene across different camera poses over time. Our $\mathcal{S}^2$ algorithm consists of two concurrent execution paths: *speculative sorting* and *sorting-shared rendering*.

Speculative Sorting. Speculative sorting is to predict the future camera pose and pre-sort Gaussians at this predicted pose before actual rendering. Fig. 7 highlights the speculative sorting in yellow. The red arrows highlight the data dependency between Sorting and Rasterization. At pose F_j, the $\mathcal{S}^2$ algorithm first predicts a future pose S_k using the current position and velocity at F_j. The velocity at F_j is approximated from the last two camera poses, F_{j-1} and F_j:

$$v_j = \frac{F_j - F_{j-1}}{\Delta t}, \tag{2}$$

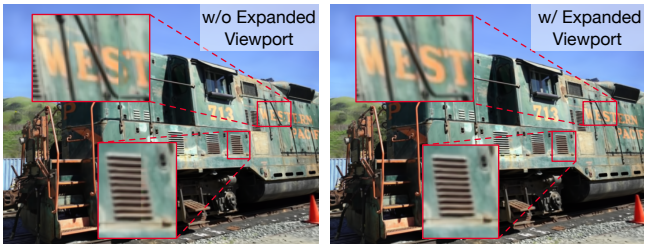


Fig. 8: Comparison of the rendering results with and without the expanded viewport. The one without the expanded viewport has noticeable artifacts at the tile edges (across 'S').

where Δt is the interval between two consecutive frames. The future pose S_k is then predicted using:

$$S_k = T_k + v \times t_r,\ t_r = \frac{N}{2}\Delta t, \tag{3}$$

where N is the number of rendered frames that share the same sorting result. We define N as *sharing window*. Using $\frac{N}{2}$ allows the predicted pose to be approximately at the center of its related frames, increasing the overlap of the sorted results with its rendered frames. Once the future pose of S_k is predicted, the $\mathcal{S}^2$ algorithm performs the Projection and Sorting steps and saves the sorting result for later use. Note that, the trajectory prediction is similar to that used in Cicero [25]. We do *not* claim this as our contribution.

Sorting-Shared Rendering. Fig. 7 colors the sorting-shared rendering in green. During rendering, as the camera moves to a new pose F_i, our $\mathcal{S}^2$ algorithm skips the repetitive Projection and Sorting steps by reusing the previous sorting result at the predicted pose S_k. One caution here is that, before Rasterization, each Gaussian color needs to be recalculated using pretrained Spherical Harmonic coefficients to preserve view-dependent rendering accuracy.

The subsequent frames (e.g., F_{i+1} and F_{i+2} in Fig. 7) within the same sharing window reuse the same sorting result and only perform sorting-shared rendering rather than the entire 3DGS pipeline. We empirically find that applying a fixed-size sharing window yields a good rendering quality, while prior works [31, 82] have exploited various size-adaptive strategies, which are not the *main* focus of this work. Our evaluation discusses the sensitivity of $\mathcal{S}^2$ to the window size in Sec. 6.3.

Expanded Viewport. During rendering, each pose is associated with a viewport that includes all the points that can potentially contribute to the pixels under that pose, as shown in Fig. 7. This means that the sorting viewport, S_k, needs to cover all the rendering viewports in its sharing window. If the sorting viewport is too small, Gaussians outside the sorting viewport can introduce rendering artifacts. As illustrated in the left part of Fig. 8, the rendered image on the left has artifacts across the letter 'S'.

To mitigate this issue, we expand the sorting viewport at the speculative sorting stage as outlined by the yellow box S_k in Fig. 7. The viewport at S_k is required to accommodate all viewports in the rendered frames, i.e., F_i, F_{i+1}, and F_{i+2}, in the sharing window. In practice, since 3DGS operates on a tile-by-tile basis, our $\mathcal{S}^2$ algorithm also expands the viewport at a tile granularity. Sec. 6.3 provides a quantitative analysis of how the expanded viewport impacts overall performance.

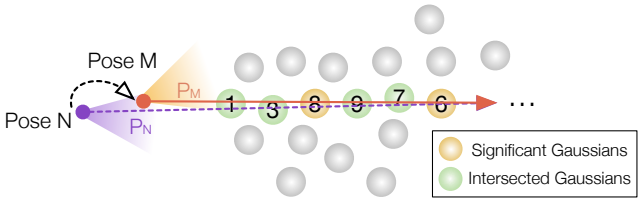


Fig. 9: The intuition behind radiance caching. The first six intersected Gaussians by pixel P_N and pixel P_M are the same, thus, their ray directions are similar. The previously rendered pixel P_N can be used to approximate pixel P_M.

3.2 Radiance Caching, $\mathcal{RC}$

While $\mathcal{S}^2$ can hide the execution time of Sorting, Rasterization itself is still the most dominant step in 3DGS. Here, we propose our technique, *radiance caching*, to reduce the computation in Rasterization. Although our technique shares the name with prior techniques in ray tracing [43, 58, 68], our technique fundamentally differs from conventional radiance cache in ray tracing as discussed in Sec. 7.

Intuition. We begin by explaining the intuition behind our technique. One fundamental property of 3DGS is that two rays will yield approximately the same pixel values if they meet two conditions: 1) they share the same direction, and 2) they intersect the same sequence of Gaussians.

The challenge lies in how to identify such rays and thus avoid redundant computations. Recall in geometry that any two distinct points on a straight line completely determine that line. Therefore, if two rays intersect with two distinct and sufficiently small Gaussians, these two rays satisfy the two requirements above and, thus, the pixel value of one ray can effectively approximate that of the other. The more Gaussians are intersected and shared by two rays, the more confident we are to approximate one using the other.

Fig. 9 illustrates our idea. Here, poses M and N are close to each other. In Fig. 9, for pixel P_N at pose N and pixel P_M at pose M, their first six intersected Gaussians are identical. Although the remaining thousands of Gaussian intersections for each pixel have not yet been computed, the similarity in their initial intersected Gaussians allows us to approximate one pixel value using the other. By saving the pixel value of P_N from pose N, the Rasterization step of P_M can safely terminate at the sixth Gaussian and use P_N's pixel value to approximate that of P_M.

Algorithm. Next, we explain our algorithm, radiance caching, using a toy example illustrated in Fig. 10:

❶ In the original 3DGS algorithm, each pixel iterates through a set of Gaussians and integrates Gaussians' contributions in order according to Eqn. 1. At the first frame 0, because the radiance cache is empty, our $\mathcal{RC}$ algorithm performs the exact Rasterization as in the original 3DGS, computing pixel values from scratch.

❷ After each pixel is computed, we concatenate the IDs of the first two intersected significant Gaussians (those with α values > $\frac{1}{255}$ highlighted in yellow in Fig. 10) as the cache tag, and use the computed pixel value serves as the cache value. Then, the radiance cache is updated accordingly in Fig. 10, where the initially empty cache is populated by four pixels. We later explain why we select significant Gaussians instead of all Gaussians for caching.

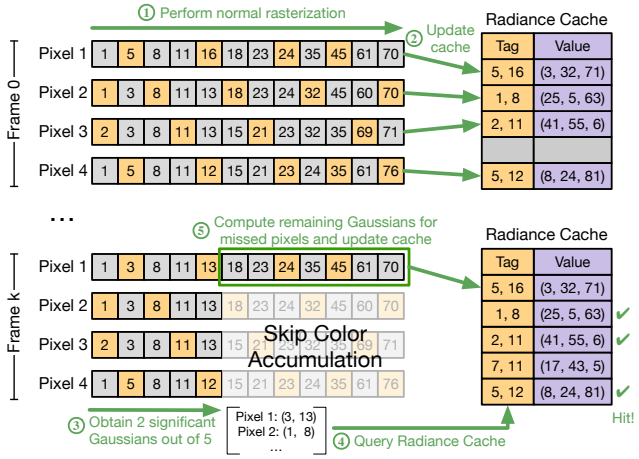


Fig. 10: The overall procedure of cached Rasterization step. At the initial frame, normal Rasterization is performed, and the radiance cache is populated. For each pixel, the cache tag is composed of the first two significant Gaussian IDs (with $\alpha > \frac{1}{255}$ shown in yellow), and the cache value is the pixel value. In subsequent frames, each pixel only needs to identify the first two significant Gaussians to query the radiance cache. Cache-hit pixels skip the remaining Rasterization. Cache-missed pixels continue the remaining Rasterization and update the radiance cache accordingly.

❸ After rendering the first frame and updating the radiance cache, we now can accelerate the rendering of subsequent frames using the radiance cache. For instance, when rendering frame k, each pixel computes the initial five Gaussians to identify the first two significant Gaussians.

❹ With the significant Gaussian IDs identified, each pixel concatenates these two Gaussian IDs as the cache tag and queries the radiance cache. If the tag matches any entry in the cache (e.g., for pixels 2, 3, and 4), those cache-hit pixels can directly use the cached pixel values, and their remaining color integration can be skipped. In this case, pixels 2, 3, and 4 bypass the color integration of the remaining seven Gaussians.

❺ For cache-missed pixels (e.g., pixel 1), they are still required to iterate through the remaining Gaussians to obtain the final pixel values. Once the color integration is completed, the radiance cache is then updated using the new values of these cache-missed pixels according to its cache policy.

Overall, our $\mathcal{RC}$ algorithm leverages the inherent color integration process in 3DGS and introduces a caching technique, which identifies similar rays during the initial process of Rasterization and eliminates redundant computations. Importantly, in our $\mathcal{RC}$ algorithm, only the first frame requires the complete 3DGS rendering, while all subsequent frames benefit from $\mathcal{RC}$.

Why Significant Gaussians? There are two reasons why we choose the first few significant Gaussian IDs as cache tags instead of any arbitrary Gaussian IDs.

First, significant Gaussians contribute more toward the final pixel values, making them an effective indicator of ray similarity.

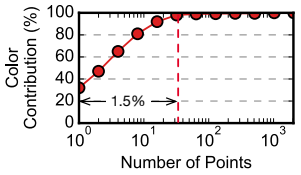


Fig. 11: The significance of Gaussian points towards the final radiance. Points are sorted by their contributions.

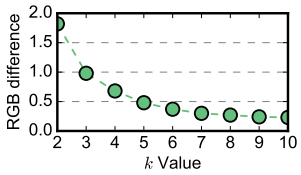


Fig. 12: The average color difference between Gaussians that share the same initial significant Gaussians, k.

Fig. 13: An example of the imprecise rendering without cache-aware fine-tuning. The artifact on the top of the train is due to large Gaussians.

As discussed in Sec. 2.2, only about 10% of Gaussians contribute to the final pixel value due to the sparsity of color integration. We further examine the significance of Gaussians towards the final pixel value in Fig. 11, where Gaussians are sorted by their contribution. The result shows that over 99% of the pixel value is derived from less than 1.5% of the Gaussians. This indicates that the remaining 98.5% of Gaussians have a negligible impact on rendering quality.

Second, we aim to use the fewest possible Gaussians to index the cache. Naturally, using initial significant Gaussians as a cache tag allows for a compact cache tag while avoiding the remaining color integration. We analyzed the average color difference between pixels that share the same initial significant Gaussians. In Fig. 12, increasing the number of initial significant Gaussians, k, reduces the color difference between pixels. Specifically, the average color difference remains below 1.0 when k is set to 3, and falls below 0.5 when k increases to 5. Both values are negligible compared to the maximum value of 255. This demonstrates that the initial significant Gaussians are a reliable metric for ray similarity.

General Applicability. Although we propose radiance caching to accelerate 3DGS, this technique is *not* limited to 3DGS alone. Radiance caching leverages the fundamental concept in neural rendering: representing scenes with trainable primitives (e.g., Gaussians in 3DGS or voxels in NeRF) and using rays to intersect with these primitives and integrate colors. Thus, even as neural rendering evolves toward new primitives, our technique remains beneficial, i.e., using the first k number of "new primitives" for caching, as long as this core rendering principle holds.

3.3 Putting it Together

Overview. Fig. 14 illustrates the overall workflow of LuminSys. Our system begins by taking the user's pose history to predict

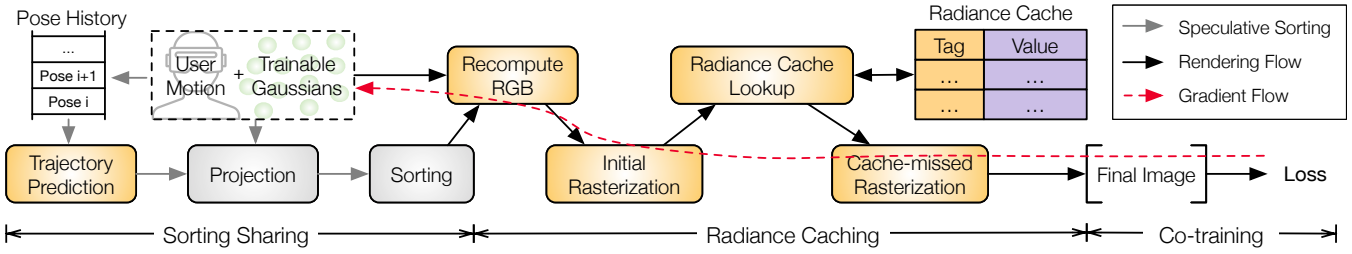


Fig. 14: The overall LuminSys system. Our $\mathcal{S}^2$ algorithm predicts the future pose and proactively computes sorting which is later shared across multiple frames. Meanwhile, our radiance caching accelerates the Rasterization step by exploiting similarities across rays. The framework is end-to-end trainable, with sorting and cache lookup untouched by gradient descent.

future camera poses. LuminSys then performs Projection and Sorting at the recently predicted pose and saves the sorting results for subsequent frames. When LuminSys receives a user pose update, it directly uses the recent sorting result and recomputes the RGB value of each Gaussian based on the current camera pose before Rasterization. During Rasterization, each ray identifies the first k significant Gaussians and queries the radiance cache using these Gaussian IDs. Cache-hit pixels would terminate Rasterization early and use the cached pixel values directly, whereas cache-missed pixels continue the full Rasterization process and update the radiance cache accordingly. The final image is constructed by combining both cache-hit and cache-missed pixels.

End-to-End Fine-Tuning. One key assumption of radiance caching is that the first few significant Gaussian points are sufficiently small and can be a good approximator of the ray similarity. In cases where the Gaussian points are too large, this assumption may lead to imprecise rendering as shown in Fig. 13. There are noticeable artifacts due to the large Gaussians on the top of the train. We propose a technique called cache-aware fine-tuning which is designed to enhance the accuracy of radiance caching in these cases. In particular, we introduce a scale-constrained loss function during training, which constrains the scale of each Gaussian:

$$L_{total} = L_{orig} + \alpha * L_{scale}(S, \theta) \quad (4)$$

where L_{orig} is the original 3DGS loss and L_{scale} is scale-constrained loss. S is the geometric mean of three Gaussian point scale parameters. L_{scale} penalties the geometric mean of any Gaussian point greater than θ. Sec. 6.1 shows the effectiveness of scale-constrained loss. It is worth mentioning that both sorting and cache lookup do not participate in gradient descent as the red dashed line shows. Therefore, the model is end-to-end differentiable even though sorting and cache lookup are not.

4 LuminCore

Why Accelerator? While radiance caching (Sec. 3.2) reduces computations in Rasterization, it cannot eliminate Rasterization. Recall, Sec. 2.2 shows that GPUs are fundamentally ill-suited for Rasterization due to warp divergence. Radiance-cached Rasterization exacerbates the divergence, transforming a dense pixel rendering into a sparse one in Fig. 15. The red dots highlight the cache-hit pixels, which are uniformly distributed across the scene. This further increases the percentage of masked threads after cache lookup.

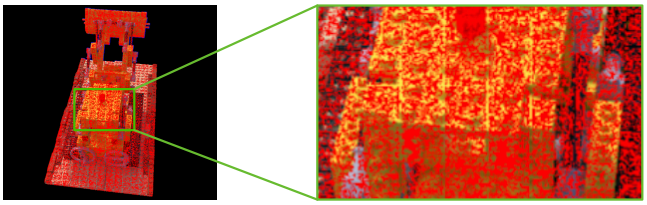


Fig. 15: An example of the sparse distribution of cache-hit pixels. The red dots highlight the cache-hit pixels which are uniformly distributed across the entire image.

In the current GPU's execution model, assigning one thread per pixel requires completed threads to wait until all threads in the warp finish, leading to severe GPU under-utilization. In addition to warp divergence, implementing radiance caching directly on GPU also introduces additional caching overhead and lock contentions. Our results in Sec. 6.2 will show that the GPU implementation of $\mathcal{RC}$ slows down Rasterization rather than speeding it up. Thus, we propose dedicated hardware to address the inefficiencies of sparse Rasterization, which is exacerbated by $\mathcal{RC}$, on GPUs.

Overview. Our LuminCore architecture is integrated with a mobile SoC to support the cached Rasterization. Fig. 17 describes our SoC architecture, comprising two primary components: a mobile GPU and our LuminCore. During rendering, Rasterization is delegated to LuminCore, while the mobile GPU executes Projection and Sorting. Fig. 17 highlights LuminCore (in color) from the baseline hardware.

LuminCore includes double-buffered Feature and Output Buffers, which store Gaussian features and the output pixel values, respectively. Meanwhile, LuminCore is coupled with a local cache, LuminCache, tailored for radiance caching. LuminCache is shared across multiple Neural Rendering Units (NRUs) to accelerate Rasterization.

LuminCache. Fig. 16 illustrates our cache architecture designed to accelerate the cache lookup in Sec. 3.2. Our cache design resembles a classic N-way set associative cache, with N set to 4 in this example. However, LuminCache differs in both the indexing method and the cache tag and value design.

Recall, $\mathcal{RC}$ algorithm requires identifying the first k significant Gaussians (k is set to 2 in this toy example) and using these Gaussian IDs to find the ray with the same initial Gaussian IDs in the cache. The index of each query is formed by concatenating the lower bits

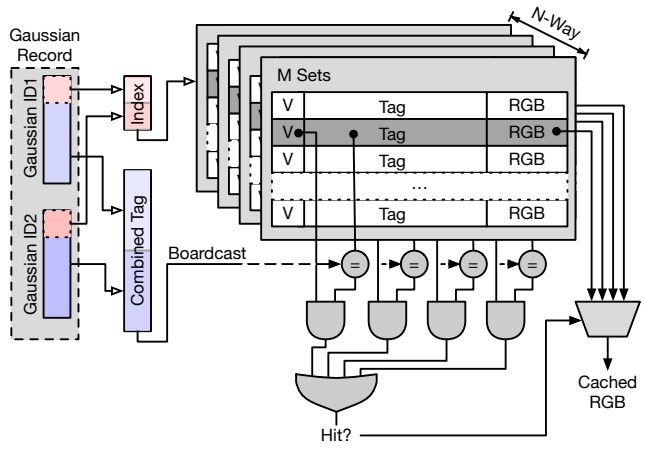


Fig. 16: Hardware support for radiance caching. The lower bits of the first k significant Gaussians are combined to form an index for accessing the cache. Meanwhile, their higher bits are concatenated to serve as a tag and compared against the tags stored within the cache. If a cache hit happens, the corresponding cached pixel value is returned. Here, k is set to 2 for illustration purposes.

of Gaussian IDs as shown in Fig. 16. The higher bits of all Gaussian IDs are concatenated to form a combined tag to validate the cache hit. If there is a cache hit, the radiance cache directly returns the cached RGB value to that pixel. In the case of a cache miss, the cache-missed pixel continues the remaining Rasterization. Once completed, the pixel updates the radiance cache according to a widely-used pseudo least-recently-used (LRU) cache policy [39].

Note that, we design the entire cache to be shared across 2×2 image tiles. Rendering the next batch of 2×2 tiles requires first saving the current cache data to memory, flushing the entire cache, and loading data related to the new batch from memory. When we render the same 2×2 tiles in the next frame, we first reload the corresponding cache data from the memory and then perform rasterization. In addition, our cache is designed to be double-buffered to hide the latency of data loading.

Neural Rendering Unit. GPU is inefficient for radiance-cached Rasterization primarily due to the warp divergence, as only a small number of Gaussians have low transparency and thus contribute to the color integration process. Prior 3DGS accelerators [24, 46] inherit the same inefficiency. To address this issue, our idea is to decouple transparency computation and color integration into two separate algorithmic stages, each of which is scheduled to fully utilize the corresponding hardware structures.

In particular, our NRU is divided into a frontend for calculating Gaussian transparency and a backend for performing color integration. The backend is designed to be shared across multiple processing elements (PEs) in the frontend so that we can ensure the full utilization of the backend despite the sparsity inherent in color integration.

The frontend is responsible for the lightweight computation that would apply to all Gaussians. It consists of a set of PEs, where each PE computes the transparency of Gaussians for each pixel. These PEs check whether the transparency is significant enough (i.e., $\alpha > 1/255$) to affect the pixel's final value, ensuring that only significant Gaussians would be inserted into a FIFO implemented by shift registers for color integration.

The detailed design of PE is described in Fig. 18. Overall, each PE is implemented as a three-stage pipeline. Each PE is implemented with three multipliers and three multiply-and-add (MAC) units to compute the α values of Gaussians. Additionally, each PE uses a comparator and a sign checker to assess whether the Gaussian's α is significant enough to grant subsequent processing.

The backend, which is shared among multiple PEs, is for compute-intensive but sparse color integration that is applied only to significant Gaussians, including dedicated components for computing exponent, etc. Each time, the backend takes one Gaussian from the FIFO and performs color integration for its corresponding pixels. The backend also contains a set of register files (α-records) to cache significant Gaussian IDs for different pixels which will be used for radiance cache lookup. This frontend-backend design improves PE utilization by leveraging the sparsity of significant Gaussians.

Sparsity-Aware Remapping. Recall that radiance caching saves computations of cache-hit pixels, while the cache-miss pixels still require full rasterization. Fig. 15 shows that these cache-miss pixels are sparsely distributed across the entire image. Retaining the same pixel-to-PE mapping, where each PE is responsible for a single pixel, would lead to PE under-utilization, because the cache-hit PEs would remain idle after cache lookup. To improve the PE utilization, we allow the NRUs to be reconfigured so that all the PEs within a single NRU can collaborate on rendering a *single* pixel.

For all the PEs to collaboratively render the same pixel in this mode, the PEs read different Gaussians (that belong to a pixel/tile) from a sorted Gaussian list in order, instead of reading the same Gaussian (to process different pixels in parallel). Then, the PEs would write the per-Gaussian intermediate results (e.g., transparency, Gaussian ID) into the shift registers in order if the Gaussian transparencies are so high that require further processing by the backend. The backend executes in the same way as in the normal mode. By doing so, we avoid PE under-utilization once some pixels are terminated early by radiance caching.

SoC Integration. LuminCore is a standalone SoC IP block and can be integrated well with different GPU architectures. The data communication between LuminCore and a GPU is via standard SoC-level interconnect (e.g., AXI). In our case, there is no direct interaction between LuminCore and the GPU. LuminCore only read data from DRAM through DMA. Thus, our design is agnostic to and can accommodate different GPU architectures.

Broad Applicability. Note that, LuminCore accelerates the general field of point-based neural rendering, which has wide applicability not only in rendering but also in areas like SLAM [54, 75] and beyond. With the recent rise of neural rendering, rendering primitives are rapidly evolving from voxels [56] to 3D Gaussians [40] and 2D Gaussians [36]. Despite this rapid development, the overall rendering process, i.e., color accumulation, stays the same. Our LuminCore holds the potential to be used beyond 3DGS, as we extend to new rendering primitives. In addition, LuminCore essentially performs parallel and sparse accumulation, a computing

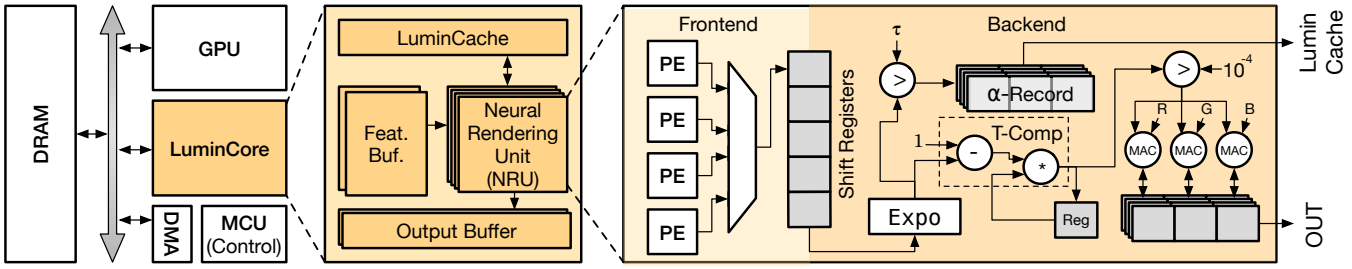


Fig. 17: The overall SoC architecture design. We integrate our LuminCore into a mobile SoC architecture. The uncolored is the baseline architecture. LuminCore consists of four main components: a LuminCache, a set of Neural Rendering Units (NRUs), a double-buffered Feature, and an Output Buffer. NRUs execute Rasterization and are designed in a dual-segment fashion to support both dense and sparse populated Rasterization.

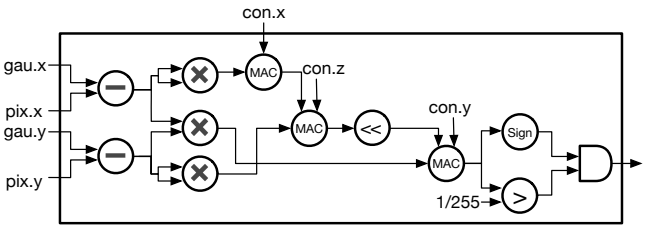


Fig. 18: The detailed architecture of a three-stage PE. Each PE calculates the exponential decay and assesses the significance of each Gaussian before sending it to the backend.

paradigm that is also prevalent in other domains such as sparse linear algebra [19], graph neural networks [72], and more. By making LuminCore programmable, we can support other domains as well.

5 Experimental Setup

Hardware Setup. The LuminCore has 8×8 NRUs clocked at 1 GHz, each consists of four three-stage PEs. The feature buffer, shared across all NRUs, is double-buffered with a total size of 176 KB. The output buffer is also double-buffered with a size of 6 KB. Each NRU includes a 160 B shift register array to store temporal α values of significant Gaussians, with 88 B of register files to store α records.

The LuminCache is shared among the NRUs. This cache is designed as a 4-way associative cache, comprising 4×1024 entries with a total size of 52 KB. The tag of each cache entry is constructed by the IDs of 5 significant Gaussians (using the 3rd to 18th least significant bits of each Gaussian ID, 10 bytes in total), with the corresponding RGB color values as cache values. Overall, LuminCache can cache 64×64 pixels to share across 4×4 image tiles with a size of 16×16. LuminCache is also double-buffered to hide the latency of loading cached values from previous renderings.

Experimental Methodology. The performance of the GPU, including kernel launch times, is directly measured on the mobile Volta GPU in Nvidia's Xavier SoC. Power metrics are also directly obtained using the built-in power measurement features of the device [5]. The LuminCore 's datapath is developed using an EDA process that includes synthesis, placement, and routing with Synopsys and Cadence tools on TSMC's 16 nm FinFET technology. These results are then scaled to the 12 nm node of Nvidia's Xavier SoC using DeepScaleTool to be compatible with the Nvidia GPU [62, 66]. SRAM components are produced using the Arm Artisan memory compiler, with power estimates determined via Synopsys PrimeTimePX with annotated switching activities.

Simulation Methodology. We simulate the entire system with our cycle-accurate simulator, which is implemented with component-level latencies and power measurements. The latency and energy of NRU are obtained from the post-synthesis results of its RTL design and scaled down to 12 nm node using DeepScaleTool [62] to match the mobile Volta GPU on Nvidia Xavier SoC [5]. The latency of GPU execution is directly measured, including kernel launch time. The GPU power consumption is obtained via the built-in power measurement on Nvidia Xavier SoC.

The DRAM model in our simulation is based on Micron's 16 Gb LPDDR3-1600, utilizing four channels according to its datasheet [3], with energy consumption sourced from Micron System Power Calculators [4]. The energy ratio between a random DRAM access and an SRAM access is about 25:1 aligned with prior work[30, 76].

The system energy is the sum of GPU, LuminCore, and DRAM. The overall latency is derived from the combined execution of GPU, NRU, and DRAM. Note that, the total latency excludes the parallel execution between sorting on GPU and rasterization on NRU. Due to the double buffering in NRU and LuminCache, the overall latency is dominated by the compute latency, not memory.

Area Overhead. Lumina introduces minimal area overhead with the LuminCore design, primarily due to the 8×8 NRU array. The area overhead, amounting to 1.05 mm^2, is negligible when compared to the entire mobile SoC area, which is approximately 350 mm^2 for Nvidia's Xavier SoC [6].

Datasets. To evaluate the efficiency and robustness of $\mathcal{S}^2$ and $\mathcal{RC}$, we evaluate both synthetic and real-world scenes.

For synthetic scenes, we select four out of eight scenes from Synthetic-NeRF (S-NeRF) [56]. We use the raw Blender files to generate videos and simulate a typical VR scenario with the average head rotation of 25 degrees at 90 FPS [1, 2, 34, 69].

In addition, we use Tanks&Temples (T&T) [42], a real-world dataset, from which we choose four sequences for our experiments. We extract a 10-second video clip from the raw video sequence in each sequence and use COLMAP [64], a well-known photogrammetry tool, to generate the necessary camera poses for our evaluations. Note that, the raw videos in T&T are captured at 30 FPS, much

lower than the 90 FPS typically used in VR scenarios. The rendering quality of $\mathcal{S}^2$ and $\mathcal{RC}$ is assessed using Peak Signal-to-Noise Ratio (PSNR), SSIM, and LPIPS as the standard metrics. We cannot evaluate our techniques on MipNeRF360 (U360) [10] and DeepBlending (DB) [32] datasets used in the original 3DGS paper because they contain individual images, *not* continuous video sequences.

Hardware Baselines. We compare two hardware baselines in our evaluation. One is the mobile Volta GPU in Nvidia's Xavier SoC. The other, NRU+GPU, is the SoC architecture in Fig. 17, but excluding LuminCache. Both baselines execute the full-fledged 3DGS algorithm. NRU+GPU executes Projection and Sorting on GPU while executing Rasterization on NRU.

Variants. To dissect our contributions in algorithm and architecture, we evaluate five variants of Lumina to separate the contributions proposed in our paper:

- $\mathcal{S}^2$-GPU: executes the $\mathcal{S}^2$ algorithm on a mobile Volta GPU with radiance caching disabled.
- $\mathcal{RC}$-GPU: executes the original 3DGS algorithm with $\mathcal{RC}$ mechanism on a mobile Volta GPU.
- $\mathcal{S}^2$-Acc: executes the $\mathcal{S}^2$ algorithm on our proposed architecture with radiance caching disabled.
- $\mathcal{RC}$-Acc: executes the original 3DGS algorithm with $\mathcal{RC}$ mechanism on our proposed architecture.
- Lumina: the full-fledged Lumina, with both $\mathcal{S}^2$ algorithm and $\mathcal{RC}$ mechanism on our architectures.

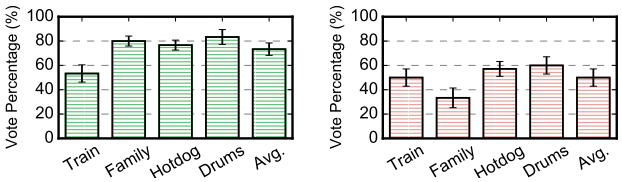
User Study. We also conduct a user study to assess the subjective rendering quality of our techniques, the procedure is approved by our Internal Review Board (IRB). Our user study includes 30 participants (20 males and 10 females; age 18-30). No participants were aware of the research objectives, experimental hypothesis, or the number of conditions. All participants had normal or corrected-to-normal vision. Each participant reviews 4 traces in our evaluation. We use the classic Two-Interval Forced Choice (2IFC) procedure [11]. For each trace, we display the renderings of these two methods side-by-side on a 16-inch monitor in a randomized order to each participant, with 30 seconds to rest and answer questions. We ask two questions to each participant:

- Whether they notice any difference between these two.
- If they observe any difference, pick one that they prefer.

If participants do not notice any difference, they are required to choose the version they prefer. Each trace is repeated three times, and the trace order is also randomized each time.

6 Evaluation

We first demonstrate that Lumina achieves comparable rendering quality against the baseline both qualitatively and quantitatively (Sec. 6.1). Next, we show the speedup and energy reduction of Lumina against two hardware baselines (Sec. 6.2). We then conduct a sensitivity study on Lumina (Sec. 6.3). Finally, we show that Lumina performs better against GSCore [46] (Sec. 6.4). Finally, we show that Lumina achieves better performance compared to the state-of-the-art 3DGS accelerator, GSCore [46] (Sec. 6.4).



(a) Question one: any difference between two versions. Numbers show participants who notice no difference. **(b) Question two: pick one you prefer. The percentages show participants who prefer our method.**

Fig. 19: User study of Lumina compared against original 3DGS. Among 27% of participants who notice differences, we achieve a 50%-50% tie.

6.1 Rendering Quality

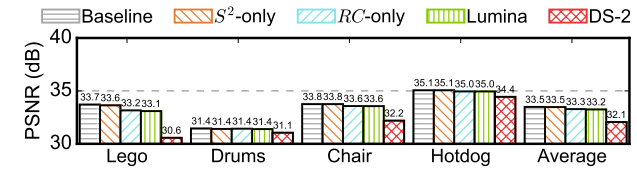
Subjective Evaluation. Fig. 19a shows the percentage of votes that do not notice any difference from our user study. On average, over 73% of users do not notice any difference between our method and the baseline 3DGS. Fig. 19b shows the percentage of votes that notice any difference and prefer our method. Among the remaining 27% of participants who noticed differences, 50% preferred the renderings by Lumina. This shows that our rendering method has an equal visual quality compared to the baseline 3DGS. We also set up a website to show our rendering results: link.

Quantitative Evaluation. Fig. 20 compares the rendering quality of various methods on both synthetic and real-world scenes. We use *sharing window* to denote the number of frames that share a single sorting result. We use *expanded margin* to denote the number of pixels that the sorting viewport expands at each dimension (in Sec. 3.1). Unless specified otherwise, the sharing window is set to be 6, with an expanded margin of 4. In addition to the baseline algorithm, we compare against a variant, DS-2, which first renders a 2× downsampled frame and then upsamples it to the original resolution by 2. $\mathcal{S}^2$-only and $\mathcal{RC}$-only denote only apply $\mathcal{S}^2$ algorithm and $\mathcal{RC}$ mechanism, respectively.

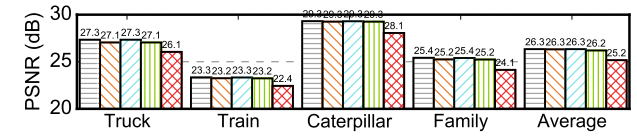
Synthetic Scenes. Fig. 20a shows the rendering quality on synthetic scenes. Overall, $\mathcal{S}^2$-only demonstrates robust performance, matching the baseline in accuracy while only requiring sorting once every 6 frames. Both $\mathcal{RC}$-only and Lumina can maintain a similar accuracy across all scenes, with minimal quality losses of 0.2 dB and 0.3 dB, respectively. In Comparison, DS-2 generally shows a substantial drop in PSNR, with an average 1.4 dB accuracy drop.

Real Scenes. Fig. 20b shows the quality evaluation on real scenes. Unlike the synthetic scenes, the real-world scenes have a lower frame rate (30 FPS) than a typical VR scenario (90 FPS), resulting in larger inter-frame movements. Consequently, $\mathcal{S}^2$-only shows a slight decrease in rendering quality by 0.1 dB compared with the baseline. Interestingly, $\mathcal{RC}$-only achieves the same accuracy as the baseline, showing the resilience of radiance caching even at lower frame rates. Overall, Lumina manages to align closely with the baseline. In contrast, DS-2 is 1.0 dB lower than the baseline, vastly underperforming compared with other variants.

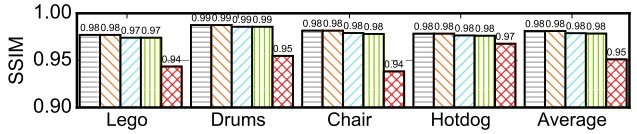
Fig. 20c to Fig. 20f show the rendering quality of synthetic and real-world scenes on the other two metrics, SSIM and LPIPS. The overall trend still holds. Lumina can achieve a similar rendering quality compared to the corresponding baselines.



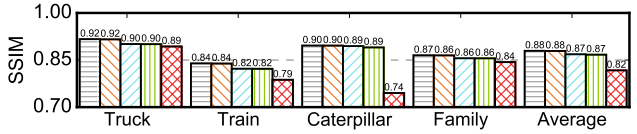
(a) Synthetic scenes on PSNR. Higher is better.



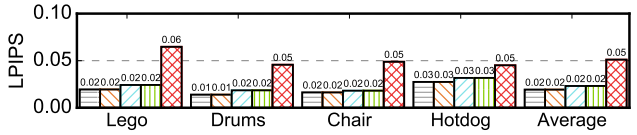
(b) Real scenes on PSNR. Higher is better.



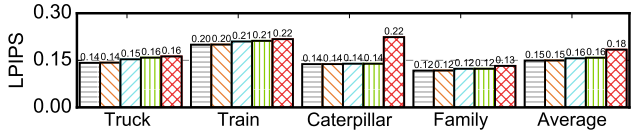
(c) Synthetic scenes on SSIM. Higher is better.



(d) Real scenes on SSIM. Higher is better.



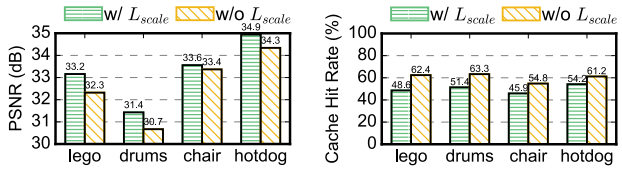
(e) Synthetic scenes on LPIPS. Lower is better.



(f) Real scenes on LPIPS. Lower is better.

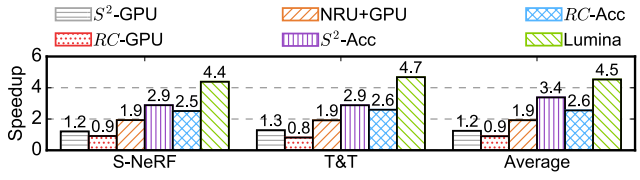
Fig. 20: Image quality comparison. Both $\mathcal{S}^2$-ONLY and LUMINA configure the expanded margin (i.e., the number of pixels that the sorting viewport expands) of 4 and the skipping window (i.e., the number of frames share a single sorting result) of 6. DS-2 first renders a 2× downsampled frame and then upsamples it to the original resolution by 2.

Cache-Aware Finetuning. Fig. 21 compares the effects of incorporating a scale-constrained loss, L_{scale}, in Sec. 3.3 on both the rendering quality and cache hit rate on $\mathcal{RC}$-ONLY. Introducing L_{scale} yields a notable improvement in PSNR across various scenes, with an average improvement of 0.6 dB. Including L_{scale} marginally decreases the cache hit rate, which might lead to slight performance

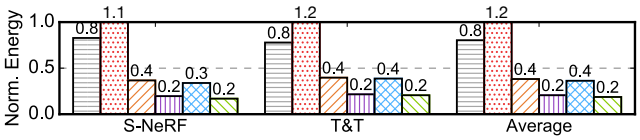


(a) Quality evaluation. (b) Cache hit rate.

Fig. 21: Rendering quality and cache hit rate of $\mathcal{RC}$-ONLY with and without scale-constrained loss, L_{scale}, in Sec. 3.3.



(a) Normalized speedup. Higher is better.



(b) Normalized energy. Lower is better.

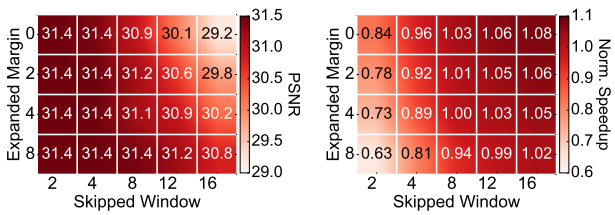
Fig. 22: Speedup and normalized energy consumptions of different variants over GPU baseline described in Sec. 5.

degradation, as shown in Fig. 21b. The decrease in cache hit rate is attributed to more stringent constraints on Gaussian point scaling imposed by L_{scale}, leading to fewer cache hits. A quantitative study of the relationship between cache hit rate and performance is presented in Sec. 6.3.

6.2 Performance and Energy

Performance. Fig. 22a shows the normalized speedup of different variants compared to the GPU baseline across scenes. With pure software implementation of $\mathcal{S}^2$ and $\mathcal{RC}$, $\mathcal{S}^2$-GPU can achieve 1.2× speedup by skipping Projection and Sorting. In comparison, $\mathcal{RC}$-GPU slows down the overall rendering process despite achieving over 50% cache hit rate. This slowdown is primarily attributed to GPU warp divergence, as discussed in Sec. 4, where the inefficiencies in sparse Rasterization negate the benefits of caching.

With the integration of NRU, NRU+GPU yields 1.9× speedup against the GPU baselines by accelerating the Rasterization stage. On average, our LUMINCORE accelerates the Rasterization stage itself by 6.4×. On top of that, $\mathcal{S}^2$-Acc further enhances performance by skipping the execution of the Projection and Sorting stages, boosting the performance to 3.1×. Since NRU+GPU already reducing the execution latency of Rasterization, $\mathcal{RC}$-Acc shows a moderate speedup on top of NRU+GPU. The overall speedup of $\mathcal{RC}$-Acc ranges from 1.7× to 2.7× by leveraging LUMINCACHE to reduce computational redundancy further. On the Rasterization



(a) Quality evaluation. (b) Normalized speedup.

Fig. 23: Sensitivity of rendering quality and performance to the expanded margin and skipped window of the $\mathcal{S}^2$ algorithm on the 'Drums' scene in SyntheticNeRF dataset. The speedup is normalized to $\mathcal{S}^2$-ONLY with an expanded margin of 4 and a skipped frame of 6.

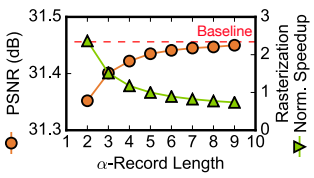


Fig. 24: The sensitivity of rendering quality and normalized speedup to the number of significant Gaussians.

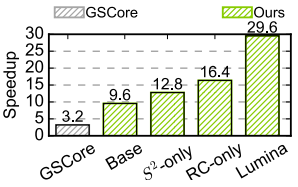


Fig. 25: Comparison of speedup between LUMINA and GSCore. Values are normalized to the GPU baseline.

stage alone, $\mathcal{RC}$-ACC can achieve 2.5× speedup. Together, LUMINA achieves 4.5× speedup against the GPU baseline across scenes. On average, LUMINA achieves 218.5 FPS and 97.9 FPS on synthetic and real-world scenes, respectively.

Energy Reduction. The normalized energy consumption of various variants are illustrated in Fig. 22b. Similar to the performance result, $\mathcal{S}^2$-GPU achieves 20% of energy saving while $\mathcal{RC}$-GPU introduces an additional 20% energy overhead. By solely leveraging the LUMINCORE architecture, NRU+GPU already demonstrates 62% of energy reduction, drastically reducing the latency of Rasterization. With Rasterization being accelerated, $\mathcal{S}^2$-ACC outperforms $\mathcal{RC}$-ACC in terms of energy savings, achieving 79% and 64% energy efficiency, respectively. Overall, by integrating all techniques, LUMINA achieves energy savings by 81%, highlighting the synergistic benefits of combining $\mathcal{S}^2$ algorithm and $\mathcal{RC}$ mechanism. Note that, only LUMINA achieves real-time (90 FPS) on the real-world dataset. If we set the performance target to be real-time, the energy savings of LUMINA would be 93% and 80% over the baseline on the synthetic and real-world scenes, respectively.

6.3 Sensitivity Study

Fig. 23 illustrates the sensitivity of rendering quality and normalized speedup to two algorithmic configurations, expanded viewport and skipped window, on the *drums* scene from the SyntheticNeRF dataset. We use *expanded margin* to denote the number of pixels that the sorting viewport expands at each dimension (in Sec. 3.1). The speedup is normalized to $\mathcal{S}^2$-ONLY with an expanded margin of 4 and a skipped window of 6.

Expanded Viewport. Fig. 23a shows that, as the expanded margin increases, there is an improvement in rendering quality. For instance, with a skipped window of 8, the rendering quality starts at 30.9 dB (with an expanded margin of 2) and increases to 31.4 dB as the expanded margin expands to 8. However, increasing the expanded margin results in a speedup decrease, as shown in Fig. 23b. Initially, at an expanded margin of 2, the speedup ranges from 0.8 to 1.1× compared to the baseline. As the expanded margin increases to 8, the speedup diminishes to 0.6 - 1.0×, indicating a trade-off between efficiency and quality.

Skipped Window. Meanwhile, as the number of skipped frames increases, the rendering quality tends to decrease. For example, at the expanded margin of 4, PSNR drops from 31.4 dB to 30.2 dB when the number of skipped frames increases from 2 to 16. Conversely, the speedup increases with more skipped frames. For instance, at the expanded margin of 2, the speedup enhances from 0.8 to 1.1× with more frames being skipped.

α-Record Length. Fig. 24 exploits the sensitivity of rendering quality and normalized speedup to variations in α-record length, ranging from 1 to 10. Recall, α-record stores the IDs of significant Gaussians. By default, α-record length is 5. Here, we normalize the speedup to the α-record length of 5 and only show the normalized speedup in terms of the Rasterization stage.

As the α-record length increases, the rendering quality, represented by the orange circles, gradually increases but eventually plateaus, aligning with the baseline (red dashed line) at 31.4 dB. Meanwhile, the normalized speedup of Rasterization, shown by green triangles, shows a gradual decrease from 2.3× to 0.7×. This decrease in speedup is primarily due to a reduction in cache hit rate, which falls from 82% to 31%, and an increase in the computational workload before the cache lookup as the α-record length extends.

6.4 Comparison with GSCore

Fig. 25 compares the speedup of LUMINA against the state-of-the-art accelerator, GSCore [46]. For a fair comparison, we incorporate the dedicated accelerator units: Culling& Conversion Unit (CCU) and Gaussian Sorting Unit (GSU) from GSCore. In our baseline hardware, projection and sorting are executed on CCU and GSU, respectively, rather than being performed on the GPU.

In Fig. 25, the results are obtained from both synthetic and real-world datasets, with all values normalized to the GPU baseline. Notably, LUMINA significantly outperforms GSCore across all variants. Even our baseline hardware (9.6×) can achieve better performance against GSCore (3.2×), thanks to the frontend-backend design of our LUMINCORE. Most of the Gaussian points can avoid unnecessary color integration. On top of that, $\mathcal{S}^2$-ONLY and $\mathcal{RC}$-ONLY can reach 12.8× and 16.4×. Here, $\mathcal{RC}$-ONLY achieves a higher speedup compared to $\mathcal{S}^2$-ONLY as Rasterization now becomes the dominant stage. With all the techniques combined, LUMINA can achieve 29.6× speedup against the GPU baseline.

7 Related Work

Neural Rendering Acceleration. In recent years, neural rendering has gained significant attention, leading to the development of

numerous accelerators specifically designed for neural rendering algorithms [24, 25, 27, 44, 48, 49, 51, 57, 60]. Despite that, most prior designs focus exclusively on NeRF and not on 3DGS, which remains less explored. Some proposed software-based acceleration techniques, such as pruning [20, 21] and quantization [47], seek to enhance the 3DGS performance but still fall short of achieving real-time performance on mobile devices. By far, GScore [46] remains the only accelerator dedicated to 3DGS.

This paper introduces techniques that are generally applicable across different 3DGSs and are orthogonal to the approaches used in GScore. Our caching can potentially extend its applications beyond 3DGS to enhance NeRFs.

Radiance Caching. Prior studies [43, 58, 63, 68] have explored radiance caching to accelerate ray tracing, but their approach differs fundamentally from ours. Their techniques primarily focus on caching radiance samples from ray-object intersections. Their primary focus is to reduce the caching overhead [63, 68] and enable the lightweight computation to radiance interpolation from cached samples using spherical harmonics [43] or neural network [58].

Applying conventional radiance caching to 3DGS would introduce significant storage overhead with no computational savings, as 3DGS does not require multi-bounce irradiance collection [38]. Our radiance caching leverages the 3DGS characteristics, where a single ray intersects multiple Gaussians with no bounce. This makes Lumina a natural fit to accelerate 3DGS.

Temporal Correlation. Prior studies leverage the temporal correlations by classic image warping techniques [13, 16, 17]. Recent hardware-algorithm co-design generalize warping to use temporal correlations across frames for reducing computation in real-time vision [12, 22, 23, 26, 45, 65, 74, 78, 80–82]. However, these techniques typically exploit task-specific characteristics and cannot be directly applicable to neural rendering. The most recent work, Cicero [25], requires known object meshes and applies warping techniques to NeRFs but at a cost of rendering quality. In contrast, our techniques, $\mathcal{S}^2$ and $\mathcal{RC}$, boost performance with a marginal quality loss.

8 Discussion and Limitations

As with all prior work [12, 26, 65, 78, 82] that leverages the temporal correlations, a pathological case with rapid head rotations would be detrimental to the performance of $\mathcal{S}^2$. To avoid catastrophic cases, we can simply disable $\mathcal{S}^2$ by detecting the rapid rotation data from IMU. Nevertheless, under common scenarios, our results show that $\mathcal{S}^2$ is effective in real-world cases (Fig. 20).

GPUs have a tradition of incorporating custom hardware to support new rendering paradigms such as ray tracing. We set out to understand whether custom hardware can enable real-time 3DGS, an emerging rendering paradigm. Starting from a clean architectural slate allows us to fully exploit the algorithmic characteristics of 3DGS without being constrained by the current GPU design. Although it is important to investigate improvements to the GPU architecture for 3DGS and revisit classical techniques that tackle issues like warp divergence [28, 55, 59], the value of work lies in demonstrating how custom hardware for 3DGS might look like *if* it is warranted. While the jury is still out, we argue that a dedicated accelerator could have its value in freeing up the GPU resources for other rendering/computation workloads, as future mobile SoCs will undoubtedly execute multiple workloads that exercise the GPUs.

9 Conclusions

Neural rendering has revolutionized the landscape of VR/AR and photo-realistic rendering lately, offering unprecedented realism. 3DGS emerges as a promising alternative to the conventional rasterization pipeline. It is important to rethink and develop brand-new architectures tailored to this future rendering technology.

This paper exploits a key insight, Gaussian-ray intersection in 3DGS, to design a caching mechanism applicable to 3DGS pipelines. With our framework, LuminSys, we achieve a 4.5× speedup with minimal yet principle hardware augmentations. By integrating our design into a dedicated neural rendering accelerator, Lumina can further boost the performance up to 30×.

References

[1] [n. d.]. Apple Vision Pro screen refresh rate is up to 100Hz. https://appleinsider.com/articles/24/01/19/apple-vision-pro-rate-is-up-to-100hz-it-has-bluetooth-53-and-more-technical-details

[2] [n. d.]. Meta Quest Pro specs. https://vr-compare.com/headset/metaquestpro

[3] [n. d.]. Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/mobile-dram/low-power-dram/lpddr3/178b_8-16gb_2c0f_mobile_lpddr3.pdf

[4] [n. d.]. Micron System Power Calculators. https://www.micron.com/support/tools-and-utilities/power-calc

[5] [n. d.]. NVIDIA Reveals Xavier SOC Details. https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/

[6] [n. d.]. NVIDIA's Xavier System-on-Chip, HotChips 30. https://fuse.wikichip.org/news/1618/hot-chips-30-nvidia-xavier-soc/

[7] [n. d.]. Qualcomm Powers Next-Gen Spatial Computing With XR2 Gen 2 And AR1 Gen 1 Platforms. https://www.forbes.com/sites/moorinsights/2023/09/27/qualcomm-powers-next-gen-spatial-computing-with-xr2-gen-2-and-ar1-gen-1-platforms/

[8] [n. d.]. Qualcomm QCS8550/QCM8550 Processors. https://docs.qualcomm.com/bundle/publicresource/87-61717-1_REV_A_Qualcomm_QCS8550_QCM8550_Processors_Product_Brief.pdf

[9] 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.

[10] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. 2022. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. *CVPR* (2022).

[11] Rafal Bogacz, Eric Brown, Jeff Moehlis, Philip Holmes, and Jonathan D Cohen. 2006. The physics of optimal decision making: a formal analysis of models of performance in two-alternative forced-choice tasks. *Psychological review* 113, 4 (2006), 700.

[12] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. 2018. EVA^2: Exploiting temporal redundancy in live computer vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 533–546.

[13] Gaurav Chaurasia, Arthur Nieuwoudt, Alexandru-Eugen Ichim, Richard Szeliski, and Alexander Sorkine-Hornung. 2020. Passthrough+ real-time stereoscopic view synthesis for mobile mixed reality. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 1 (2020), 1–17.

[14] Guikun Chen and Wenguan Wang. 2024. A survey on 3d gaussian splatting. *arXiv preprint arXiv:2401.03890* (2024).

[15] Mingfei Chen, Jianfeng Zhang, Xiangyu Xu, Lijuan Liu, Yujun Cai, Jiashi Feng, and Shuicheng Yan. 2022. Geometry-guided progressive nerf for generalizable and efficient neural human rendering. In *European Conference on Computer Vision*. Springer, 222–239.

[16] Shenchang Eric Chen. 1995. Quicktime VR: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 29–38.

[17] Shenchang Eric Chen and Lance Williams. 2023. View interpolation for image synthesis. In *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 423–432.

[18] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. 2023. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field

rendering on mobile architectures. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16569–16578.

[19] Iain S Duff, Michael A Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)* 28, 2 (2002), 239–267.

[20] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. 2023. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *arXiv preprint arXiv:2311.17245* (2023).

[21] Guangchi Fang and Bing Wang. 2024. Mini-Splatting: Representing Scenes with a Constrained Number of Gaussians. *arXiv preprint arXiv:2403.14166* (2024).

[22] Yu Feng, Nathan Goulding-Hotta, Asif Khan, Hans Reyserhove, and Yuhao Zhu. 2022. Real-time gaze tracking with event-driven eye segmentation. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, 399–408.

[23] Yu Feng, Patrick Hansen, Paul N Whatmough, Guoyu Lu, and Yuhao Zhu. 2023. Fast and Accurate: Video Enhancement Using Sparse Depth. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 4492–4500.

[24] 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. *ACM Transactions on Architecture and Code Optimization* (2024).

[25] 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. *arXiv preprint arXiv:2404.11852* (2024).

[26] Yu Feng, Paul Whatmough, and Yuhao Zhu. 2019. Asv: Accelerated stereo vision system. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 643–656.

[27] 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 *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–12.

[28] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 407–420.

[29] Kyle Gao, Yina Gao, Hongjie He, Dening Lu, Linlin Xu, and Jonathan Li. 2022. Nerf: Neural radiance field in 3d vision, a comprehensive review. *arXiv preprint arXiv:2210.00379* (2022).

[30] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[31] Shangchen Han, Beibei Liu, Randi Cabezas, Christopher D Twigg, Peizhao Zhang, Jeff Petkau, Tsz-Ho Yu, Chun-Jung Tai, Muzaffer Akbay, Zheng Wang, et al. 2020. MEgATrack: monochrome egocentric articulated hand-tracking for virtual reality. *ACM Transactions on Graphics (ToG)* 39, 4 (2020), 87–1.

[32] 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.

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

[34] PL Hendicott, B Brown, KL Schmid, and S Fisher. 2002. Head movement amplitude and velocity during a common visual task. *Investigative Ophthalmology & Visual Science* 43, 13 (2002), 4668–4668.

[35] Tao Hu, Shu Liu, Yilun Chen, Tiancheng Shen, and Jiaya Jia. 2022. Efficientnerf efficient neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12902–12911.

[36] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2024. 2d gaussian splatting for geometrically accurate radiance fields. In *ACM SIGGRAPH 2024 conference papers*. 1–11.

[37] Wei Jiang, Kwang Moo Yi, Golnoosh Samei, Oncel Tuzel, and Anurag Ranjan. 2022. Neuman: Neural human radiance field from a single video. In *European Conference on Computer Vision*. Springer, 402–418.

[38] Nathaniel L Jones and Christoph F Reinhart. 2016. Parallel Multiple-Bounce Irradiance Caching. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 57–66.

[39] Kamil Kędzierski, Miquel Moreto, Francisco J Cazorla, and Mateo Valero. 2010. Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[40] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics* 42, 4 (2023), 1–14.

[41] Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. 2024. A hierarchical 3d gaussian representation for real-time rendering of very large datasets. *ACM Transactions on Graphics (TOG)* 43, 4 (2024), 1–15.

[42] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. *ACM Transactions on Graphics* 36, 4 (2017).

[43] Jaroslav Krivánek, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch. 2005. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics* 11, 5 (2005), 550–561.

[44] Junseo Lee, Kwanseok Choi, Jungi Lee, Seokwon Lee, Joonho Whangbo, and Jaewoong Sim. 2023. NeuRex: A Case for Neural Rendering Acceleration. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[45] Junseo Lee, Jaisung Kim, Junyong Park, and Jaewoong Sim. 2025. VR-Pipe: Streamlining Hardware Graphics Pipeline for Volume Rendering. *arXiv preprint arXiv:2502.17078* (2025).

[46] 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.

[47] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. 2023. Compact 3d gaussian representation for radiance field. *arXiv preprint arXiv:2311.13681* (2023).

[48] 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 *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[49] 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 *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–13.

[50] Ruofan Liang, Huiting Chen, Chunlin Li, Fan Chen, Selvakumar Panneer, and Nandita Vijaykumar. 2023. ENVIDR: Implicit Differentiable Renderer with Neural Environment Lighting. *arXiv preprint arXiv:2303.13022* (2023).

[51] Weikai Lin, Yu Feng, and Yuhao Zhu. 2025. MetaSapiens: Real-Time Neural Rendering with Efficiency-Aware Pruning and Accelerated Foveated Rendering. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 669–682.

[52] David B Lindell, Dave Van Veen, Jeong Joon Park, and Gordon Wetzstein. 2022. Bacon: Band-limited coordinate networks for multiscale scene representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 16252–16262.

[53] Yang Liu, He Guan, Chuanchen Luo, Lue Fan, Junran Peng, and Zhaoxiang Zhang. 2024. Citygaussian: Real-time high-quality large-scale scene rendering with gaussians. *arXiv preprint arXiv:2404.01133* (2024).

[54] Hidenobu Matsuki, Riku Murai, Paul HJ Kelly, and Andrew J Davison. 2024. Gaussian splatting slam. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18039–18048.

[55] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture*. 235–246.

[56] 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.

[57] Muhammad Husnain Mubarik, Ramakrishna Kanungo, Tobias Zirr, and Rakesh Kumar. 2023. Hardware Acceleration of Neural Graphics. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*. 1–12.

[58] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. 2021. Real-time neural radiance caching for path tracing. *arXiv preprint arXiv:2106.12372* (2021).

[59] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 308–317.

[60] Chaolin Rao, Huangjie Yu, Haochuan Wan, Jindong Zhou, Yueyang Zheng, Minye Wu, Yu Ma, Anpei Chen, Binzhe Yuan, Pingqiang Zhou, et al. 2022. ICARUS: A Specialized Architecture for Neural Radiance Fields Rendering. *ACM Transactions on Graphics (TOG)* 41, 6 (2022), 1–14.

[61] Sara Rojas, Jesus Zarzar, Juan C Pérez, Artsiom Sanakoyeu, Ali Thabet, Albert Pumarola, and Bernard Ghanem. 2023. Re-ReND: Real-time Rendering of NeRFs across Devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 3632–3641.

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

[63] Daniel Scherzer, Chuong H Nguyen, Tobias Ritschel, and Hans-Peter Seidel. 2012. Pre-convolved radiance caching. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1391–1397.

[64] Johannes Lutz Schönberger and Jan-Michael Frahm. 2016. Structure-from-Motion Revisited. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[65] 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 *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 698–710.

[66] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* 58 (2017), 74–81.

[67] Matthew Tancik, Vincent Casser, Xinchen Yan, Sabeek Pradhan, Ben Mildenhall, Pratul P Srinivasan, Jonathan T Barron, and Henrik Kretzschmar. 2022. Block-nerf: Scalable large scene neural view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8248–8258.

[68] K Vardis, G Papaioannou, and A Gkaravelis. 2014. Real-time radiance caching using chrominance compression. *Journal of Computer Graphics Techniques Vol* 3, 4 (2014).

[69] 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. *IEEE Transactions on Visualization and Computer Graphics* 29, 5 (2023), 2478–2488.

[70] Chung-Yi Weng, Brian Curless, Pratul P Srinivasan, Jonathan T Barron, and Ira Kemelmacher-Shlizerman. 2022. Humannerf: Free-viewpoint rendering of moving people from monocular video. In *Proceedings of the IEEE/CVF conference on computer vision and pattern Recognition*. 16210–16220.

[71] Tong Wu, Yu-Jie Yuan, Ling-Xiao Zhang, Jie Yang, Yan-Pei Cao, Ling-Qi Yan, and Lin Gao. 2024. Recent Advances in 3D Gaussian Splatting. *arXiv preprint arXiv:2403.11134* (2024).

[72] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[73] Yuanbo Xiangli, Linning Xu, Xingang Pan, Nanxuan Zhao, Anyi Rao, Christian Theobalt, Bo Dai, and Dahua Lin. 2022. Bungeenerf: Progressive neural radiance field for extreme multi-scale scene rendering. In *European conference on computer vision*. Springer, 106–122.

[74] Lei Xiao, Salah Nouri, Matt Chapman, Alexander Fix, Douglas Lanman, and Anton Kaplanyan. 2020. Neural supersampling for real-time rendering. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 142–1.

[75] Chi Yan, Delin Qu, Dan Xu, Bin Zhao, Zhigang Wang, Dong Wang, and Xuelong Li. 2024. Gs-slam: Dense visual slam with 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 19595–19604.

[76] Amir Yazdanbakhsh, Kambiz Samadi, Nam Sung Kim, and Hadi Esmaeilzadeh. 2018. GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks.

[77] Weicai Ye, Shuo Chen, Chong Bao, Hujun Bao, Marc Pollefeys, Zhaopeng Cui, and Guofeng Zhang. 2023. Intrinsicnerf: Learning intrinsic neural radiance fields for editable novel view synthesis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 339–351.

[78] 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 *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1073–1084.

[79] Jian Zhang, Yuanqing Zhang, Huan Fu, Xiaowei Zhou, Bowen Cai, Jinchi Huang, Rongfei Jia, Binqiang Zhao, and Xing Tang. 2022. Ray priors through reprojection: Improving neural radiance fields for novel view extrapolation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18376–18386.

[80] 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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 241–253.

[81] Shulin Zhao, Haibo Zhang, Cyan Subhra Mishra, Sandeepa Bhuyan, Ziyu Ying, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita Das. 2021. HoloAR: On-the-fly optimization of 3D holographic processing for augmented reality. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 494–506.

[82] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. *arXiv preprint arXiv:1803.11232* (2018).