SPLATONIC: Architectural Support for 3D Gaussian Splatting SLAM via Sparse Processing

Xiaotong Huang^{1,†}, He Zhu^{1,†}, Tianrui Ma^{3}, Yuxiang Xiong^{1}, Fangxin Liu^{1}
Zhezhi He^{1}, Yiming Gan^{3}, Zihan Liu^{1,2,*}, Jingwen Leng^{1,2}, Yu Feng^{1,2,*}, Minyi Guo^{1,2}
^{1}Shanghai Jiao Tong University, ^{2}Shanghai Qi Zhi Institute
^{3}Institute of Computing Technology, Chinese Academy of Science
{hxt0512, zhcon16, xiongyuxiang, liufangxin, zhezhi.he, altair.liu, leng-jw, y-feng, myguo}@sjtu.edu.cn
{matianrui, ganyiming}@ict.ac.cn
†Equal contribution, *Corresponding authors
Project site: https://stonesix16.github.io/splatonic/

***Abstract*—3D Gaussian splatting (3DGS) has emerged as a promising direction for SLAM due to its high-fidelity reconstruction and rapid convergence. However, 3DGS-SLAM algorithms remain impractical for mobile platforms due to their high computational cost, especially for their tracking process.**

This work introduces SPLATONIC, a sparse and efficient real-time 3DGS-SLAM algorithm-hardware co-design for resource-constrained devices. Inspired by classical SLAMs, we propose an adaptive sparse pixel sampling algorithm that reduces the number of rendered pixels by up to 256× while retaining accuracy. To unlock this performance potential on mobile GPUs, we design a novel pixel-based rendering pipeline that improves hardware utilization via Gaussian-parallel rendering and preemptive α-checking. Together, these optimizations yield up to 121.7× speedup on the bottleneck stages and 14.6× end-to-end speedup on off-the-shelf GPUs. To further address new bottlenecks introduced by our rendering pipeline, we propose a pipelined architecture that simplifies the overall design while addressing newly emerged bottlenecks in projection and aggregation. Evaluated across four 3DGS-SLAM algorithms, SPLATONIC achieves up to 274.9× speedup and 4738.5× energy savings over mobile GPUs and up to 25.2× speedup and 241.1× energy savings over state-of-the-art accelerators, all with comparable accuracy.

***Index Terms*—3D Gaussian Splatting, SLAM, Accelerator.**

I. INTRODUCTION

Simultaneous localization and mapping (SLAM) is a key component to intelligent automation across various applications [7], [10], [35], [67], [76]. Among SLAM algorithms, 3D Gaussian splatting (3DGS)-based algorithms [36], [56], [61], [78], [81] have recently emerged as a promising direction due to their superior reconstruction fidelity and fast convergence [6], [72], [74], [84], over their alternatives. For example, iterative closest point (ICP)-based methods [31], [59], [83] often struggle in low-texture and poor lighting environments. Neural radiance field (NeRF)-based methods [26], [33], [65], [85] suffer from high compute costs and slow rendering.

Nevertheless, 3DGS-SLAM algorithms remain constrained in mobile applications due to their substantial computation overheads. The state-of-the-art 3DGS-SLAM algorithms often fail to achieve real-time on off-the-shelf mobile SoCs [1], [4], [5]. For instance, the average frame rate of a 3DGS-SLAM algorithm, SplaTAM [36], is only 0.1 Hz on a mobile Ampere GPU [1], far from real-time (10-30 Hz). This gap between the computation demands and the hardware capability motivates the need for an acceleration solution on mobile devices.

In general, 3DGS-SLAM algorithms consist of two concurrent processes: *tracking* and *mapping*. Tracking estimates the camera pose of each frame based on the reconstructed scene, while mapping reconstructs unseen regions of the scene by generating new 3D Gaussian primitives. These two processes often operate at different frequencies: tracking runs in a per-frame manner to ensure the pose accuracy; and mapping is invoked less frequently, usually every 4-8 frames. Our experiment shows that the amortized per-frame latency of mapping is only one-quarter that of tracking (Fig. 4). Thus, our work primarily focuses on accelerating the tracking process.

Idea. Although tracking and mapping serve different purposes, both share the same differentiable rendering pipeline and rely on training to obtain accurate results. Among all stages in this pipeline, *rasterization* is the primary bottleneck in both forward and backward training passes (see Fig. 5), accounting for 94.7% of the execution time, due to the need to iterate over every pixel of a frame in rasterization. Inspired by classical SLAM approaches [31], [59], [83], which accelerate localization by detecting key features, we explore whether the similar sparsity-based principle can apply to 3DGS-SLAMs. Since the workload of rasterization is proportional to the number of processed pixels, processing fewer pixels could dramatically reduce the overall computation cost [37].

Thus, we propose an adaptive sparse sampling algorithm in Sec. IV-A that can select important pixels at runtime to reduce the overall computation cost. Based on the characteristics of tracking and mapping, we tailor sampling strategies for each process. In tracking, we find that a simple yet effective approach, uniform random sampling, is sufficient to preserve pose estimation accuracy, while reducing the number of processed pixels by 256×. In mapping, on the other hand, our sampling algorithm prioritizes pixels in unseen regions or areas with rich textures to preserve reconstruction quality. Overall, we show that our sampling algorithm achieves the best accuracy compared to existing sampling strategies.

Pipeline. However, directly applying our sparse sampling algorithm to the existing 3DGS pipeline achieves merely a 4.2$\times$ speedup on rasterization, far below the compute savings (256$\times$). The fundamental reason is that current 3DGS pipelines [14], [32], [37], [75] all adopt a *tile-based* rendering, which exploits the data sharing across pixels in rasterization to amortize the compute cost of its earlier stages, i.e., projection and sorting. However, this rendering paradigm is inherently *ill-suited* to sparse pixel rendering, as sparsely distributed pixels offer little opportunity for data sharing and result in low hardware utilization on both GPUs and dedicated accelerators.

Thus, we design a *pixel-based* rendering pipeline in Sec. IV-B. Compared to tile-based rendering, our pipeline has two key advantages. First, we explicitly perform pixel-level projection that eliminates the unnecessary computation of subsequent stages that would otherwise result from the data sharing of tile-based rendering. Second, we propose a Gaussian-parallel rasterization, where multiple processing elements (PEs) co-render a single pixel, rather than assigning one pixel per PE. Our Gaussian-parallel rasterization largely improves the PE parallelism. In addition to our pixel-based rendering, we further propose *preemptive α-checking*, a technique to eliminate the warp divergence across PEs and avoid the unnecessary computations after the projection stage. With all optimizations together, we boost the rasterization performance up to 121.7$\times$ on an off-the-shelf GPU in Sec. VII-B.

Architectural Support. With rasterization no longer the bottleneck, we find that the main bottlenecks shift to the projection in the forward pass and the aggregation in the backward pass (Fig. 3). To address new bottlenecks, we propose a clean-slate pipelined architecture in Sec. V. Specifically, we introduce a lightweight rasterization engine that simplifies the rendering logic of rasterization in both the forward and backward passes. Furthermore, we augment a projection unit to mitigate the increased computational overhead by preemptive α-checking. Finally, we design a specialized aggregation unit to address the frequent pipeline stalls due to aggregation.

Result. We evaluate SPLATONIC on four popular 3DGS-SLAM algorithms. With our sampling algorithm and pixel-based rendering, SPLATONIC achieves 14.6$\times$ end-to-end speedup and 86.1% energy reduction on the mobile Ampere GPU [1] with comparable accuracy against the baseline algorithms. With hardware support, SPLATONIC achieves up to 274.9$\times$ speedup and 4738.5$\times$ energy savings against the GPU baseline, while achieving up to 25.2$\times$ speedup and 241.1$\times$ energy savings against the prior accelerators [29], [77]. Even with the same sparse sampling algorithm, SPLATONIC still can achieve up to 12.7$\times$ speedup and 200.8$\times$ energy savings against prior accelerators [29], [77].

The contributions of this paper are as follows:

- We propose a sparse pixel sampling algorithm for 3DGS-SLAMs that achieves up to 256$\times$ pixel reduction in tracking with an even better task accuracy.
- We introduce a pixel-based rendering pipeline that improves the parallelism in sparse pixel processing and achieves up to 121.7$\times$ speedup on the bottleneck stages.
- We co-design a pipelined accelerator architecture, which further addresses the remaining bottlenecks in pixel-based rendering and further improves performance.

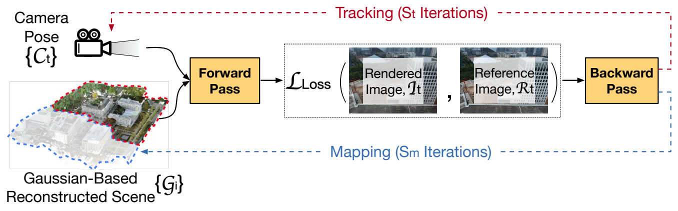

Camera Pose $\{C_t\}$
Tracking (S_t Iterations)
Forward Pass
$\mathcal{L}$Loss
Rendered Image, $\mathfrak{I}_t$
Reference Image, $\mathfrak{R}_t$
Backward Pass
Mapping (S_m Iterations)
Gaussian-Based Reconstructed Scene $\{G_i\}$

Fig. 1. Overview of 3DGS-SLAM process. Tracking and mapping share the same optimization pipeline with different optimization targets. Tracking optimizes camera poses $\{C_t\}$ while mapping reconstructs the scene $\{G_i\}$.

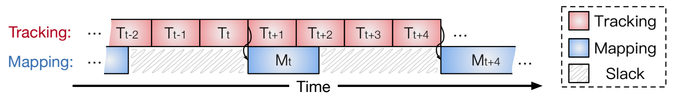


Fig. 2. The timing diagram of 3DGS-SLAM process. Tracking often runs more frequently compared to mapping. Mapping, M_t, at the same time, t, needs to be executed after tracking, T_t, due to the dependency.

II. BACKGROUND

A. 3DGS-based SLAM

Overview. The overall goal of SLAM is to reconstruct the scene of an unknown environment while simultaneously estimating the agent's trajectory within that scene. 3DGS-SLAM algorithms [36], [56], [61], [81] achieve this by leveraging 3DGS rendering pipelines, as illustrated in Fig. 1. The overall process can be divided into two concurrent processes: *tracking* and *mapping*. These two processes jointly optimize two sets of trainable parameters: the camera trajectory, $\{C_t\}$, and a set of Gaussian points, $\{G_i\}$, that represent the reconstructed scene. We next describe these two processes separately.

Tracking. The goal of tracking is to estimate each camera pose, C_t, along the trajectory. During tracking, we assume that the current reconstructed scene is valid, as shown by the red dashed block in Fig. 1. Under this assumption, we fix the trainable parameters of the Gaussian representation, $\{G_i\}$, and render an image, I_t, at the current estimated camera pose, C_t, in the forward pass. By calculating the loss between the rendered image, I_t, and the reference image, R_t, the backward pass of the 3DGS pipeline then back-propagates the loss to the unfixed parameters, i.e., C_t, in a self-supervised manner. By S_t iterations, the estimated camera pose C_t progressively converges to a value that is close to the true pose.

Mapping. The purpose of mapping, on the other hand, is to reconstruct previously unreconstructed regions of the scene (e.g., the blue dashed blocks in Fig. 1) based on the current observations. At time t, we select w recent camera poses and fix their pose parameters. We then fine-tune the 3D Gaussian representations using these poses and their corresponding images. This fine-tuning follows the same training process as tracking, except that mapping updates only Gaussian parameters $\{G_i\}$ rather than $\{C_t\}$. Over S_m iterations, the Gaussian representation is progressively refined by inserting

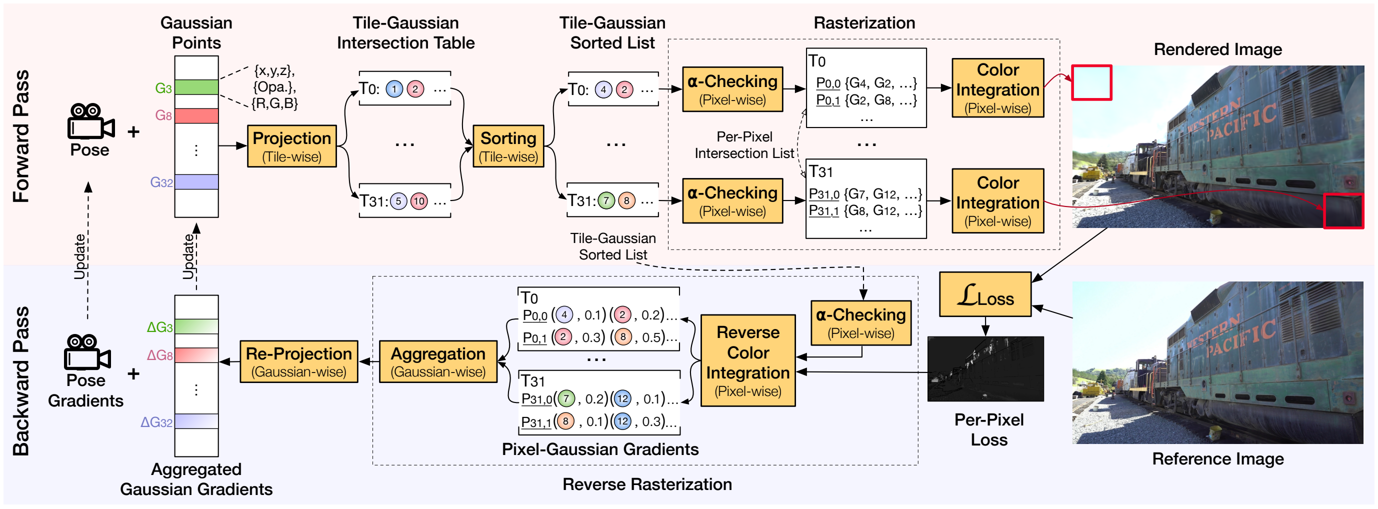


Fig. 3. The overview of 3DGS forward and backward passes. The forward pass consists of three stages: *projection*, *sorting*, and *rasterization*. Both projection and sorting are performed at tile granularity to amortize the computational cost across pixels, while rasterization must be performed at the pixel level to render individual pixels correctly. Because different pixels within a tile need to integrate different subsets of Gaussians. The backward pass mainly comprises two stages: *reverse rasterization* and *re-projection*. Reverse rasterization computes the partial gradients of all pixel-Gaussian pairs and aggregates them to the corresponding Gaussians. Re-projection then transforms the accumulated gradients from the camera coordinate system to the world coordinate system.

new Gaussian points until convergence. Note that, mapping is typically performed less frequently than tracking.

Order. Fig. 2 shows the timing diagram of a 3DGS-SLAM process. As shown, tracking and mapping are executed concurrently but at different frequencies. Tracking (in red) runs continuously at each frame to estimate the camera pose. On the other hand, mapping (in blue) is invoked less frequently. However, mapping, M_t, at time t must be executed after T_t due to the dependency between tracking and mapping.

B. 3DGS Training Pipeline.

The core of the 3DGS-SLAM algorithm is the 3DGS training process, which maintains two sets of tunable parameters: the camera poses that capture the overall trajectory, $\{C_t\}$, and the 3D Gaussian points, $\{G_i\}$, that model the reconstructed scenes. Each Gaussian has a set of attributes that capture the geometric and textural properties of the scene. As shown in Fig. 3, the training process can be classified into two passes: the *forward pass* and the *backward pass*.

Forward Pass. The overall forward pass consists of three stages: *projection*, *sorting*, and *rasterization*.

Projection. The purpose of projection is to filter out Gaussians that lie outside the current view frustum. To amortize computational overhead, current pipelines perform this filtering at the tile level, rather than pixel-by-pixel, to identify the intersection between Gaussians and rendering tiles. The results are then written into a tile-Gaussian intersection table.

Sorting. Once each tile obtains its intersected Gaussians, sorting determines the rendering order of those Gaussians within individual tiles. This stage ensures that all Gaussians are rendered in a correct order, from the closest to the farthest.

Rasterization. The sorted Gaussians in the tile-Gaussian sorted list are then rendered pixel-by-pixel. All pixels within a tile would first iterate over the Gaussians in the sorted list and perform α-checking. α-checking is used to filter out the Gaussians that do not intersect with the individual pixels. A Gaussian is considered to intersect with a pixel if its computed transparency, α_i, at that pixel exceeds a predefined threshold, α^*. Conceptually, each pixel forms its own list of intersected Gaussians after α-checking, as shown in Fig. 3.

Once this is complete, each pixel would integrate the color contributions from its own list and accumulate the final pixel value. This color integration process is governed by,

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

where $C(\mathbf{p})$ is the final color of pixel $\mathbf{p}$, α_i and $\mathbf{c}_i$ denote the transparency and color of the ith Gaussian, and Γ_i represents the accumulated transmittance along the ray from the first Gaussian to the $(i-1)$th Gaussian.

Backward Pass. Once the image is rendered, the backward pass first calculates the pixel-wise loss between the rendered image and the reference image. This loss is then backpropagated to the relevant Gaussians to update their parameters. Overall, the backward pass consists of two main stages: *reverse rasterization* and *re-projection*.

Reverse Rasterization. Overall, this stage computes the partial gradients of every pixel-Gaussian pair and then aggregates the relevant gradients to individual Gaussians.

This process begins by identifying the intersected Gaussian list for each pixel via α-checking. Similar to the forward pass, α-checking uses the previously cached tile-Gaussian sorted list from the forward pass to obtain the per-pixel intersection lists.

Given these per-pixel lists, the reverse color integration reverses the color integration process defined in Eqn. 1 and computes the partial gradients of individual Gaussians for each pixel. Unlike forward color integration, the reverse color

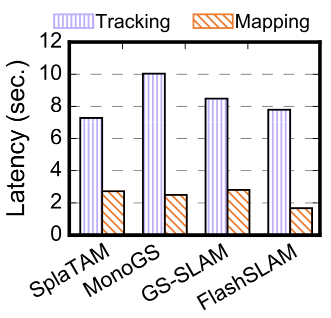


Fig. 4. The amortized latency of *tracking vs. mapping* across algorithms [36], [56], [61], [81]. Tracking dominates the execution.

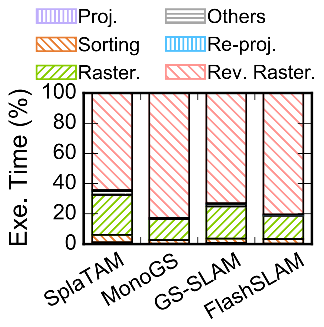


Fig. 5. Normalized execution breakdown across algorithms. Rasterization and reverse rasterization dominate the execution.

integration processes Gaussians from the Nth Gaussian to the 1st Gaussian. The resulting gradients for each pixel-Gaussian pair are then collected into the pixel-Gaussian gradient list.

Using the pixel-Gaussian gradients, the aggregation stage collects all gradients associated with each Gaussian and calculates the accumulated gradient for each Gaussian.

Re-Projection. This stage then transforms the accumulated gradients from the camera coordinate system to the world coordinate system. This stage's computation is often lightweight.

III. Motivation

We begin by showing the overall performance of 3DGS-SLAM algorithms. We then examine the key factors that contribute to the main computational overheads.

A. Performance Characteristics

Latency. Fig. 4 shows the amortized per-frame latency of tracking and mapping across different 3DGS-SLAM algorithms on a Nvidia Ampere mobile GPU [1]. The results are averaged over all scenes in the Replica dataset [70]. Tracking has much higher per-frame latency than mapping, because tracking is executed more frequently than mapping. Thus, the latency of mapping can often be hidden behind tracking.

Execution Breakdown. We further break down the execution time of tracking and mapping. Since both passes share the same pipeline in Fig. 3, Fig. 5 shows the execution time breakdown of key stages in the forward and backward passes. Across different algorithms, the primary execution bottleneck in the forward pass is rasterization, while reverse rasterization dominates the time of the backward pass. Together, these two stages account for 94.7% of the execution time.

B. Performance Bottlenecks

We further dissect the bottlenecks in forward and backward passes in the following characterizations, using SplaTAM [36].

Warp Divergence. As shown in Fig. 3, both projection and sorting in the forward pass are performed at the tile granularity to amortize computational cost across pixels within the same tile. However, rasterization must be executed at the pixel level to guarantee the rendering correctness: it first performs α-checking to identify the contributing Gaussians for each pixel and then integrates only those Gaussians.

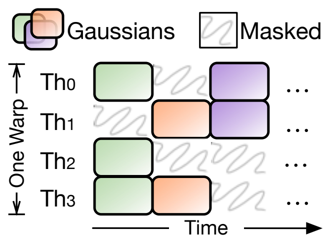


Fig. 6. An example of warp divergence when different threads integrate Gaussians in color integration.

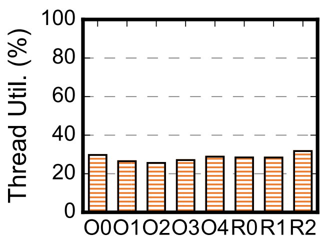


Fig. 7. The thread utilization in rasterization is low. The x-axis shows different scenes from Replica [70].

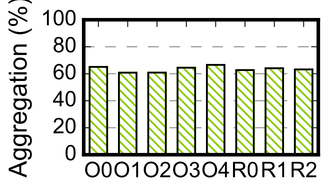


Fig. 8. The execution percentage of aggregation in the reverse rasterization stage using Replica [70] dataset.

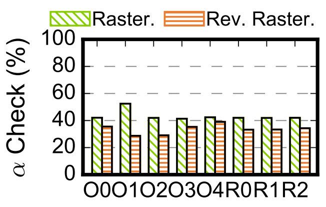


Fig. 9. The execution percentage of α-checking in both rasterization and reverse rasterization.

In the current rasterization pipeline, each thread is responsible for rendering one pixel. To leverage GPU parallelism, the color integration process broadcasts Gaussians within a GPU warp and masks those threads that do not need to integrate these Gaussians, as illustrated in Fig. 6. Such a process would inevitably cause warp divergence. We further measure GPU thread utilization during color integration, as shown in Fig. 7. On average, thread utilization is only 28.3%. This shows that the warp divergence during rasterization is severe.

Aggregation. As prior studies [11], [29], [77] show, one main reason that reverse rasterization is a key bottleneck is that Gaussians must gather partial gradients from different pixels. To avoid race conditions, current GPU pipelines rely on `atomicAdd` operations during the aggregation stage of reverse rasterization. However, these atomic operations would lead to frequent pipeline stalls. Fig. 8 shows the aggregation overhead in reverse rasterization. Over 63.5% of its execution time is spent on aggregation. Thus, reducing data contention in aggregation is critical for improving its performance.

α-Checking. Lastly, another reason that both rasterization and reverse rasterization are time-consuming is that α-checking must be performed for every pixel-Gaussian pair. Each α-checking requires evaluating an expensive operation, the exponential function, which must be executed by special functional units (SFUs), rather than massive compute cores [41]. As shown in Fig. 9, α-checking accounts for roughly 43.4% and 33.6% of the total time in rasterization and reverse rasterization, respectively. Thus, reducing the number of α-checks is critical to speed up these two stages.

IV. Algorithm

This section introduces our sparse processing framework, which reduces the overall computation by two orders of magnitude while maintaining task accuracy. Sec. IV-A presents our adaptive pixel sampling tailored for tracking and mapping.

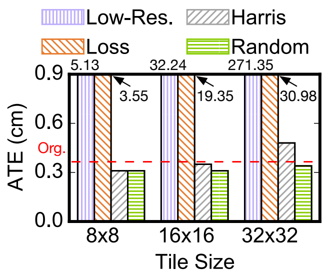


Fig. 10. The SLAM tracking error with different sampling strategies and tile sizes during tracking, using SplaTAM [36]. Here, lower is better. The red line shows the baseline accuracy. Random sampling achieves the best performance with robustness.

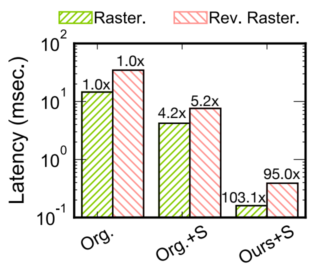


Fig. 11. The execution latency of rasterization and reverse rasterization on a mobile Ampere GPU [1] during tracking, using SplaTAM [36]. "S": applying sparse pixel sampling. Numbers represent speedups compared to the original pipeline.

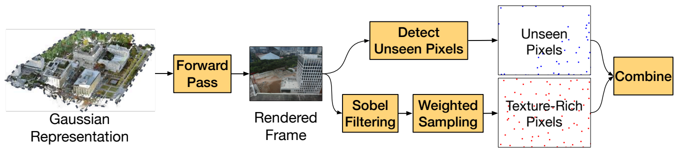


Fig. 12. The sampling algorithm for mapping. Two types of pixels are sampled in mapping. The first type is the pixels that were unseen in previous reconstructions, allowing the algorithm to focus on refining newly observed regions. The second type includes texture-rich pixels sampled across the entire image to capture global structural information.

Sec. IV-B then introduces our *pixel-based rendering* pipeline that fully unleashes the performance potential of sparse pixel processing. Lastly, Sec. IV-C revisits the remaining bottlenecks, which motivate our hardware support in Sec. V.

A. Adaptive Pixel Sampling

In Fig. 3, the overall computation of both the forward and backward passes in 3DGS-SLAM is roughly proportional to the number of rendered pixels. To reduce the computation in 3DGS-SLAM, our sparse sampling algorithm aims to select a subset of pixels for processing. Similar to prior studies [12], [16], [39], [42], [73], our sampling algorithm also exploits task-specific characteristics of tracking and mapping to reduce pixel sampling rates without impacting accuracy. However, prior work targets different domains, e.g., object detection [39] and eye tracking [16], [21], [73]. Their sampling techniques cannot be directly applied to 3DGS-SLAM. Thus, we design a new sampling algorithm for SLAM in the following paragraphs.

Tracking. The optimization target of tracking is to estimate the pose of the current frame via an iterative training process. Inspired by traditional SLAM algorithms [31], [59], [83], which detect and match key feature points for localization, we observe that, for each frame, tracking in 3DGS-SLAM only requires optimizing a single camera pose, i.e., a 4×4 transformation matrix. Such a process should not necessitate using all pixels from a dense frame. Therefore, our algorithm selectively processes only the necessary pixels in tracking.

Unlike prior study [77], which performs sampling at the granularity of image tiles, our sampling algorithm operates at the *pixel* level. Specifically, our algorithm sparsely selects one pixel per $w_t \times w_t$ image tile. Such a design has two main purposes: 1) adjacent pixels often carry similar information, tile-based selection would introduce redundant computation; 2) sampling one pixel per tile could capture the global features, making tracking more robust.

Fig. 10 shows the tracking accuracies of different sampling strategies under the same sampling rate. Here, "Low-Res." stands for downsampling to low-resolution images, while "Loss" stands for the method from GauSPU [77]. Both "Random" and "Harris" apply our sampling strategy with different selection metrics. "Random" select one pixel from $w_t \times w_t$ tile randomly, while "Harris" selects based on Harris descriptor [28]. Overall, methods lacking global coverage, e.g., GauSPU [77] or simply reducing the resolution, lead to low accuracy. In comparison, random sampling achieves equivalent or better accuracy compared to feature-based methods. For simplicity, we choose random sampling for tracking.

Mapping. Unlike tracking, which is used to estimate poses, the purpose of mapping is to reconstruct the previously unseen regions of the scene. In Fig. 12, we design our sampling algorithm to prioritize the unseen pixels. These pixels are typically concentrated along previously occluded object boundaries, where depth variations are significant, or in regions that were previously unexplored. At a given timestamp t, we define whether a pixel, $\mathbf{p}$, in the current image frame is unseen based on its accumulated transmittance, $\Gamma_{\text{final}}(\mathbf{p})$,

$$\mathcal{F}(\mathbf{p}) = \begin{cases} \text{unseen, if } \Gamma_{\text{final}}(\mathbf{p}) \; > \; 0.5, \\ \text{seen, otherwise.} \end{cases} \tag{2}$$

Here, $\Gamma_{\text{final}}(\mathbf{p})$ is computed during the first forward pass (we perform only once per mapping). A high transmittance means that very few Gaussians have contributed to this pixel. Therefore, function $\mathcal{F}(\mathbf{p})$ selects those pixels to better improve regions that still require reconstruction.

However, selecting only unseen pixels often leads to poor tracking accuracy, as such pixels tend to be too sparse (see results in Sec. VII-D). Thus, in addition to the unseen pixels, we also sample additional pixels across the image using a weighted sampling strategy, in Fig. 12. Specifically, we assign texture-rich pixels with higher probability and select one pixel per $w_m \times w_m$ tile. The probability, $\mathcal{P}(\mathbf{p})$, is defined as follows,

$$\mathcal{P}(\mathbf{p}) = w_R(\mathbf{p}) \times r, \text{ where } w_R(\mathbf{p}) = \sqrt{G_\text{x}^2 + G_\text{y}^2}, \tag{3}$$

where G_x and G_y are the horizontal and vertical gradients at pixel $\mathbf{p}$ using Sobel filters [34]. w_R stands for the overall gradient, which approximates the local texture richness. r is a random floating number between 0 and 1. Sec. VII-D further compares our strategy against other methods.

B. Pixel-Based Rendering

Motivation. While our sparse sampling algorithm largely reduces the number of processed pixels during both the forward and backward passes, we find that its actual speedup on existing mobile GPUs is limited. In Fig. 11, we test a case where we process one pixel per 16×16 tile on a mobile Ampere GPU [1], we expect a $256\times$ speedup on the two

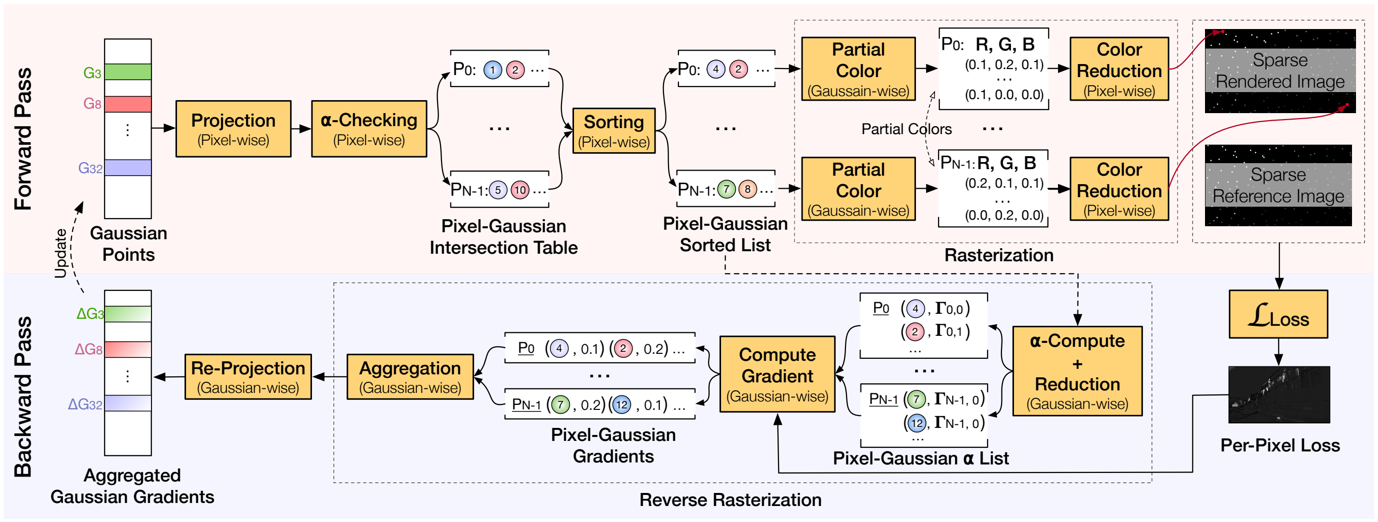


Fig. 13. Overview of our *pixel-based rendering* pipeline for sparse 3DGS-SLAM. To improve GPU thread utilization, we shift from pixel-wise parallelism to Gaussian-wise parallelism in both the rasterization and reverse rasterization stages. Instead of assigning one thread per pixel, our pipeline enables threads within a GPU warp to co-render a single pixel. Additionally, we introduce an optimization, preemptive α-checking, that moves α-checking from rasterization to projection in the pipeline. This not only reduces the workload of subsequent stages but also eliminates warp divergence in rasterization.

dominated stages: rasterization and reverse rasterization, since their computations are proportional to the number of pixels.

However, Fig. 11 shows that applying our sampling algorithm to the original SplaTAM (denoted as "Org.+S") yields only 4.2$\times$ and 5.2$\times$ speedup in rasterization and reverse rasterization, respectively, far below the theoretical gain. The root cause of this gap is the thread-to-pixel mapping in the original 3DGS pipeline, where each GPU thread is responsible for rendering a single pixel. Under sparse sampling, only one thread in a GPU warp does meaningful work, while other threads are idle, resulting in severe PE under-utilization.

Pipeline. To address this issue, we propose a new rendering paradigm, *pixel-based rendering* in Fig. 13, that parallelizes rasterization and reverse rasterization at the *Gaussian* level. By doing so, our pipeline significantly mitigates the PE under-utilization introduced by the original pipeline.

In the forward pass, we make the two key changes.

❶ We process projection and sorting at the pixel level, rather than at the tile level as in the original pipeline. Since our algorithm requires only a sparse subset of pixels, tile-level projection, i.e., identifying Gaussians that intersect with an entire tile, introduces redundant computations by including Gaussians that do not intersect with the sampled pixels. In contrast, performing the projection stage at the pixel level ensures that only the intersected Gaussians would be included in the subsequent stages, eliminating unnecessary work.

❷ We redesign the rasterization stage to operate at the granularity of Gaussians, such that each pixel is co-rendered by a warp of threads. Specifically, all threads within a warp share the same Gaussian list associated with a target pixel. The Gaussian list is then evenly distributed across the threads in the warp to ensure an even workload across threads. During rasterization, each thread computes the partial colors from a subset of Gaussians, as illustrated in Fig. 13. Once partial colors are computed, a reduction operation is performed to accumulate the final color of that pixel.

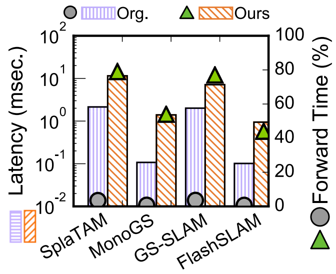
In the backward pass, the primary change is on the reverse rasterization stage. Similar to the forward rasterization, we shift from pixel-level parallelism to Gaussian-level parallelism. Unlike forward rasterization, our reverse rasterization inverts the color integration process, with two rounds of reductions. The first round is introduced by our pixel-based rendering.

❶ In the first reduction, we need first to compute the individual Gaussian transparency, α_i, in parallel across threads, and then perform a cross-thread reduction to obtain Γ_i for each Gaussian by accumulating the transparencies $\prod_{j=1}^{i-1}(1-\alpha_j)$ of all preceding Gaussians as defined in Eqn. 1. The original pipeline does not require this reduction because Γ_i values are computed and accumulated sequentially by a single thread.

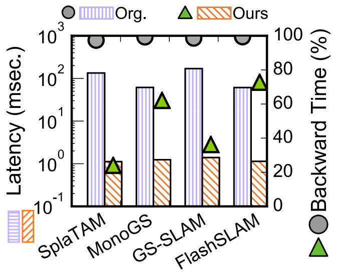
❷ Once Γ_i is computed, each thread can then independently calculate the partial gradients from the pixel to its relevant Gaussians. This step is also fully parallelizable in Fig. 13. Lastly, a second reduction is required to aggregate all partial gradients associated with each Gaussian, similar to the original pipeline. Although the second reduction involves atomic operations that cause data contentions, luckily, our sparse processing naturally alleviates these data contentions.

Preemptive α-Checking. In addition to pixel-based rendering, we also introduce *preemptive α-checking*, an optimization that eliminates the warp divergence in rasterization. In the original pipeline, α-checking is needed during rasterization to filter out Gaussians that do not intersect with the target pixel or whose transparency is below a threshold, α^*. Since different pixels often integrate different subsets of Gaussians, α-checking would introduce warp divergence (Fig. 6).

In our pipeline, the Gaussian list in rasterization is no longer shared across pixels, which enables us to move α-checking earlier in the pipeline. This way, we can effectively pre-filter



(a) *Projection* becomes the new bottleneck in the forward pass.



(b) *Reverse rasterization* remains the key bottleneck in backward pass.

Fig. 14. The execution latency (left y-axis) and the relative time (right y-axis) of *projection* and *reverse rasterization* on mobile GPU during tracking.

Gaussians that will have no contribution to the target pixel. As a result, we reduce the workloads of subsequent sorting and rasterization. More importantly, the pixel-Gaussian sorted list in Fig. 13 contains only relevant Gaussians that need to be integrated into the final pixel value, ensuring no warp divergence in rasterization and reverse rasterization.

Walk-Through. Fig. 13 illustrates the execution flow of our algorithm. In the forward pass, we first perform projection and α-checking for individual pixels to construct an intersection table that records the set of intersected Gaussians for each pixel. Next, the intersected Gaussians are sorted by depth for each pixel. Once the pixel-Gaussian sorted list is obtained, threads within a warp collaboratively compute the partial colors of all Gaussians in parallel. Finally, we perform a color reduction to integrate partial colors into the final pixel.

In the backward pass, we first compute the per-pixel loss. We then reuse the pixel-Gaussian sorted list from the forward pass to compute the transparency α_i of each intersected Gaussian in parallel. Next, each intersected pixel-Gaussian pair computes the accumulated transmittance Γ_i by accumulating the α_i values of all preceding Gaussians via across-thread reductions (Eqn. 1). Using the computed Γ_i values, each thread then calculates the partial gradient of each intersected pixel-Gaussian pair. A second reduction is then used to aggregate the partial gradients for each Gaussian. Finally, the aggregated gradients are backpropagated to Gaussians via re-projection.

C. *Bottleneck Analysis*

With all the optimizations together in Sec. IV-B, Fig. 11 shows that our pixel-based rendering achieves 103.1$\times$ and 95.0$\times$ speedup on rasterization and reverse rasterization on SplaTAM [36]. However, by further analyzing the execution time breakdown of our pixel-based rendering pipeline, our experiment shows that the performance bottleneck shifts from rasterization to projection in the forward pass, as shown in Fig. 14a. The proportion of forward time spent in projection increases from 2.1% to 63.8%. The main reason is that we move α-checking to the projection stage, which substantially increases the workload of the projection stage.

Meanwhile, in the backward pass, although the overall execution time is reduced significantly, Fig. 14b shows that the relative backward time spent in reverse rasterization decreases from 98.7% to 48.76%. However, reverse rasterization still accounts for the majority of the backward time (up to 72.7%). There are primarily two reasons: 1) our reverse rasterization introduces an additional round of across-thread reductions, which incurs synchronization overhead; 2) aggregation remains a key bottleneck due to frequent pipeline stalls caused by atomic operations, despite that sparse pixel processing helps to alleviate this data contention across threads.

V. Architectural Support

To address the bottleneck shifts brought by our pixel-based rendering, we propose our accelerator, Splatonic, to further boost the performance. We first give an overview (Sec. V-A) and discuss how we design a lightweight pipelined architecture (Sec. V-B) and address the key bottlenecks (Sec. V-C).

A. *Overview*

Following the philosophy of our pixel-based rendering, we design a dedicated architecture, as shown in Fig. 15. Our design builds upon MetaSapiens [51], a pipelined 3DGS accelerator for the forward pass only. We extend MetaSapiens to support both forward and backward passes, with the colored components highlighting our augmentations.

Specifically, we make the following contributions. First, we design a streaming architecture for sorting, rasterization, and reverse rasterization stages, and enable the pipelining across those stages. Second, to support pipelining, we propose a lightweight rasterization engine in Sec. V-B that removes the redundant α-checking components that are targeted for tile-based rendering and addresses the synchronization overhead in reverse rasterization, i.e., the first round of across-thread reduction due to pixel-based rendering. Lastly, to address the bottleneck shifts described in Sec. IV-C, we further augment our architecture with: 1) a projection unit to address the increased workload in projection and accelerate the preemptive α-checking, and 2) an aggregation unit to alleviate the pipeline stalls due to data contention and frequent off-chip traffic.

B. *Rasterization Engine*

In this section, we present our rasterization engine, which simplifies the core computation in prior designs [29], [77] and enables high parallelism across Gaussians. Overall, the rasterization engine contains two sets of processing units: render units for rasterization and reverse render units for reverse rasterization. We next describe them individually.

Render Unit. One key inefficiency in prior render unit designs [46], [51] is that each Gaussian undergoes α-checking to determine if it contributes to a pixel. This often leads to PE under-utilization on accelerators [17]. To address this, our architectural design adopts the preemptive α-checking from Sec. IV-B, which moves the α-checking logic from rasterization to projection. By doing so, we guarantee that only Gaussians that contribute to the target pixel proceed to rasterization. This optimization allows us to skip the α-checking logic from render units and compute the partial color of each Gaussian directly, as shown in Fig. 15. During rasterization, multiple render units read different Gaussian data

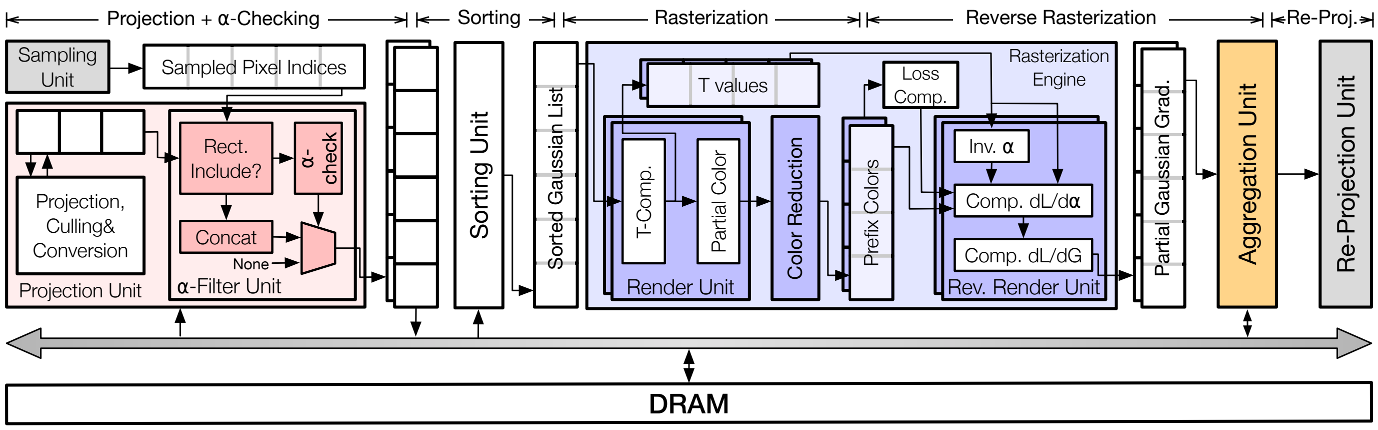


Fig. 15. Overview of our pipelined architecture. Our architecture is built upon METASAPIENS [51], a 3DGS accelerator for the forward pass only. We augment the baseline architecture to support the backward pass. Specifically, we co-design a simplified rasterization engine (purple-colored) that mitigates the PE under-utilization in rasterization and reverse rasterization. We also propose a caching technique between these two stages to avoid the across-thread reduction in reverse rasterization (Sec. V-B). Meanwhile, we augment the projection unit (pink-colored) to support preemptive α-checking and design a dedicated aggregation unit (yellow-colored) to accelerate reverse rasterization (Sec. V-C).

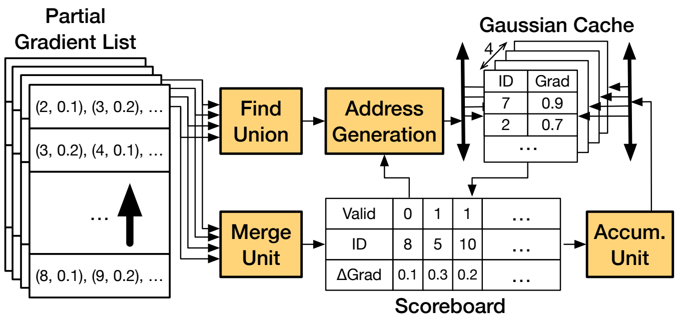


Fig. 16. The design of the aggregation unit. We batch process the partial gradients of multiple pixels. A merge unit is to perform on-chip gradient reduction before accumulating with partial accumulated gradients from Gaussian cache. A scoreboard records which Gaussian is ready for accumulation.

and are executed in parallel. A dedicated reduction unit then gathers these partial colors to accumulate the final pixel value.

Reverse Render Unit. Sec. IV-C shows that, after shifting to pixel-based rendering, one overhead in reverse rasterization is the need to perform one additional round of across-thread reductions. The additional round of reduction is to compute the accumulated transmittance Γ_i for each Gaussian i and the prefix color C_i, which integrates the partial colors from the first to the ith Gaussian. Note that, both Γ_i and C_i are intermediate results during forward rasterization. However, caching these intermediate values in the original tile-based rendering is prohibitively expensive. Because every pixel in a dense frame requires storing these two values for all contributed Gaussians. However, in our pixel-based rendering pipeline, we only need to store the data for one single pixel at a time, which is largely manageable and can be held entirely on-chip.

By exploiting this feature, we co-design the reverse render unit with the render unit and color reduction unit in the forward pass to output and store the intermediate values, i.e., $\{\Gamma_i\}$ and $\{C_i\}$, for one pixel in the on-chip buffer, as shown in Fig. 15. These cached data are then forwarded directly to the reverse render units. Using these data, the reverse render units avoid implementing additional logic for reduction and inter-PE communication. More importantly, eliminating these reduction operations removes inter-PE dependencies, so that the partial gradients for each Gaussian can be computed in parallel. Once computed, the partial gradients are buffered on-chip for the aggregation stage, which is introduced in Sec. V-C.

C. Addressing Key Bottlenecks

Aggregation Unit. To support accumulating partial gradients during reverse rasterization, we introduce a dedicated aggregation unit following the rasterization engine. In the backward pass, all partial gradients in the pixel-Gaussian gradient list need to be accumulated into their corresponding Gaussians. However, the total number of partial gradients (roughly the product of the number of Gaussians and the number of pixels) is too large to perform on-chip reduction. Moreover, the highly irregular accumulation patterns require frequently reloading unfinished accumulated gradients from off-chip memory. Thus, we design an aggregation unit to hide frequent pipeline stalls introduced by off-chip memory traffic.

Fig. 16 shows the design of our aggregation unit. Here, each entry in the partial gradient list contains the partial gradients from a single pixel. During aggregation, the aggregation unit first reads the Gaussian IDs from n entries, i.e., $n = 4$ in Fig. 16. We compute the union of Gaussian IDs across these entries and then load the corresponding accumulated gradients from off-chip memory into the Gaussian cache.

In parallel, the aggregation unit reads n Gaussian tuples, i.e., the Gaussian ID and its partial gradient, and forwards them to the merge unit. The merge unit performs intra-batch reductions by combining gradients with identical Gaussian IDs and stores the results in the scoreboard. The role of the scoreboard is to maintain the current status of stored Gaussians: their IDs and whether their partial accumulated gradients have been loaded into the cache. The partial accumulated gradients mean those accumulated Gaussian gradients that are done partially and are unfinished. Meanwhile, each cycle, the accumulation unit checks the scoreboard for ready Gaussians, i.e., those whose partial accumulated gradients are available in the cache. Then, the accumulation unit reads the partial accumulated gradients from the Gaussian cache, updates them with the partial gradients stored in the scoreboard, and writes back to the cache. This way, we can effectively hide the off-chip traffic latency by simultaneously updating other Gaussian gradients.

Projection Unit. In Sec. IV-C, we show that once applying our pixel-based rendering, the bottleneck in the forward pass shifts from rasterization to projection due to preemptive α-checking (Fig. 14a). Such a shift comes from two factors: 1) each Gaussian's bounding box (BBox) must be checked against all sampled pixels; and 2) α-checking involves expensive exponential computations, as mentioned in Sec. III-B.

To address the first issue, we propose a direct indexing method using the four corners of each Gaussian's BBox to limit the number of pixels we iterate. Since our sampling algorithm selects one pixel per tile, we can directly compute the index range in the sampled pixel list using the minimal and maximal coordinates. This way, we avoid exhausting the entire pixel list and avoid unnecessary α-checking. Note that, the unseen pixel indices in mapping are stored separately, so that the unseen pixel indices do not interrupt our indexing strategy. Second, to mitigate the computational cost of exponentiation, we approximate the exponential function with a lookup table (LUT) [53]. Our empirical evaluation shows that a LUT with a size of 64 entries is sufficient to maintain the same accuracy.

VI. Experimental Setup

Hardware Configuration. Overall, SPLATONIC has a basic configuration similar to METASAPIENS [51]. SPLATONIC consists of eight projection units, four hierarchical sorting units, four rasterization engines, and one aggregation unit. We augment each projection unit with four α-filter units. Each rasterization engine has 2×2 render units and 2×2 reverse render units with one color reduction unit in between. To store the intermediate Γ_i and C_i values, each rasterization engine is designed with an 8 KB double buffer. In addition, a 64 KB global double buffer is used to hold the intermediate data of the entire pipeline. Lastly, the aggregation unit is designed with four channels to process the partial gradients of four pixels in parallel with a 32 KB Gaussian cache and a 8 KB scoreboard.

Experimental Methodology. For GPU performance, we measure latency, including the execution time as well as the kernel launch on the mobile Ampere GPU. The GPU power is directly obtained using the built-in power sensing circuitry on Orin. For accelerator performance, we develop a RTL implementation of our pipelined architecture. The design is clocked at 500 MHz. The RTL design is implemented via Synposys synthesis and Cadence layout tools in TSMC 16nm FinFET technology. The numbers of our RTL design are then scaled down to 8 nm node using DeepScaleTool [66], [69] to match the mobile Ampere GPU on Nvidia Orin SoC in 8 nm node [1]. The SRAMs are generated using the Arm Artisan memory compiler. The DRAM is modeled after 4 channels of Micron 16 Gb LPDDR3-1600 memory [2]. The DRAM energy is obtained using Micron System Power Calculators [3].

Area. Overall, SPLATONIC has a smaller area (1.07 mm^2) compared to other 3DGS accelerators, such as GSCore (1.77 mm^2) [46] and GSArch (3.42 mm^2) [29], with all areas scaled down to 16nm node using DeepScaleTool [66], [69]. The primary contributor to SPLATONIC compact design is its efficient rasterization engine, which accounts for only 28% of the total area. The remaining stages occupy 57% of the area. Some are due to the larger projection units. The rest are SRAMs, which comprise 15% of the area.

Software Setup. We evaluate SPLATONIC on the two widely used indoor SLAM datasets, Replica [70] and TUM RGB-D [71]. Replica comprises eight sequences. Each sequence consists of 2000 RGB-D images. TUM RGB-D is a more complex real-world dataset with fast camera motion. To evaluate the effectiveness of our sampling algorithms, we use four different 3DGS-SLAM algorithms: SplaTAM [36], MonoGS [56], GS-SLAM [81], and FlashSLAM [61].

Metrics. We report two standard accuracy metrics: absolute trajectory error (ATE), which is used to measure the accuracy of pose estimation, and peak signal-to-noise ratio (PSNR) which is to measure the reconstruction quality.

Baselines. We compare three hardware baselines:

- GPU: a mobile Ampere GPU on Nvidia Orin SoC [1].
- GSARCH [29]: a dedicated 3DGS training architecture for the tile-based rendering pipeline. Here, we compare the edge configuration reported in the GSArch paper.
- GAUSPU [77]: a dedicated accelerator for 3DGS-SLAM. It executes projection and sorting on GPU, and the remaining stages are executed on the dedicated accelerator. Here, we model the GPU performance using the parameters obtained from the mobile GPU on Orin.

To ensure a fair comparison with the mobile GPU performance on Orin, we scale both designs down to 8 nm node using DeepScaleTool [66], [69] and clock them at 500 MHz. We do not include inference-only 3DGS accelerators [17], [45]–[47], [51], [79] in our evaluation, because their delay will be dominated by the backward pass latency (see Fig. 5).

Variants. We evaluate two variants of SPLATONIC to separate the contributions in our paper: SPLATONIC-SW, which executes our pixel-based rendering on mobile GPUs; and SPLATONIC-HW, which executes our pixel-based rendering on our proposed pipelined architecture.

VII. Evaluation

A. Accuracy

In our sampling algorithm, we set the tile size to $w_t = 16 \times 16$ for tracking and $w_m = 4 \times 4$ for mapping. During mapping, we perform one full-frame mapping for every four frames. Unless otherwise specified, this configuration is used as the default setting for the remaining evaluations.

Tracking Accuracy. Fig. 17a and Fig. 18a show the tracking accuracy comparison between the original 3DGS-SLAM algorithms and the ones with our sparse sampling

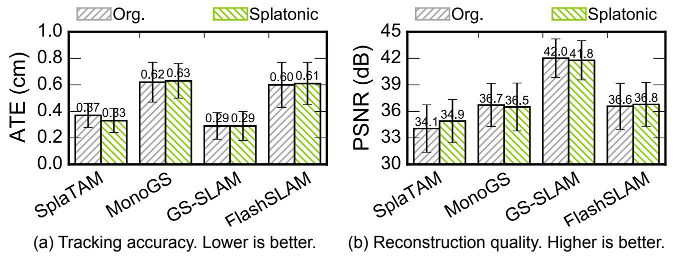


Fig. 17. The tracking accuracy and reconstruction quality comparison between the baselines and our sampling algorithm across 8 sequences on Replica.

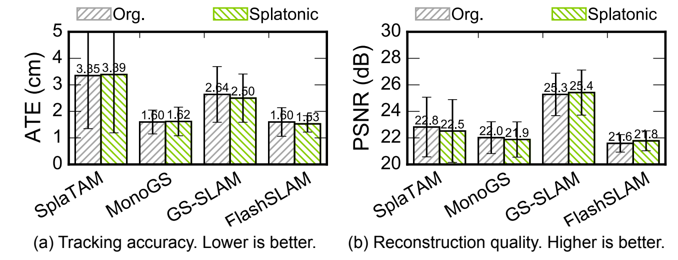


Fig. 18. The tracking accuracy and reconstruction quality comparison between the baselines and our sampling algorithm over 3 sequences on TUM RGB-D.

algorithm the tracking accuracy. Overall, SPLATONIC matches or outperforms the performance of original algorithms. Across four different algorithms, the average ATEs of SPLATONIC are 0.46 cm and 2.26 cm on Replica and TUM RGB-D, respectively. SPLATONIC are 0.01 and 0.03 lower than the baselines. Our better accuracy stems from the fact that sparsely sampled pixels can help eliminate the false matching in regions with repetitive patterns, similar to extracting keypoint descriptors in conventional SLAM. The same principle in conventional SLAM is also applied to 3DGS-SLAMs.

Reconstruction Quality. Fig. 17b and Fig. 18b compare the reconstruction quality between the original algorithms and the ones with our proposed sparse sampling approach. Across all four 3DGS-SLAM algorithms, SPLATONIC achieves higher PSNR values on average. Notably, SPLATONIC outperforms the baseline by 0.8 dB on SplaTAM [36]. This demonstrates that our sampling strategy in the mapping process effectively directs the training focus toward unseen regions and texture-rich areas, thus enhancing the overall reconstruction quality.

B. GPU Performance.

We first show that our sparse pixel sampling with proposed pixel-based rendering can already achieve significant speedups and energy savings on off-the-shelf GPUs *without* hardware support. Fig. 19 shows the end-to-end speedup and energy savings of SPLATONIC on the Nvidia Orin SoC.

In our evaluation, we assume that tracking and mapping are executed separately on two identical mobile GPUs with equal compute power, allowing these two stages to run in parallel. For comparison, we also include a variant, ORG.+S, which applies only our sparse pixel sampling algorithm without integrating our pixel-based rendering pipeline.

End-to-End Performance. Fig. 19 shows the end-to-end speedup and normalized energy of SPLATONIC compared to baseline algorithms on a mobile Ampere GPU on Nvidia Orin.

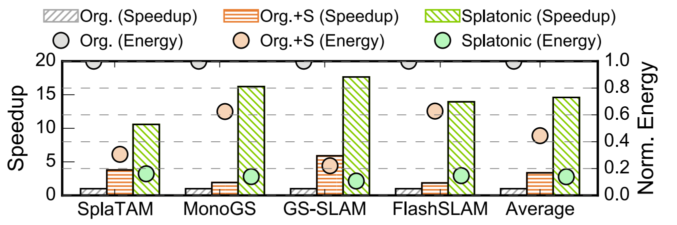


Fig. 19. The end-to-end speedup of SPLATONIC over the baseline algorithms on mobile Ampere GPU Orin. ORG.+S is the baseline that applies sparse pixel sampling without our pixel-based rendering pipeline. Note that, the tracking speedup is aligned with the end-to-end speedup.

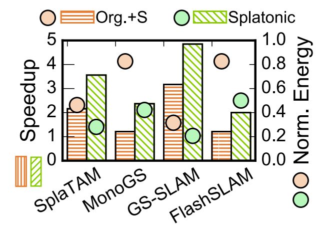


Fig. 20. The speedup and energy savings comparison on mapping. Despite limited speedup, the latency of mapping still can be hidden by tracking.

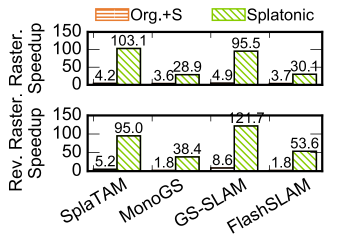


Fig. 21. The speedup on two key bottleneck stages, rasterization in forward pass and reverse rasterization in the backward pass, during tracking.

The left y-axis shows the speedup against the GPU baselines, while the right y-axis shows the normalized energy. Overall, SPLATONIC achieves 14.6× speedup and saves 86.1% energy compared to the original algorithms. The end-to-end speedup is aligned with the tracking speedup, as the latency of mapping can be hidden by tracking. In comparison, ORG.+S achieves only 3.4× speedup and 55.5% energy savings. This is because the original tile-based rendering pipeline would result in low GPU utilization and severe warp divergence.

Meanwhile, we also report the standalone speedup and energy savings for mapping in Fig. 20. On average, SPLATONIC can achieve only 3.2× speedup and 60.0% energy savings on mapping. This is because mapping needs to render more pixels (roughly one pixel per 4×4 tile) to reconstruct unseen regions. As the number of rendered pixels increases, the advantages of our pixel-based pipeline might be offset by the additional overhead introduced in projection and sorting stages due to no sharing of computation between pixels. Sec. VII-E further shows the sensitivity of speedup to the pixel sampling rate.

Bottleneck stages. The main performance gain of SPLATONIC on GPU is from addressing two key bottlenecks, rasterization in forward pass and reverse rasterization in the backward pass. Fig. 21 further analyzes the speedup of these two bottleneck stages during the tracking process. Without modifying the pipeline, applying sparse sampling alone yields only 4.1× and 4.3× speedup on rasterization and reverse rasterization, respectively. In contrast, our pipeline achieves 64.4× and 77.2× speedup on these two stages, respectively.

C. Hardware Performance

Since tracking dominates the overall execution, we primarily focus on the tracking performance in this section. Fig. 22 shows the comparison of the performance and energy savings

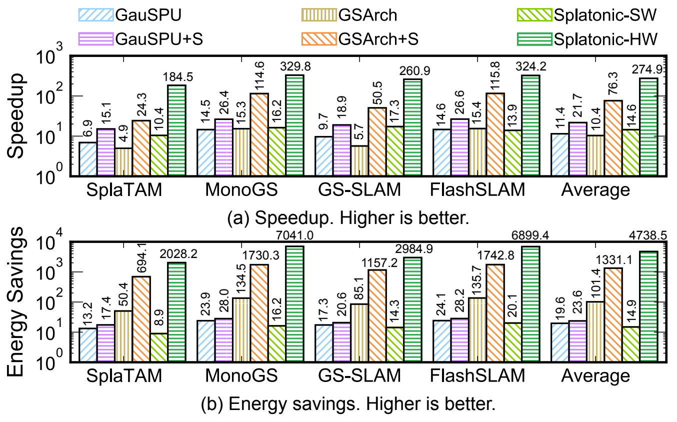


Fig. 22. The performance and energy consumption comparison across different dedicated architectures during tracking. "S": applying sparse pixel sampling. Numbers are normalized against GPU.

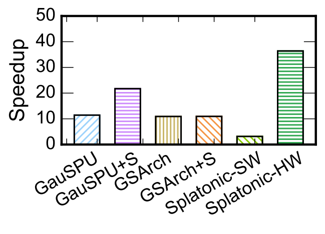


Fig. 23. The mapping speedup comparison across different dedicated architectures. SPLATONIC still outperforms the other two accelerators. The legend is shared with Fig. 22.

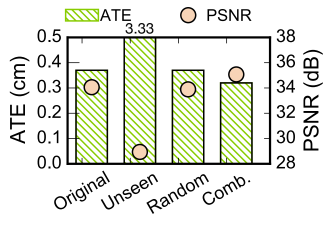


Fig. 24. Ablation study of different sampling strategies in mapping. We only show the results of SplaTAM. "Comb": uses both weighted random sampling and unseen pixels.

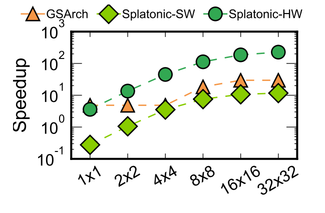


Fig. 25. The sensitivity of performance to the sampling rate. Numbers are normalized to GPU.

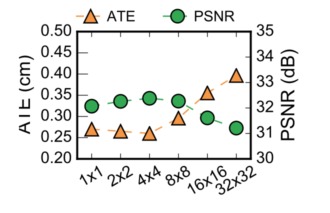


Fig. 26. The sensitivity of accuracy to the sampling rate.

across different architectures. For a fair comparison, we also include variants of GAUSPU and GSARCH that incorporate our sparse sampling algorithm, denoted with the "+S" suffix. All numbers are normalized to the GPU baseline on Orin. To better show the difference across hardware variants, we define the energy saving as the ratio of energy consumption between GPU and each corresponding variant.

In Fig. 22a, SPLATONIC-HW achieves 274.9× speedup, the highest performance compared to GAUSPU+S and GSARCH+S. Because both GAUSPU+S and GSARCH+S are accelerators designed for tile- or subtile-based rendering, where sparse pixel sampling leads to poor PE utilization in rasterization and reverse rasterization stages. Also, our simplified rasterization engine further reduces both the overall computation and off-chip data traffic in these two stages. Surprisingly, our software version, SPLATONIC-SW, outperforms GAUSPU and GSARCH, both of which are variants that execute the dense 3DGS-SLAM.

On energy savings, Fig. 22b shows a similar trend with the performance results. Overall, SPLATONIC-HW has the highest energy efficiency with 4738.5× of energy savings. In comparison, GAUSPU+S and GSARCH+S achieves 23.6× and 1331.1× of energy savings, respectively. The relatively low energy efficiency of GAUSPU+S is because it relies on GPUs to execute the projection and sorting stages. Meanwhile, GSARCH+S has relatively smaller energy savings compared to SPLATONIC-HW due to its sub-tile rendering, whereas SPLATONIC-HW benefits from a simplified rasterization.

In addition, we also show the mapping performance comparison in Fig. 23. The overall trends stay the same as tracking.

D. Ablation Study

Fig. 24 shows an ablation study that evaluates the contributions of different components in our mapping sampling algorithm. The results show that combining weighted random sampling (for texture-rich pixels) with unseen pixels yields the highest accuracy. Note that, our combined strategy outperforms the original algorithm in both pose tracking and reconstruction quality. Overall, our combined variant achieves a 0.05 cm reduction in pose error and a 1.0 dB quality improvement compared to the baseline algorithm.

E. Sensitivity Study

Fig. 25 shows the sensitivity of performance to the sampling rate. The x-axis represents different tile sizes, where each tile contains one sampled pixel. The results show that our pixel-based rendering is not always the optimal choice. As the tile size decreases, the data sharing among adjacent pixels increases. Tile-based rendering can amortize computation, thereby achieving higher speedup. For example, at a tile size of 1×1, SPLATONIC-HW yields lower speedup than GSARCH. However, when rendered pixels are sparse, SPLATONIC-HW significantly outperforms tile-based accelerators.

Meanwhile, Fig. 26 shows the sensitivity of mapping to the sampling rate. Here, we present the SplaTAM result of a single sequence, *Office 2*, in the Replica dataset [70]. We find that a tile size of 4×4 gives the best trade-off between performance and reconstruction accuracy. The sensitivity of tracking to the sampling rate has already been shown in Fig. 10.

Fig. 27 shows the sensitivity of performance to the number of projection units and render units. We do not conduct the sensitivity study on different buffer sizes, as each buffer size is tightly coupled with the corresponding PEs to support double buffering without wasting the on-chip buffer resources. In Fig. 27, the buffer size of each configuration is proportional to its corresponding PE counts. All performance numbers are normalized to the default configurations, 8 projection units and 4 render units. Overall, we find that the performance gain is mainly affected by the number of projection units, especially when there is a small number of projection units. As the number of projection units increases, projection is no longer a bottleneck. Instead, further increasing the number of render units improves the overall performance.

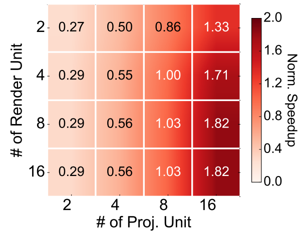


Fig. 27. The sensitivity of performance to the number of projection units and render units.

VIII. Related Work

3DGS Acceleration. There is a wide range of studies that have proposed different techniques to improve the efficiency of 3DGS on GPUs. Several studies have proposed various compression schemes or data structures to optimize storage and performance [23], [38], [43], [53], [60], [63]. For instance, CompactGS [43] proposes to use vector quantization to reduce the model size. While HierarchicalGS [38] and OctreeGS [63] leverage the tree-like data structure to avoid unnecessary computations. Other approaches propose various pruning techniques [13], [14], [51], [55] to eliminate insignificant Gaussians. Additional efforts [15], [27], [32], [75] focus on GPU-level optimizations to mitigate workload imbalance and warp divergence during rasterization. For instance, both AdR-Gaussian [75] and Seele [32] tame the warp divergence with co-training techniques.

In contrast to those application-agnostic methods, SPLATONIC exploits the unique algorithmic characteristics of SLAM by introducing a pixel sampling algorithm and a pixel-based rendering pipeline, significantly improving efficiency.

Architecture for Neural Rendering. A wide range of studies have been done on accelerating neural rendering. Earlier work primarily focused on Neural Radiance Fields (NeRF) [19], [22], [44], [48], [49], [52], [57], [62], [68], which is the predecessor of 3DGS. Recent years, due to the superior efficiency of 3DGS, studies have shifted their focus to 3DGS accelerations [11], [17], [29], [45]–[47], [50], [51], [79], [82].

Some studies propose dedicated hardware accelerators. For instance, both GSCore [46] and GBU [79] are designed for accelerating the forward pass in 3DGS. MetaSapiens [51] leverages the human visual perception and designs a dedicated rendering system for VR. Lumina [17] proposes a caching technique to amortize the per-pixel rendering cost. Meanwhile, GSArch [29] and GauSPU [77] focus on the training procedure and address the frequent pipeline stalls due to off-chip traffic. On the other side of the spectrum, some approaches augment general-purpose GPUs to improve 3DGS performance. For example, VR-Pipe [45] improves GPU-based inference, while ARC [11] proposes architectural optimizations for training.

Nevertheless, all these architectural designs are still built on top of the conventional tile-based rendering paradigm, which is inherently inefficient for sparse pixel processing. In contrast, our co-designed pipelined architecture eliminates redundant computation and significantly improves PE utilization. Meanwhile, SPLATONIC can extend beyond SLAM applications. Recent studies have explored sparse training techniques [8], [63] to reduce the computational overhead of 3DGS training. Other studies have proposed sparse rendering approaches for foveated rendering in VR [9], [18], [51]. By leveraging our pixel-based rendering pipeline, these methods can be further accelerated, demonstrating the broader applicability of our design across diverse 3DGS-related domains.

Sparse Processing. Quite a few studies have proposed sparse processing techniques in various vision tasks to reduce the overall computation and data transmission [16], [21], [24], [25], [30], [39], [40], [64]. For instance, fast R-CNN [24] and faster R-CNN [64] introduced region proposal networks for general object detection. Others [16], [39], [58] react based on the previous results or intermediate data to pre-filter related data before sending it through compute-heavy backbones. For instance, Rhythmic pixel regions [39] provides a flexible interface that allows applications to dynamically select region-of-interests. More recent efforts [20], [21], [54], [80] also co-design algorithms with the camera sensor to reduce the overall energy consumption across the entire sensor-compute system.

IX. Conclusion and Discussion

In this work, we present SPLATONIC, a hardware-software co-designed solution to accelerate 3DGS-based SLAM for real-time performance on mobile platforms. By leveraging classic algorithmic insights from traditional SLAM with our dedicated pixel-based rendering, we show that SPLATONIC achieves one order of magnitude higher performance on off-the-shelf GPUs, and we further boost performance and efficiency with our dedicated hardware support.

Acknowledgment

We are sincerely grateful to Weikai Lin from University of Rochester for providing valuable support throughout this research. This work was supported by the National Natural Science Foundation of China (NSFC) Grants (62532006 and 62402312) and Shanghai Qi Zhi Institute Innovation Program SQZ202316.

References

[1] "Jetson orin for next-gen robotics." [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[2] "Micron 178-Ball, Single-Channel Mobile LPDDR3 SDRAM Features." [Online]. Available: 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

[3] "Micron System Power Calculators." [Online]. Available: https://www.micron.com/support/tools-and-utilities/power-calc

[4] "Nvidia jetson orin nx 16 gb." [Online]. Available: https://www.techpowerup.com/gpu-specs/jetson-orin-nx-16-gb.c4086

[5] "Nvidia reveals xavier soc details." [Online]. Available: https://www.forbes.com/sites/moorinsights/2018/08/24/nvidia-reveals-xavier-soc-details/amp/

[6] L. Bai, C. Tian, J. Yang, S. Zhang, M. Suganuma, and T. Okatani, "Rp-slam: Real-time photorealistic slam with efficient 3d gaussian splatting," *arXiv preprint arXiv:2412.09868*, 2024.

[7] G. Bresson, Z. Alsayed, L. Yu, and S. Glaser, "Simultaneous localization and mapping: A survey of current trends in autonomous driving," *IEEE Transactions on Intelligent Vehicles*, vol. 2, no. 3, pp. 194–220, 2017.

[8] Y. Chen, J. Jiang, K. Jiang, X. Tang, Z. Li, X. Liu, and Y. Nie, "Dashgaussian: Optimizing 3d gaussian splatting in 200 seconds," in *Proceedings of the Computer Vision and Pattern Recognition Conference*, 2025, pp. 11 146–11 155.

[9] N. Deng, Z. He, J. Ye, B. Duinkharjav, P. Chakravarthula, X. Yang, and Q. Sun, "Fov-nerf: Foveated neural radiance fields for virtual reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 11, pp. 3854–3864, 2022.

[10] J. Duan, S. Yu, H. L. Tan, H. Zhu, and C. Tan, "A survey of embodied ai: From simulators to research tasks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 6, no. 2, pp. 230–244, 2022.

[11] S. Durvasula, A. Zhao, F. Chen, R. Liang, P. K. Sanjaya, Y. Guan, C. Giannoula, and N. Vijaykumar, "Arc: Warp-level adaptive atomic reduction in gpus to accelerate differentiable rendering," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 64–83.

[12] K. Egiazarian, A. Foi, and V. Katkovnik, "Compressed sensing image reconstruction via recursive spatially adaptive filtering," in *2007 IEEE International Conference on Image Processing*, vol. 1. IEEE, 2007, pp. I–549.

[13] Z. Fan, K. Wang, K. Wen, Z. Zhu, D. Xu, and Z. Wang, "Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps," *arXiv preprint arXiv:2311.17245*, 2023.

[14] G. Fang and B. Wang, "Mini-splatting: Representing scenes with a constrained number of gaussians," *arXiv preprint arXiv:2403.14166*, 2024.

[15] G. Feng, S. Chen, R. Fu, Z. Liao, Y. Wang, T. Liu, Z. Pei, H. Li, X. Zhang, and B. Dai, "Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering," *arXiv preprint arXiv:2408.07967*, 2024.

[16] Y. Feng, N. Goulding-Hotta, A. Khan, H. Reyserhove, and Y. Zhu, "Real-time gaze tracking with event-driven eye segmentation," in *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, 2022, pp. 399–408.

[17] Y. Feng, W. Lin, Y. Cheng, Z. Liu, J. Leng, M. Guo, C. Chen, S. Sun, and Y. Zhu, "Lumina: Real-time mobile neural rendering by exploiting computational redundancy," *arXiv preprint arXiv:2506.05682*, 2025.

[18] Y. Feng, W. Lin, Z. Liu, J. Leng, M. Guo, H. Zhao, X. Hou, J. Zhao, and Y. Zhu, "Potamoi: Accelerating neural rendering via a unified streaming architecture," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 4, pp. 1–25, 2024.

[19] Y. Feng, Z. Liu, J. Leng, M. Guo, and Y. Zhu, "Cicero: Addressing algorithmic and architectural bottlenecks in neural rendering by radiance warping and memory optimizations," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2024.

[20] Y. Feng, T. Ma, A. Boloor, Y. Zhu, and X. Zhang, "Learned in-sensor visual computing: From compression to eventification," in *International Conference on Computer-Aided Design*, 2023.

[21] Y. Feng, T. Ma, Y. Zhu, and X. Zhang, "Blisscam: Boosting eye tracking efficiency with learned in-sensor sparse sampling," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 1262–1277.

[22] Y. Fu, Z. Ye, J. Yuan, S. Zhang, S. Li, H. You, and Y. Lin, "Gen-nerf: Efficient and generalizable neural radiance fields via algorithm-hardware co-design," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–12.

[23] S. Girish, K. Gupta, and A. Shrivastava, "Eagles: Efficient accelerated 3d gaussians with lightweight encodings," in *European Conference on Computer Vision*. Springer, 2024, pp. 54–71.

[24] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[25] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[26] Z. Gong, F. Tosi, Y. Zhang, S. Mattoccia, and M. Poggi, "Hs-slam: Hybrid representation with structural supervision for improved dense slam," *arXiv preprint arXiv:2503.21778*, 2025.

[27] H. Gui, L. Hu, R. Chen, M. Huang, Y. Yin, J. Yang, and Y. Wu, "Balanced 3dgs: Gaussian-wise parallelism rendering with fine-grained tiling," *arXiv preprint arXiv:2412.17378*, 2024.

[28] C. Harris, M. Stephens *et al.*, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, no. 50. Manchester, UK, 1988, pp. 10–5244.

[29] H. He, G. Li, F. Liu, L. Jiang, X. Liang, and Z. Song, "Gsarch: Breaking memory barriers in 3d gaussian splatting training via architectural support," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 366–379.

[30] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[31] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2016, pp. 1271–1278.

[32] X. Huang, H. Zhu, Z. Liu, W. Lin, X. Liu, Z. He, J. Leng, M. Guo, and Y. Feng, "Seele: A unified acceleration framework for real-time gaussian splatting," *arXiv preprint arXiv:2503.05168*, 2025.

[33] M. M. Johari, C. Carta, and F. Fleuret, "Eslam: Efficient dense slam system based on hybrid representation of signed distance fields," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 17 408–17 419.

[34] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, "Design of an image edge detection filter using the sobel operator," *IEEE Journal of solid-state circuits*, vol. 23, no. 2, pp. 358–367, 1988.

[35] I. A. Kazerouni, L. Fitzgerald, G. Dooly, and D. Toal, "A survey of state-of-the-art on visual slam," *Expert Systems with Applications*, vol. 205, p. 117734, 2022.

[36] N. Keetha, J. Karhade, K. M. Jatavallabhula, G. Yang, S. Scherer, D. Ramanan, and J. Luiten, "Splatam: Splat track & map 3d gaussians for dense rgb-d slam," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21 357–21 366.

[37] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3d gaussian splatting for real-time radiance field rendering," *ACM Transactions on Graphics*, vol. 42, no. 4, pp. 1–14, 2023.

[38] B. Kerbl, A. Meuleman, G. Kopanas, M. Wimmer, A. Lanvin, and G. Drettakis, "A hierarchical 3d gaussian representation for real-time rendering of very large datasets," *ACM Transactions on Graphics (TOG)*, vol. 43, no. 4, pp. 1–15, 2024.

[39] V. Kodukula, A. Shearer, V. Nguyen, S. Lingutla, Y. Liu, and R. LiKamWa, "Rhythmic pixel regions: multi-resolution visual sensing system towards high-precision visual computing at low power," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 573–586.

[40] T. Kong, A. Yao, Y. Chen, and F. Sun, "Hypernet: Towards accurate region proposal generation and joint object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 845–853.

[41] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, "Nvidia ampere architecture in-depth," 2020. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[42] V. Kravets and A. Stern, "Progressive compressive sensing of large images with multiscale deep learning reconstruction," *Scientific reports*, vol. 12, no. 1, p. 7228, 2022.

[43] J. C. Lee, D. Rho, X. Sun, J. H. Ko, and E. Park, "Compact 3d gaussian representation for radiance field," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 21 719–21 728.

[44] J. Lee, K. Choi, J. Lee, S. Lee, J. Whangbo, and J. Sim, "Neurex: A case for neural rendering acceleration," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[45] J. Lee, J. Kim, J. Park, and J. Sim, "Vr-pipe: Streamlining hardware graphics pipeline for volume rendering," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 217–230.

[46] J. Lee, S. Lee, J. Lee, J. Park, and J. Sim, "Gscore: Efficient radiance field rendering via architectural support for 3d gaussian splatting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 497–511.

[47] C. Li, S. Li, L. Jiang, J. Zhang, and Y. C. Lin, "Uni-render: A unified accelerator for real-time rendering across diverse neural renderers," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 246–260.

[48] C. Li, S. Li, Y. Zhao, W. Zhu, and Y. Lin, "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*, 2022, pp. 1–9.

[49] S. Li, C. Li, W. Zhu, B. Yu, Y. Zhao, C. Wan, H. You, H. Shi, and Y. Lin, "Instant-3d: Instant neural radiance field training towards on-device ar/vr 3d reconstruction," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[50] X. Li, J. Jiang, Y. Feng, Y. Gan, J. Zhao, Z. Liu, J. Leng, and M. Guo, "Sltarch: Towards scalable point-based neural rendering by taming workload imbalance and memory irregularity," in *2025 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 2025, pp. 1–9.

[51] W. Lin, Y. Feng, and Y. Zhu, "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*, 2025, pp. 669–682.

[52] T. Liu, X. Song, Z. Yue, R. Wen, X. Hu, Z. Song, Y. Wen, Y. Hao, W. Li, Z. Du *et al.*, "Cambricon-sr: An accelerator for neural scene representation with sparse encoding table," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 1254–1268.

[53] Z. Liu, H. Zhu, X. Li, Y. Wang, Y. Shi, W. Li, J. Leng, M. Guo, and Y. Feng, "Voyager: Real-time splatting city-scale 3d gaussians on your phone," *arXiv preprint arXiv:2506.02774*, 2025.

[54] T. Ma, W. Cao, F. Qiao, A. Chakrabarti, and X. Zhang, "Hogeye: neural approximation of hog feature extraction in rram-based 3d-stacked image sensors," in *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 2022, pp. 1–6.

[55] S. S. Mallick, R. Goel, B. Kerbl, M. Steinberger, F. V. Carrasco, and F. De La Torre, "Taming 3dgs: High-quality radiance fields with limited resources," in *SIGGRAPH Asia 2024 Conference Papers*, 2024, pp. 1–11.

[56] H. Matsuki, R. Murai, P. H. Kelly, and A. J. Davison, "Gaussian splatting slam," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 18 039–18 048.

[57] M. H. Mubarik, R. Kanungo, T. Zirr, and R. Kumar, "Hardware acceleration of neural graphics," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–12.

[58] B. A. Mudassar, P. Saha, Y. Long, M. F. Amir, E. Gebhardt, T. Na, J. H. Ko, M. Wolf, and S. Mukhopadhyay, "A camera with brain–embedding machine learning in 3d sensors," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 680–685.

[59] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon, "Kinectfusion: Real-time dense surface mapping and tracking," in *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee, 2011, pp. 127–136.

[60] M. Niemeyer, F. Manhardt, M.-J. Rakotosaona, M. Oechsle, D. Duckworth, R. Gosula, K. Tateno, J. Bates, D. Kaeser, and F. Tombari, "Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps," *arXiv preprint arXiv:2403.13806*, 2024.

[61] P. Pham, D. Conover, and A. Bera, "Flashslam: Accelerated rgb-d slam for real-time 3d scene reconstruction with gaussian splatting," *arXiv preprint arXiv:2412.00682*, 2024.

[62] C. Rao, H. Yu, H. Wan, J. Zhou, Y. Zheng, M. Wu, Y. Ma, A. Chen, B. Yuan, P. Zhou *et al.*, "Icarus: A specialized architecture for neural radiance fields rendering," *ACM Transactions on Graphics (TOG)*, vol. 41, no. 6, pp. 1–14, 2022.

[63] K. Ren, L. Jiang, T. Lu, M. Yu, L. Xu, Z. Ni, and B. Dai, "Octree-gs: Towards consistent real-time rendering with lod-structured 3d gaussians," *arXiv preprint arXiv:2403.17898*, 2024.

[64] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, pp. 91–99, 2015.

[65] E. Sandström, Y. Li, L. Van Gool, and M. R. Oswald, "Point-slam: Dense neural point cloud-based slam," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 18 433–18 444.

[66] S. Sarangi and B. Baas, "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, 2021, pp. 1–5.

[67] X. Sheng, S. Mao, Y. Yan, and X. Yang, "Review on slam algorithms for augmented reality," *Displays*, p. 102806, 2024.

[68] Z. Song, H. He, F. Liu, Y. Hao, X. Song, L. Jiang, and X. Liang, "Srender: Boosting neural radiance field efficiency via sensitivity-aware dynamic precision rendering," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 525–537.

[69] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, 2017.

[70] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma *et al.*, "The replica dataset: A digital replica of indoor spaces," *arXiv preprint arXiv:1906.05797*, 2019.

[71] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A benchmark for the evaluation of rgb-d slam systems," in *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2012, pp. 573–580.

[72] F. Tosi, Y. Zhang, Z. Gong, E. Sandström, S. Mattoccia, M. R. Oswald, and M. Poggi, "How nerfs and 3d gaussian splatting are reshaping slam: a survey," *arXiv preprint arXiv:2402.13255*, vol. 4, p. 1, 2024.

[73] H. Wang, W. Liu, K. Chen, Q. Sun, and S. Q. Zhang, "Process only where you look: Hardware and algorithm co-optimization for efficient gaze-tracked foveated rendering in virtual reality," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 344–358.

[74] H. Wang and M. Li, "A new era of indoor scene reconstruction: A survey," *IEEE Access*, 2024.

[75] X. Wang, R. Yi, and L. Ma, "Adr-gaussian: Accelerating gaussian splatting with adaptive radius," in *SIGGRAPH Asia 2024 Conference Papers*, 2024, pp. 1–10.

[76] C. Wu, Z. Gong, B. Tao, K. Tan, Z. Gu, and Z.-P. Yin, "Rf-slam: Uhf-rfid based simultaneous tags mapping and robot localization algorithm for smart warehouse position service," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 12, pp. 11 765–11 775, 2023.

[77] L. Wu, H. Zhu, S. He, J. Zheng, C. Chen, and X. Zeng, "Gauspu: 3d gaussian splatting processor for real-time slam systems," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 1562–1573.

[78] C. Yan, D. Qu, D. Xu, B. Zhao, Z. Wang, D. Wang, and X. Li, "Gs-slam: Dense visual slam with 3d gaussian splatting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 19 595–19 604.

[79] Z. Ye, Y. Fu, J. Zhang, L. Li, Y. Zhang, S. Li, C. Wan, C. Wan, C. Li, S. Prathipati *et al.*, "Gaussian blending unit: An edge gpu plug-in for real-time gaussian-based rendering in ar/vr," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 353–365.

[80] C. Young, A. Omid-Zohoor, P. Lajevardi, and B. Murmann, "A data-compressive 1.5/2.75-bit log-gradient qvga image sensor with multi-scale readout for always-on object detection," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 11, pp. 2932–2946, 2019.

[81] V. Yugay, Y. Li, T. Gevers, and M. R. Oswald, "Gaussian-slam: Photo-realistic dense slam with gaussian splatting," *arXiv preprint arXiv:2312.10070*, 2023.

[82] C. Zhang, Y. Feng, J. Zhao, G. Liu, W. Ding, C. Wu, and M. Guo, "Streaminggs: Voxel-based streaming 3d gaussian splatting with memory optimization and architectural support," in *62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2025, pp. 1–9.

[83] J. Zhang, S. Singh *et al.*, "Loam: Lidar odometry and mapping in real-time." in *Robotics: Science and systems*, vol. 2, no. 9. Berkeley, CA, 2014, pp. 1–9.

[84] W. Zhang, Q. Cheng, D. Skuddis, N. Zeller, D. Cremers, and N. Haala, "Hi-slam2: Geometry-aware gaussian slam for fast monocular scene reconstruction," *arXiv preprint arXiv:2411.17982*, 2024.

[85] Z. Zhu, S. Peng, V. Larsson, W. Xu, H. Bao, Z. Cui, M. R. Oswald, and M. Pollefeys, "Nice-slam: Neural implicit scalable encoding for slam," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12 786–12 796.